Software Transactional Memory In Mercury

Leon Mika
Supervisor: Zoltan Somogyi

October 31, 2007

Abstract

Concurrent programming is now becoming more important than ever. But for concurrent programs to work deterministically, sections of the code must be synchronised. The most common method of synchronising code is to protect the code with locks. However, code which uses locks is difficult to write and even more difficult to debug. Locking also makes it difficult to compose large programs from smaller ones. A relatively new method of synchronisation, known as Software Transactional Memory, is promising to be a much easier method of synchronisation. This thesis describes the design and implementation of a Software Transactional Memory system in Mercury.

Contents

1 Introduction **3**

2 Mercury **4**
2.1 Mercury . . . 4
2.2 Concurrency In Mercury . . . 5

3 Software Transactional Memory **6**
3.1 Locks . . . 6
3.2 Software Transactional Memory . . . 9
3.3 Previous Work On STM . . . 11

4 Language Constructs **12**
4.1 Transaction Variables . . . 12
4.2 Atomic Scope . . . 14
4.3 Retry . . . 16
4.4 The Or_Else Operator . . . 17
4.5 Nested Atomic Scopes . . . 20
4.6 Exceptions Within the Atomic Goal . . . 21

5 Implementation **21**
5.1 Stages of the Mercury Compiler . . . 22
5.2 The Front End . . . 22
5.3 Source To Source Transformation . . . 23
5.3.1 Top Level Atomic Goal . . . 24
5.3.2 Nested Atomic Goals . . . 27
5.3.3 The Or_Else Alternatives . . . 27
5.4 Library and Runtime . . . 31
5.4.1 Transaction Variables . . . 31
5.4.2 Transaction Logs . . . 31
5.4.3 Nested Transaction Logs . . . 32
5.4.4 Validation and Commit . . . 32
5.4.5 Thread Blocking . . . 33

6 Discussion **33**
6.1 Limitations . . . 33
6.2 Future Work . . . 34

7 Conclusion **36**

8 Acknowledgements **36**

A Example Source To Source Transformation **36**

1 Introduction

Up until very recently, software developers were enjoying a "free lunch" when it came to improving the performance of their code[17]. It was possible, at one point, for programmers to dramatically increase the performance of their code by simply upgrading to a faster machine. The past few decades has seen a dramatic increase in processor clock speeds from 1 MHz processors in the 1970s to 2.4 GHz processors in 2005. But recently, CPU clock speed has stabilised: the processors of tomorrow will not be much faster than the processors of today. There are a number of reason for this, the main one being that the amount of heat processors are emitting is becoming harder to manage[15].

Because of this, processor manufacturers have adopted a different approach to increasing processor performance. The focus is now on *parallel execution*: trying to get as many threads of execution as possible running at the same time. Multi-core machines are now becoming the norm and processor manufacturers are now more concerned about how many processors they can fit on one chip rather than how many instructions each processor could execute per unit time. This shift in viewpoint will eventually have a dramatic impact on software developers, as it implies that the only way future programs can fully utilise the CPU is if they engage in concurrent execution.

The standard method of building concurrent programs is to write self-contained bits of code and run each one as a separate thread. Threads are cheaper to spawn and require fewer system resources to manage than processes[1]. However, since threads share the same address space, they need to be coordinated so that one thread does not interfere with another while it is accessing shared data. The act of coordinating threads this way is known as *synchronisation*.

The standard method of synchronising threads is by isolating a piece of code from concurrent execution. This is known as mutual exclusion and the synchronised portions of code known as the critical section[19]. The standard method of enforcing mutual exclusion is through the use of locks. When a thread wishes to execute code within the critical section, it first must acquire the lock. If the lock is free, the thread continues execution and must release the lock once it has left the critical section. If the lock is not free, the thread is blocked until the lock becomes free. Despite how simple this may sound, locks do have a number of disadvantages. Writing and debugging non-trivial code which uses locks is very difficult. Locks also prevent the development of composable code (code which is built by combining smaller bits of existing code). Since composability is a technique used often in the development of large software systems, the use of locks can further increase the complexity associated with developing, modifying and maintaining these systems.

However, recent research has led to the development of an optimistic synchronisation method known as *Software Transactional Memory*[6]. Software transactional memory is similar to transactions in database applications in that it promises atomicity through the creation of a transaction log which is used to record the reads and writes to shared memory. It is a more declarative approach to synchronisation since the programmer only needs to specify what need to be synchronised rather than how. Also, there is no need for the programmer to acquire or release locks when synchronising threads using software transactional memory. Because of this, software transactional memory permits the development of composable programs.

This thesis describes the design and implementation of a software transactional memory system in Mercury, a pure, logical and declarative language developed in the department. Section 2 gives an introduction to the Mercury language, along with a summary on Mercury's existing support for concurrency and thread control. Section 3 discusses the inadequacy of locks for composable programs. It will then introduce the principles of software transactional memory and provide information on a number of existing implementations. Section 4 introduces the language constructs we have added into the Mercury language which permits the use of the software transactional memory system. Section 5 describes how we implemented the software transactional memory system itself. Section 6 discusses the limitations of our implementation, as well as any possible extensions that can be included in the system. Finally, Section 7 summarises and concludes this thesis.

2 Mercury

This section provides a brief introduction to the parts of Mercury that is relevant to the remainder of this thesis. It also describes the current support for concurrency in the runtime. A more complete discussion of the language can be found at [8]. The material in this section has come largely from [2] and [7].

2.1 Mercury

The Mercury type system is based on a many-sorted logic type system derived from the Hindley-Milner type system. Mercury supports parametric polymorphism as well as higher-order types.

Mercury contains a strong mode system. The *mode* of a predicate is the mapping from the initial instantiation state of the predicate's arguments before the call to their final instantiation state after the return. The compiler uses this information, along with information obtained by the type checker, to check the correctness of the mode declarations. It is possible for a predicate to have more than one mode — each different mode of the same predicate is called a *procedure*. The most commonly used modes are `in` and `out`. An argument of a procedure with the mode `in` expects a ground term whilst an argument of a procedure with the mode `out` expects a free variable and is expected to bind that variable to a value before the predicate exits. As such, arguments with mode `out` can be used to return results calculated by the called predicate. The mode system also requires the producer of values to come before their consumer. The compiler will try to reorder the goals in the clause to satisfy this requirement but if no possible order can be found, the compiler rejects the program.

Mercury also has a strong determinism system. Determinism is associated with procedures and indicates how many solutions, if any, a procedure could return. The four main determinisms are "`det`", "`semidet`", "`multi`" and "`nondet`". A procedure with determinism `det` will always succeed and produce exactly one solution. A `semidet` procedure will also produce a single solution if it succeeds but can possibly fail, producing no solutions. Procedures with determinisms `multi` and `nondet` are capable of producing more than one solution. Procedures that are `nondet` can also fail whilst procedures that are `multi` must produce at least one solution. Another determinism a procedure can have is

`erroneous`. This determinism identifies procedures which do not properly return. These are mainly used by predicates that will throw an exception or will halt the program.

The mode system can also record the *uniqueness* of a term. A term is "unique" if there is only one live reference to it. A term which has no live references to it is said to be "clobbered". With these instantiations, the argument modes `di` and `uo` can be used to indicate an instantiation mapping from unique to clobbered (destructive input) and an instantiation mapping from free to unique (unique output), respectively. With uniqueness, it is possible to allow for destructive updates whilst maintaining declarative semantics. Mercury uses this method to handle I/O. Mercury views I/O predicates as describing a relationship between two states of the world: the one before the action and the one after. The difference between the initial and final I/O states includes the effect of the action, and the effects of all other actions executed by other programs since the creation of the initial I/O state. An I/O predicate uses the `di` and `uo` modes to destructively updates the state of the world. Since the world has only one state at any one time, the old states of the world can no longer be used. This prevents backtracking over predicates which perform I/O.

Mercury also supports the selection and modification of fields from a data structure without explicit construction and deconstruction. A field can be selected by using the "`^`" operator. When used in the form "A `=` B `^` f", the value of A will be set to the value of field f from the data structure stored in B. Fields can also be updated using the "`:=`" operator. When used in the form "A `=` B `^` f `:=` V", the value of A will be set to a newly constructed data structure which is the same data structure in B but with field f being set to V. Since Mercury is a single assignment language, the only way to update the value of a field is to create a new copy of it: the data structure in B has not been modified.

2.2 Concurrency In Mercury

The existing support for parallelism was designed and implemented by Thomas Conway[2] and a significant number of improvements have been made by Peter Wang[22].

Concurrency in Mercury works by dividing up the runtime into two parts: the *engine* and the *context*. An engine works on a context like a processor works on a process. The Mercury scheduler selects a context from the pool of available contexts and allocates it to an engine. The engine will work on the context until it is allocated a new one by the scheduler. Each engine is mapped onto its own thread which, if there are enough processors, is mapped onto a physical processor. The use of threads was chosen over whole processes due to their cheap creation cost and implicit shared memory properties.

There are two ways of developing concurrent code in Mercury. One way is through the use the parallel conjunction operator ("`&`"). The syntax "`A & B`" is an indication from the programmer to the compiler that the two goals `A` and `B` would benefit from parallel execution. When encountered, the compiler generates code to run goal A in a new context, whilst the parent context executes goal B. It is possible to pass information produced by goal A to goal B but this only extends to data within shared variables. Also, data between the two goals can only go one way.

For situations in which the programmer needs more control over the creation of threads and the sharing of data between them, Mercury also supports explicit thread creation.

Threads are created by the `spawn` library predicate. `spawn` takes a closure to a predicate as an argument, along with the I/O state. When invoked, `spawn` creates a new context with the predicate argument on the call stack. This predicate has access to the I/O state along with any other arguments required. Explicit threading allows the development of concurrent programs with threads that can communicate with other threads. Threads do not suffer from the restriction of shared data that the parallel conjunction operator does. However, the programmer is responsible for setting up and synchronising shared data explicitly.

3 Software Transactional Memory

3.1 Locks

The simplest and most common method of synchronisation between concurrently running threads is the use of locks[18]. Locking is used to enforce mutual exclusion: preventing all but one thread from running a piece of code (the critical section) at any one time. The idea behind synchronising with locks is that a thread must acquire a lock before executing the critical section. Once a thread has the lock, it can proceed to execute the code within the critical section with the knowledge that it is the only thread which can execute this code. Once the thread has exited the critical section, it must release the lock so that another thread can acquire it.

The principle of synchronising execution by locking sounds easy enough. However, there are many problems with using locks as a method of synchronisation. One of these is that most programming languages do not have a mechanism that associates a lock with the data it is protecting. The programmer must rely on documentation of locking protocols to determine when and under what circumstances to acquire a lock. There is no standard to specifying locking protocols: they are dependent on what the locks are protecting and how the locks are protecting it. When there are multiple locks to be acquired, the programmer must know the right order in which to acquire them. Common errors involve taking too few locks, taking too many locks, taking the wrong locks or taking the locks in the wrong order. Many of these problems can lead to possible race conditions, deadlocks or other errors. Generally, these errors are very difficult to recreate and, hence, to debug [16].

But the main problem with using locks is that it prevents the development of composable programs. Whilst it is relatively easy to develop small pieces of concurrent code which may work correctly within themselves, combining them can produce unexpected results and may also introduce deadlocks or race conditions.

To demonstrate the pitfalls of building programs with locks, we will step through the development of a simple banking program in Mercury[1]. The banking program makes use of an account module which contains the predicates `deposit` and `withdraw`. These predicates are responsible for atomically adjusting an account's balance. They maintain this atomicity by protecting the synchronous code with Mercury semaphores (which is the locking mechanism provided by Mercury).

```
:- pred deposit(int::in, account::in, account::out, io::di, io::uo) is det.
```

[1]This example is adapted from the example in [10].

```
deposit(Amount, !Account, !IO) :-
    Lock = !.Account ^ lock,
    semaphore.wait(Lock, !IO),
    Balance0 = !.Account ^ balance,
    Balance = Balance0 + Amount,
    !:Account = !.Account ^ balance := Balance,
    semaphore.signal(Lock, !IO).


:- pred withdraw(int::in, account::in, account::out, io::di, io::uo) is det.

withdraw(Amount, !Account, !IO) :-
    deposit(-Amount, !Account, !IO).
```

Let us assume that the lock associated with an account, along with the account's balance, is private to the account module. Since the code modifying the account's balance is protected by the account's lock, we can safely state that only one thread can deposit money into the account at any one time. The same can be said of the withdraw predicate as it inherits its atomicity properties from the deposit predicate.

We will now assume that the banking program needs to handle transfers between accounts. We will also assume that transfers are frequent and that the banking program should be designed to perform as many transfers concurrently as possible. Each transfers must also be atomic as it would be extremely undesirable to have a concurrently running thread interfere with the transfer of money between two accounts. With these requirements in mind, one might produce the following code:

```
:- pred transfer(int::in, account::in, account::out,
        account::in, account::out, io::di, io::uo) is det.

transfer(Amount, !AccountA, !AccountB, !IO) :-
    withdraw(Amount, !AccountA, !IO),
    deposit(Amount, !AccountB, !IO).
```

The writer of this code might argue that since the calls to withdraw and deposit are atomic, it would not be possible for another thread to view the intermediate balance of the accounts during the transfer. This statement, unfortunately, is not true: it is possible for another thread to view the state of the accounts straight after the call to `withdraw` but before the call to `deposit`. As the intermediate state of the accounts is unprotected, the `transfer` predicate contains a race condition and, hence, would not be safe to use in a multi-threaded application.

Here is an example on how severe the problem can be. Suppose that another concurrently running thread is conducting an audit by calculating the total balance across all accounts while a transfer is occurring. If the totaling thread preempts the transfer thread just after the return from the withdraw predicate, the totaling thread will view the intermediate states of the accounts. When the transfer predicate has completed the

transfer, the total balance calculated by the totaling thread would be inconsistent with the actual total balance.

The next step then would be to protect the transfer operation itself by using a dedicated transfer lock. If a transfer needs to be made, the thread must acquire this lock before modifying the accounts. This approach, however, will make transfers sequential: it would not be possible to perform two transfers concurrently, even if they operate on completely separate accounts. Therefore, the next step might be to move the responsibility of acquiring the locks from the deposit predicate to the transfer predicate:

```
:- pred transfer(int::in, account::in, account::out,
        account::in, account::out, io::di, io::uo) is det.

transfer(Amount, !AccountA, !AccountB, !IO) :-
    get_account_lock(LockA, !.AccountA),
    get_account_lock(LockB, !.AccountB),
    semaphore.wait(LockA, !IO),
    semaphore.wait(LockB, !IO),
    withdraw(Amount, !AccountA, !IO),
    deposit(Amount, !AccountB, !IO).
    semaphore.signal(LockB, !IO),
    semaphore.signal(LockA, !IO).
```

This approach is not without problems either. Although a single transfer from account A to account B will work as expected, a deadlock can occur when a transfer from account B to account A is also running concurrently. Suppose thread 1 is performing the transfer from account A to account B and thread 2 is performing the transfer from account B to account A. A context switch to thread 2 can occur just after thread 1 has acquired the lock for account A. Thread 2 would then acquire the lock for account B but would not be able to proceed as it cannot take the lock for account A. It, therefore, waits at this step until, eventually, a context-switch occurs and thread 1 resumes. However, since thread 2 is holding the lock for account B, thread 1 cannot continue either. Thus, this solution introduces a deadlock. Furthermore, the deadlock would be difficult to find as it would only occur when a transfer from A to B and a transfer from B to A are concurrently occurring: under all other circumstances the transfer predicate will work as expected. The deadlock can be fix by acquiring the locks in an arbitrary but consistent order.

This solution also increases the coupling between the two modules. The writer of the account module must now make the locks public so they can be used within the transfer predicate. The writer must also provide detailed documentation on how to use the locks (the locking protocol) and must now consider the users of the locks when changing the synchronisation method. For the writer of the transfer predicate, they must now know the locking protocol and must review it whenever the account module changes.

Suppose now that the banking program needs to handle a sequence of transfers atomically: a so called “bulk-transfer”. Imagine how such a predicate would be implemented. The writer will need to move the responsibility for acquiring the account locks from the transfer predicate to the bulk transfer predicate. What happens if the banking program needs to perform a number of these bulk transfers atomically (as a “bulk-bulk-transfer”)?

Although it is unlikely that a banking program will need to develop an atomic bulk-bulk-transfer predicate, the message still remains: Locks do not compose! Any attempt to do so can, at best, increase the complexity of writing and maintaining the program and, at worst, can introduce subtle bugs which are difficult to find and debug. We will return to this banking example when we introduce the software transactional memory constructs we have implemented.

3.2 Software Transactional Memory

We will now introduce Software Transactional Memory (STM) which is a new approach to synchronising threads without the use of explicit locking. It promises to be very useful in situations where composability is important and, in some implementations, can be as quick or even quicker than lock based synchronisation.

Software transactional memory was first introduced by Maurice Herlihy and J. Eliot Moss as a method of synchronising memory in multiprocessors[9]. Since then, a number of software implementations have been developed, either as libraries or as language constructs. Software transactional memory is an example of an *optimistic synchronisation technique*[6]. Instead of isolating the critical section from parallel execution, it allows any number of threads to execute the critical section (call the atomic code or atomic block) at any one time. The synchronicity properties of software transactional memory come from the rules governing the access to and modification of *transaction variables*.

A transaction variable is a special form of shared variable that can only be accessed or changed by the software transactional memory system itself. Threads that need to read or write to the transaction variables can do so through special calls which can only be made when a transaction log is present. This transaction log is created and given to a thread when it enters the atomic code. Once inside the atomic code, the thread can query or change the value of transaction variables using the relevant function calls. Whilst it appears to the thread that the actual values of the transaction variables are being read or modified, these calls actually work on local copies of these values. These local copies are stored within the transaction log and returned when the read function is called and modified when the write function is called.

Once execution leaves the atomic code, the transaction log is validated. Validation of a transaction log involves going through each entry in the log and determining whether or not the associated transaction variable has changed since first accessed. Validation is important as the transaction log may not see a consistent view of memory. This happens when the locally updated values of the transaction variables within the log are different from the actual values of the transaction variables. The usual case of this happening is when the transaction variables have been changed by another, concurrently running, thread. If this is the case, the transaction log is discarded and the atomic code is re-executed. If no transaction variable that has been viewed within the atomic code has had its value changed, the transaction log is deemed valid and is, therefore, committed. Committing a transaction log involves going through all the records within the log and changing the real values of the transaction variables to the locally updated values stored in the entry which refers to this transaction variable.

The act of validating and committing a transaction log must, in itself, be an atomic operation as validating or committing two transaction logs which share the same trans-

actions variables leads to a race condition. For example, transaction 1 is being validated and transaction 2 is being committed. Both transactions have seen the same set of transaction variables. If the value of a transaction variable has been changed by transaction 2 just after transaction 1 has checked the same transaction variable, the validation result of transaction 1 will be incorrect.

This whole operation does not require the programmer to use any locks, which alleviates many of the pitfalls associated with developing synchronous code using locks. The absence of locks also means that more than one thread can execute the critical region at once, potentially increasing the parallelism of the written program[10].

A transaction must adhere to the first three ACID properties to adhere to correct synchronous behaviour[14].

Atomicity Changes made to shared memory by a transaction must be revealed all at once or not at all. No thread can view the intermediate state of a transaction being committed.

Consistency Transactions must operate on a consistent view of of memory. When they commit, they must also leave memory in a consistent state.

Isolation The effects of a transaction are invisible from all other concurrently running transactions until the transaction commits.

Durability deals with ensuring that the changes made by a transaction persist even after a system failure. Since software transactional memory controls access to volatile memory, this requirement is not applicable.

Atomicity is guaranteed as long as the act of validating and committing a transaction log is protected from interference from other, concurrently running threads[10]. This means that before a transaction is validated or committed, it must acquire a lock of some sort (either a global lock or a fine-grain locking mechanism which protects the transaction variables seen by the transaction).

Consistency is guaranteed as long as every transaction log is validated before it is committed. A transaction can only be committed if what it assumes about the value of transaction variables is correct. If this is not the case, the transaction must be re-executed. Consistency also requires that the act of committing a transaction log cannot be interrupted.

Isolation can be assured to a reasonable degree if the transaction log is local to the associated thread. Since the reads and writes do not operate on memory directly, we can ensure that each running transaction cannot see the changes made by another transaction and that other transaction can not see changes made by this transaction. However, it is possible to improve isolation by forbidding the use of functions which will produce side effects within the atomic code. Since transactions can be re-executed many times, we should not invoke code which causes irreversible side-effects (for example: deleting a file or launching a nuclear warhead)[10]. This can be handled in declarative, strongly typed languages by using the type system, especially if the type system is used to control the use of code which will produce side effects.

3.3 Previous Work On STM

There have been a number of implementations of software transactional memory in several different languages. Many of them involve extending the syntax of the language in some way so that the development of synchronous code can be done easily.

The implementation which was used as the basis for this work was of a software transactional memory system for Concurrent Haskell[6]. This was implemented by Tim Harris along with associates from the Microsoft Research Center in Cambridge, UK. Concurrent Haskell is a dialect of Haskell which permits the explicit forking of threads and the communication of data between them. This implementation introduced the STM monad that must be present when any operation needs access to the transaction log. The STM monad also forbids the invocation of I/O functions within the atomic code: I/O needs to be prevented within a transaction as it is unknown how many times the atomic code will be executed. The function which produces an I/O action from a valid transaction log is the "atomic" function. This function takes a STM action and runs it with a transaction log. If the resulting log is valid, the atomic function returns an I/O operation which represents the act of committing the transaction log to memory. This implementation has also introduced a method of forcibly retrying a transaction (blocking if necessary) and introduced the execution of alternative code if one of the transactions retries. This implementation is now part of the standard library for the Glasgow Haskell Compiler [20].

Tim Harris has also implemented is a software transactional memory system for the Java Virtual Machine [3]. In this implementation, the software transactional memory system was used to support Hoare's Conditional Critical Regions — a language construct which is similar to the "atomic" function in [6]. Since the type system in Java is weaker than the type system in Haskell, the software transactional memory system was developed to support the synchronisation of any memory location without explicit directions to do so (i.e. accesses and modifications to any memory location can be added to the transaction log). This was done by introducing two additional data structures into the virtual machine. The first data structure is used to map the addresses of values on the heap that have been accessed or modified within a Conditional Critical Region to the transaction. It is also used to indicate the version of the data held at that address: when the value at that address changes, the version number is incremented. The second data structure keeps information about the transaction itself, including the status of the log and any modifications to the heap that were caused by running the atomic code.

A similar approach to adding software transactional memory at the language level has been done for C++[13]. This implementation involved the addition of a new language construct into the syntax of the language (the "atomic" scope) and the logging of arbitrary locations from the heap. The approach taken to handle this was to develop a pre-processor and a post compilation analyser. To avoid the complexity of changing the C++ compiler itself, the pre-processor was developed to search for any instances of the atomic scope and replace it with function calls and control structures implementing the necessary operations of the software transactional memory system. The output from the pre-processor is passed to the standard C++ compiler, which is configured to generate assembly code from the source. The assembly code is then analysed by the program TARIFA (Transactions by Assembly Instrumentation Framework) which searches the assembly code, replacing any

instructions accessing memory directly within the atomic scope to equivalent instructions which read from or write to a transaction log instead. Like [3] it does not restrict the logging of memory accesses or modifications to values of a specific type.

There is also an implementation of a system similar to software transactional memory in the Scheme language[11]. This implementation involves the creation of *proposals*, which are similar to transaction logs. These proposals are validated and committed by calling one of the four available atomic functions (each one commits in a slightly different way). Since Scheme is untyped, the programmer needs to explicitly state which operations should be logged to the proposal (e.g. to return the head of a list and log it to a proposal, you call "provision-car" instead of "car"). There is also no support for forcing the re-execution of transaction or running alternative code when a transaction needs to retry.

4 Language Constructs

The main requirement for a usable software transactional memory system for Mercury is the addition of language constructs that can be used by the programmer to write synchronous code. This involves adding new syntax elements to the language itself as well as adding public types and predicates to the standard Mercury library. In this section we introduce the language constructs we have added to the Mercury language to provide the use of software transactional memory to Mercury programmers. Since there exists a tight inter-dependence between the language construct and the public library predicates, we first describe the system developed to represent data that can be shared and synchronised using the software transactional memory constructs, followed by the new language constructs themselves. Throughout this section we will be referring to the banking example in section 3 to demonstrate the use of these new constructs.

4.1 Transaction Variables

For software transactional memory to have any real utility, we need a method of representing shared data. Mercury already has a few primitive data types which permit the sharing of data between concurrently running threads (such as *mutable variables*). However, the synchronicity properties of software transactional memory come from the act of logging all reads and writes to shared memory in a transaction log and validating the log before the actual changes are committed to memory. Thus, the shared data synchronised by software transactional memory can only be changed with the presence of a transaction log. We, therefore, introduce *transaction variables* into the Mercury language. These are variables that can be used to share data across concurrently running threads but can only be accessed and modified with the presence of a transaction log. Our design of transaction variables is based upon the design of transaction variables in [6].

A functional requirement of transaction variables is that they support values of any valid Mercury type. Naturally, we do not want to restrict the synchronisation of data using the software transactional system to only one type. Thus, we must make the transaction variable polymorphic. However, we still want to restrict how the value that is held within the transaction variable is read or modified.

Mercury supports explicit parametric polymorphism: the ability to define data structures which can accept values of any type but can still maintain a strong type system [12]. This can be expressed in programs through the use of type variables. With type variables, data structures and predicates which can take values of any valid Mercury type can be declared. Using this mechanism, we have designed transaction variables as a wrapper type: a type that can hold a value of any type but requires the explicit calls to associated predicates to extract that value.

The following is the declared type of transaction variables.

```
:- type stm_var(T).
```

and the following are the associated predicates which operate on transaction variables:

```
:- pred new_stm_var(T::in, stm_var(T)::out, io::di, io::uo) is det.

:- pred read_stm_var(stm_var(T)::in, T::out, stm::di, stm::uo) is det.

:- pred write_stm_var(stm_var(T)::in, T::in, stm::di, stm::uo) is det.
```

The transaction variable type is declared as an *abstract type*: the definition of the transaction variable type is hidden from the user. Because of this, terms which have the type `stm_var` cannot be directly constructed or de-constructed in user code. The main reason for this is that the transaction variable type is defined in the runtime. But the abstract data type definition also ensures that the value of the transaction variable cannot be directly modified by the user.

The predicate `new_stm_var` creates a new transaction variable and returns its reference. The newly created transaction variable will be initialised to the value passed in as the first argument. Creating a transaction variable changes the state of the world (as the transaction variable is explicitly allocated) and, as such, `new_stm_var` needs to take a pair of I/O variables.

The predicates `read_stm_var` and `write_stm_var` are used to access and update the value of the transaction variable itself. The transaction log needs to be accessible to these predicates as "reading" from or "writing" to a transaction variable involves adding an entry to the log. The act of reading or writing to a transaction variable destroys the previous version of the log. This is necessary as we cannot allow the user to read or modify a transaction variable without adding the associated action to the log. The destruction of the previous log and creation of the new log means that the transaction log needs to be passed through these predicates as the state of the world is passed through predicates which perform I/O (but, unlike the I/O state which carries nothing, the transaction log actually contains data).

Creating transaction variables is very easy as it only consists of a call to `new_stm_var` with the initial value of the transaction variable as the first argument. Returning to the banking example, assuming there is a call which produces a new `account` data structure with a default balance, etc. (but without the associated lock — they are no longer needed), the code to create a new account transaction variable could look like the following:

```
:- pred account_init(stm_var(account)::out, io::di, io::uo) is det.

account_init(AcountTVar, !IO) :-
    new_account_with_default_balance_etc(Account),
    new_stm_var(Account, AccountTVar, !IO).
```

The original value of the account can now be accessed and modified simply by extracting the current value of the account from the associated transaction variable, making any necessary modifications and then writing it back to the transaction variable. Using this approach, the code for `deposit` and `withdraw` can be rewritten to do this. Notice that the predicates `deposit` and `withdraw` now need access to a transaction log to perform their designed operations.

```
:- pred deposit(int::in, stm_var(account)::in,
        stm::di, stm::uo) is det.

deposit(Amount, AccountTVar, !STM) :-
    read_stm_var(AccountTVar, Account0, !STM),
    Balance0 = Account0 ^ balance,
    Balance = Balance0 + Amount,
    Account = Account0 ^ balance := Balance,
    write_stm_var(AccountTVar, Account, !STM).

:- pred withdraw(int::in, stm_var(account)::in,
        stm::di, stm::uo) is det.

withdraw(Amount0, AccountTVar, !STM) :-
    Amount = -Amount,
    deposit(Amount, AccountTVar, !STM).
```

Notice also that the deposit and withdraw predicates no longer need to return a new transaction variable. Remember that `AccountTVar` contains a reference to the transaction variable, not the transaction variable itself. The predicates `read_stm_var` and `write_stm_var` modify the value of the transaction variable but the reference remains the same.

4.2 Atomic Scope

Now that we have a notion of transaction variables, we will need a method of identifying the goals which will be executed synchronously. In this section we introduce the atomic scope we have added for this very purpose.

The atomic scope is a new syntactic construct we have introduced into the language for the sole purpose of indicating goals that are to be run atomically against all other running threads. It is designed as a binary postfix operator, similar in syntax to the existential quantifier operator (“`some`”). It takes two arguments, the first argument being a list of parameters and the second argument being the atomic goal itself. This list of parameters

$$
\begin{array}{rcl}
\textit{atomic_goal} & \rightarrow & \texttt{atomic}\ [\ \textit{param_list}\ ]\ (\ \textit{goal}\ ) \\
\textit{param_list} & \rightarrow & \textit{param}\ \ \textit{param_list} \\
 & \rightarrow & \epsilon \\
\textit{param} & \rightarrow & \texttt{outer}\ (\ \textit{variable}\ ,\ \textit{variable}\ ) \\
 & \rightarrow & \texttt{inner}\ (\ \textit{variable}\ ,\ \textit{variable}\ )
\end{array}
$$

Figure 1: Syntax for the atomic scope in Backus-Naur form

must contain the parameters `outer` and `inner`, with each one taking a pair of variables. A formal definition of the syntax is presented in Figure 1.

The `outer` parameter specifies the variables which hold the state of the world in the surrounding scope. Therefore, these variables must be of type `io`[2]. Since the atomic predicate needs to modify the state of the world, the left variable has the mode `di` and the right variable has the mode `uo`. Furthermore, these variables cannot be used within the atomic subgoal.

The `inner` parameter specifies the variables that would hold the initial and final value of the transaction log. Both of these variables must be of type `stm` and, like the variables in the `outer` parameter, also have mode restrictions with the left variable having the mode `uo` and the right variable having the mode `di`.

The two parameters are used to describe the interface between the outer I/O state and the inner STM state. A similar interface was needed for the Haskell implementation in [6]. This was handled using Haskell's monad system: the `atomic` function takes a STM action and returns an I/O action which represents all the actions performed when committing the transaction log. We have adopted a similar semantics with the atomic scope but have modified it to fit the logic paradigm of Mercury. We require the user to declare the variables that represent the outer and inner states and use Mercury's strong mode system to ensure that these states remain separate. However, since the atomic goal needs to modify the state of the world, the outer state and the inner state need to interact. At the start of the atomic subgoal, the left variable in the `outer` parameter (which contains the current I/O state) is used to create the new STM state (which is stored in the left variable in the `inner` parameter). Once the atomic subgoal is complete, the final STM state is used to update the state of the world, producing a new I/O state. Therefore, it is possible to view the atomic goal as one large I/O operation with the final I/O state being the initial I/O state plus any modifications made to the state of the world when the transaction log is committed (plus any other changes to the state of the world by external entities).

Referring back to the banking example, we can now write the transfer predicate using the deposit and withdraw predicates, whilst keeping the entire operation atomic.

```
:- pred transfer(int::in, stm_var(account)::in, stm_var(account)::in,
        io::di, io::uo) is det.
```

[2] We will show an exception to this rule in section 4.5.

```
transfer(Amount, AccountA, AccountB, IO0, IO) :-
    atomic [outer(IO0, IO), inner(STM0, STM)] (
        withdraw(Amount, AccountA, STM0, STM1),
        deposit(Amount, AccountB, STM0, STM)
    ).
```

As reads and writes do not affect the actual values of transaction variables until the transaction log commits, it is safe for another thread to use the transaction variable of account A or account B while a transfer is taking place. For example, assume that thread 1 is transferring money from account A to account B. During the transfer, thread 2 deposits some money into account A. The record of the deposit only appears within the transaction log of thread 2: the actual value of the transaction variables have not changed. If thread 1 manages to finished the transfer before thread 2 has reach the end of its atomic scope, thread 1's transaction log would pass validation and would be committed to memory. Once the act of committing thread 1's transaction log has finish thread 2's transaction log would have an inconsistent view of memory. Thus, thread 2 will need to perform the deposit again but will do so with the balance of account A after the transfer.

4.3 Retry

In some circumstances, it is necessary to terminate a running transaction and restart it with a new transaction log. This may be necessary in circumstances when a transaction can not continue with the values of one or more transaction variables as they do not meet a required condition. For example, suppose the writer of the account module needs to include a withdraw predicate which ensures that the account will not be overdrawn after the call. This predicate may exist in a transaction in which every withdrawal must leave the balance of the account in the positive. The designer of the account module might, therefore, assume that if an overdraw does occur, the fault was caused by operations earlier in the transaction. Hence, the new withdraw predicate should throw away the current log and retry the transaction, with a more recent copy of the transaction variables.

We therefore included the `retry` predicate for this purpose. We could use this to continue retrying the transaction until the account has a balance which will permit the withdraw. However, continually retrying a transaction until the value of a transaction variable satisfies some required condition amounts to busy waiting which is a dreadful waste of resources. Therefore, we have adopted the technique in [6] and modified the retry predicate to wait on all the transaction variables seen so far and block the thread until any one of these transaction variables have been changed.

The declaration of retry is

```
:- pred retry(stm::ui) is erroneous.
```

The retry predicate must be given the current transaction log. Once called, it will first validate the transaction log to see if the value of any transaction variables have changed. If the transaction log is valid (i.e. the transaction variables seen by the log have not changed), the thread is blocked until at least one value of the transaction variable

is changed by another transaction (there is no point in retrying straight away as the transaction will simply retry again). If the transaction log is invalid, there is a chance that the transaction variables are already in the state necessary to allow the transaction to continue to the end. Hence, the transaction is retried immediately.

Returning to the banking example, the "withdraw-with-no-overdraw" predicate can be written to withdraw money from an account but force a retry if the withdraw causes an overdraw:

```
:- pred withdraw_with_no_overdraw(int::in, stm_var(account)::in,
        stm::di, stm::uo) is det.

withdraw_with_no_overdraw(Amount0, AccountTVar, !STM) :-
    read_stm_var(AccountTVar, Account),
    Balance = Account ^ balance,
    ( Amount0 > Balance,
        retry(!.STM)
    ;
        Amount = -Amount,
        deposit(Amount, AccountTVar, !STM)
    ).
```

After the call to this predicate, subsequent predicates be can assured that the value of the balance is positive.

4.4 The Or_Else Operator

The composable nature of software transactional memory encourages users to develop libraries of predicates which can be used within the atomic scope. However, in some circumstances, it might be that retrying the transaction is not the desired behaviour: an alternative branch of execution would be more beneficial. For example, the writer of the transfer predicate may want to ensure that a transfer does not leave an account overdrawn. For that, the writer can use the `withdraw_with_no_overdraw` predicate. But instead of blocking the thread and restarting the transaction when an overdraw would occur, the writer may want to simply do nothing except raise an error message indicating this to the user of the system.

Therefore, we introduce the `or_else` operator which allows the user to specify alternate blocks of code to execute when the previous block raises a retry.

The or_else operator is an infix binary operation with operands that can be any valid Mercury goal. Furthermore, the right goal can be another or_else operator, permitting the development of a chain of alternatives. The or_else operator associates to the left so that the first goal will execute first. The decision behind this is to make the or_else operator similar to other Mercury operations (such as the disjunction) which the user will know about.

The or_else operator is designed to support the coding of a linear list of alternatives. When a list or or_else alternatives is encountered, the first alternative is executed. If the first alternative retries, the second alternative is executed. If the second alternative

```
atomic [outer(IO0, IO), inner(STM0, STM)] (
    common_code(STM0, STM1),
    (
        main_transaction(STM1, STM)
    or_else
        alt_transaction(STM1, STM)
    )
)
```

Figure 2: Invalid use of the or_else operator.

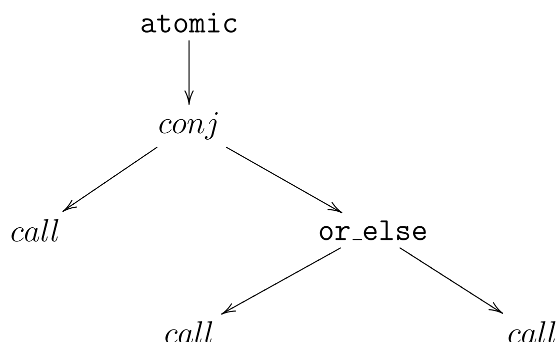

Figure 3: Parse tree generated from the code in figure 2. The or_else operator node is not directly below the atomic node as it should be.

retries, the third alternative (if present) is executed. This continues until the last alternative is encountered. If the last alternative also retries, the entire transaction retries and the first alternative is executed again.

There are two syntactic restrictions to the use of the or_else operator. The first is that the or_else operator can only appear within an atomic scope. Since the or_else operator is used to indicate the execution of an alternative transaction when the first transaction raises a retry, the only goals that the or_else operator can accept are memory transactions. The or_else operator has no semantic purpose outside an atomic scope and, as a result of this, is not permitted outside an atomic scope.

The second restriction is that if an or_else operator does appear within an atomic scope, it must be the least binding operator. That is, when the parse tree of the program is created, the or_else operator can only appear as a child node of the atomic scope node (or as a child of another or_else operator): it cannot exist in any other part of the parse tree. An example of the incorrect use of the or_else operator is in figure 2 with the parse tree presented in figure 3. Since the or_else operator is not a direct node of the parse tree, the program is rejected by the compiler. A version of the same piece of code which uses the or_else operator correctly is presented in figure 4.

We return to the banking example to show how to implement the transfer predicate to emit a error message when the transfer from account A to account B would overdraw the balance in account A. We are assuming that the writer of the transfer predicate is using

```
atomic [outer(IO0, IO), inner(STM0, STM)] (
    common_code(STM0, STM1),
    main_transaction(STM1, STM)
or_else
    common_code(STM0, STM1),
    alt_transaction(STM1, STM)
)
```

Figure 4: Equivalent code to figure 2 with the or_else operator used correctly.

the “withdraw-with-no-overdraw” predicate defined in the previous section. Therefore, the code for the transfer predicate can be as follows

```
:- pred transfer(int::in, stm_var(account)::in, stm_var(account)::in,
        io::di, io::uo) is det.

transfer(Amount, AccountA, AccountB, IO0, IO) :-
    atomic [outer(IO0, IO1), inner(STM0, STM)] (
        withdraw_with_no_overdraw(Amount, AccountA, STM0, STM1),
        deposit(Amount, AccountB, STM0, STM),
        AccountOverdrawn = no
    or_else
        STM = STM0,
        AccountOverdrawn = yes
    ),
    (
        AccountOverdrawn = no,
        IO = IO1
    ;
        AccountOverdrawn = yes,
        io.write_string("Error: Transfer would leave account A overdrawn\n",
            IO1, IO)
    ).
```

The `withdraw_with_no_overdraw` predicate will still call `retry` when the transfer overdraws account A. However, the request will be caught and the second branch of the or_else operator will be executed instead of the transaction retrying.

The or_else operator provides the user with a level of composability over synchronisation that locks could not provide. For the banking example, we could implement a predicate `maybe_withdraw` which will attempt to withdraw money from an account but if an overdraw occurs, it will do nothing and return a failure indicator. This predicate can use the `withdraw_with_no_overdraw` predicate to determine if an overdraw would have occurred.

The use of the or_else operator in this way permits the writer of library predicates to defer the decision to block to the caller. This could be useful to design non-blocking versions of the blocking predicates: predicates which behave like the blocking predicates if

the execution condition is sound but might silently fail or return a default value when the execution condition requires a retry. Using the blocking predicates in these non-blocking predicates cuts down on the amount of duplicated code and simplifies the maintenance of the system.

For the banking example, we can use this technique to implement the `maybe_withdraw` predicate which may look like this:

```
:- pred maybe_withdraw(int::in, stm_var(account)::in,
        bool::out, io::di, io::uo) is det.

maybe_withdraw(Amount0, AccountTVar, Success, IO0, IO) :-
    atomic [outer(IO0, IO), inner(STM0, STM)] (
        withdraw_with_no_overdraw(Amount0, AccountTVar, STM0, STM),
        Success = yes
    or_else
        STM0 = STM,
        Success = no
    ).
```

4.5 Nested Atomic Scopes

We have added nested atomic scopes as a method to increase the composability of the software transactional memory constructs, particularly the `or_else` operator. The use of the nested atomic scopes permits the use of the `or_else` operator in predicates which do not have access to the I/O state. Nested atomic scopes are not needed for code generation: they are included to increase programmability.

Nested atomic scopes are essentially like top-level atomic scopes except for a few major differences. The first difference is that the type of the variables listed in the `outer` must be of type `stm` and are used to represent the outer transaction log. Despite the difference in the types, the semantics of the nested atomic scope is similar to top-level atomic scopes. The initial value of the transaction log within the subgoal of the nested atomic scope is created from the current version of the surrounding atomic scope. Once the inner atomic scope is completed, it would modify the state of the outer transaction log and will return a new version of the transaction with is the old transaction log with the appropriate changes made.

The second reason deals with how nested atomic scopes are implemented. Because of reasons we will explain later on, a retry request or rollback signal within a nested atomic scopes with only one subgoal (i.e. no or_else operators) is not caught by the nested atomic scope. Instead, it is propagated up to the top level atomic scope, forcing the entire transaction within the top level atomic scope to be re-executed. This does not occur for nested atomic scopes which contain the or_else operator as this would violate the semantics of `or_else` as well as result in a large amount of repeated work (all the common code that appears before the nested atomic scope would need to be redone when the nested atomic scope passes retries and rollback to the top level atomic scope).

Using nested atomic scopes, we can rewrite `maybe_withdraw` to accept the STM state instead of the I/O state:

```
:- pred maybe_withdraw(int::in, stm_var(account)::in,
        bool::out, stm::di, stm::uo) is det.

maybe_withdraw(Amount0, AccountTVar, Success,
        OuterSTM0, OuterSTM) :-
    atomic [outer(OuterSTM0, OuterSTM), inner(STM0, STM)] (
        withdraw_with_no_overdraw(Amount0, AccountTVar, STM0, STM),
        Success = yes
    or_else
        STM0 = STM,
        Success = no
    ).
```

Now the `maybe_withdraw` predicate can be used within transactions. The writer of the transfer predicate now has access to a blocking version and a non-blocking version of a withdraw predicate which withdraws money from an account whilst ensuring the balance of the account remains positive.

4.6 Exceptions Within the Atomic Goal

Mercury has an exception system which programmers can use to indicate unexpected behaviour. Exceptions are raised with the `throw` predicate and are caught by the `try` predicate. Raising an exception will cause execution to jump to the last `try` predicate encountered.

We have added support to permit the raising of exceptions within the atomic scope. However, we have designed the atomic scope to catch any exception that has been raised within the atomic scope. The main reason for this is that the software transactional memory system uses exceptions to indicate a retry or a rollback due to an invalid transaction log. But the atomic scope also catches exceptions as it might be that the inconsistencies of the transaction log could have caused the exception. Thus, when an exception is caught by the atomic scope, the transaction log is first validated. If the transaction log is valid, the software transactional memory system assumes that the exception was thrown for reasons other than the state of the thread and it, therefore, passes it up to the next handler. If the transaction log is invalid, the transaction is discarded and the transaction is re-executed.

5 Implementation

The complete implementation of the software transaction memory constructs requires changes to both the compiler and the standard library. The changes to the compiler involve the addition of the new language constructs into the internal representation of the program, the implementation of any semantic checks that are required and, finally, the implementation of a pass which converted the software transactional memory constructs into lower level Mercury goals. We also added a new library module into the Mercury standard library which contains the primitive operations for the software transactional memory system. This section describes, in detail, the implementations we have made.

5.1 Stages of the Mercury Compiler

The following is a brief introduction to the areas of the Mercury compiler relevant to the remainder of this section. A more detailed description of the design of the compiler can be found at [21].

The compiler can be divided up into three sections: the front end, the middle end and the back end. The front end consists of the lexical analyser, the parser and the semantic checkers. The front end is also responsible for building the High Level Data Structure (HLDS). The HLDS is a simplified, internal representation of the program being compiled and is used by the rest of the front end and the middle end. Once the HLDS has been constructed, it goes through the semantic checkers. Each semanic check performs a complete traversal over the HLDS (known as a pass), reporting any semantic errors that might be in the program being compiled and annotating the HLDS with the information the checker is responsible for gathering.

If the program is semantically correct, the annotated HLDS is passed on to the so called "middle end" which performs a number of simplifications and optimisations to the HLDS. After that, the HLDS enters the back end which uses it to generate code in the target language.

5.2 The Front End

We need a way of representing the atomic scope, along with any or_else operators, in the HLDS. One possibility is to have a separate goal type for each language construct: one for the atomic scope and one for the or_else operator. However, since the or_else operator can only appear within an atomic scope, we use one goal type to represent both constructs simultaneously. We therefore added a new goal type to the HLDS, called the *atomic goal*, which is used to represent a single atomic scope along with any or_else operators that exist within it.

The information that the atomic goal is responsible for keeping is:

- The two variables listed in the `outer` parameter.
- The two variables listed in the `inner` parameter.
- Whether the atomic goal is a top level atomic goal or a nested atomic goal.
- The main atomic subgoal.
- Any or_else alternatives, stored as a list of subgoals.

The two outer variables, two inner variables, main atomic subgoal and list of or_else alternatives can be taken directly from the parse tree: an atomic scope must contain the `outer` and `inner` parameters and must also contain a valid Mercury goal which may consist of one or more or_else alternatives. Whether the atomic goal is a top level atomic goal or a nested atomic goal depends on the types of the outer variables. If the outer variables are of the type `io`, the atomic goal is a top level atomic goal. If the outer variables are of the type `stm`, the atomic goal is a nested atomic goal. These are the only two possibilities as the program will be semantically incorrect if the outer variables are

of any other type (and would not pass the type checker). Since we rely on information gathered from the type checker, we can only determined the value of this attribute once the type checker has completed.

The atomic goal has the determinism `cc_multi`. This determinism indicates that the goal can possibly return more than one solution but we are only concerned about one. There are two reasons for this: one semantic and one technical. The semantic reason is that the execution of the transaction itself can be viewed as non-deterministic: we are unsure about how many times the transaction will be executed until it is actually executed. Each time the transaction executes, we produce a new transaction log. However we do not commit to any transaction log except for the one that has been deemed valid and has been committed. Hence, the "solution" produced by the atomic goal is the committed transaction log (along with the value of any variables the transaction binds). The technical reason is that the transformations for the atomic goal include a number of calls to predicates which have the determinism `cc_multi`.

5.3 Source To Source Transformation

Now that we have the atomic goal, we need to modify the compiler to generate code for it. Code generation is mostly handled in the back end, where the HLDS is converted into code in the target language. We could have modified the back end so that the code for the atomic goal can be generated this way. However, since the compiler can generate code for multiple target languages, this would lead to a lot of work. We have elected, therefore, to take a simpler approach and apply a transformation algorithm over the HLDS which converts all instances of the atomic goal into an equivalent set of lower level HLDS goals. The act of transforming the HLDS from one representation to another is known as a "source to source transformation". The compiler passes which transform the HLDS this way exist in the Middle End of the compiler and operate on a HLDS that is deemed semantically correct and properly annotated. Therefore, our next modification to the compiler was the addition of another pass to the Middle End which converts (or "expands") all instances of the atomic goal in the HLDS into an equivalent, lower level representation.

The first thing the transformation algorithm needs to do is to classify each variable that appear within the atomic goal as one of the following:

A local variable A variable that does not appear outside the atomic scope.

An input variable A variable which appears outside the atomic scope but is bound to a term before the atomic goal.

An output variable A variable which appears outside the atomic scope but is bound within the atomic goal itself.

We can classify each variable within the atomic goal by using two pieces of information that is associated with every goal in the HLDS: the set of non-local variables and the changes in variables' instantiation state across the goal (called the "instmap-delta"). A variable that appears within an atomic goal is a member of the set of non-locals if it also appears outside the atomic scope. We can use this to calculate the variables that are

local to the atomic goal by a simple set subtraction. To work out the input and output variables, we need to work out the instantiation of the non local variables before and after the atomic goal (which is done by using the instmap-deltas).

This information is passed on to the algorithm which performs the necessary transformations. There are two factors which affect how the atomic goal is transformed: one of them is whether the atomic goal is a top level atomic goal or a nested atomic goal and the other is whether or not the atomic goal contains any or_else alternatives. Since the other transformations are built upon the transformations we have developed for a top level atomic goal with no or_else alternatives, we will discuss this scenario first. A complete example of the resulting code from a top-level atomic goal transformation with no or_else alternatives is provided in the appendix.

5.3.1 Top Level Atomic Goal

The code generated for a top level atomic goal must include the atomic code itself along with the code to create the transaction log, to validate and commit the transaction log, to catch exceptions and to re-execute the atomic goal if the transaction log is invalid. The standard way to execute code in Mercury more than once is to recursively call the predicate the code appears in. Therefore, we can create a separate predicate for each atomic goal which contains the atomic code along with the support code to create and validate a transaction log. With this approach, it is possible for the predicate to recursively call itself whenever a log is invalid. However, this approach is inadequate as it does not consider the handling of exceptions.

In Mercury, exceptions that can be raised by a predicate are caught by passing that predicate as a closure to the built-in `try` predicate. The `try` predicate will execute the closure and will return a result indicating whether the closure has exited successfully or if it has raised an exception. We are using Mercury's exception system to raise requests to rollback the log as it provides an easy method of sending execution back to the top of the atomic goal. But this means we must separate the atomic code from the code that catches the exception by placing the atomic code in a separate predicate. Therefore, we need to create not one by two auxiliary predicates to implement the correct behaviour of the atomic goal.

The approach we have taken is to create three predicates for one top level atomic goal. The created predicates are:

The Wrapper Predicate This predicate contains the original atomic goal and is also responsible for validating and committing the transaction log.

The Rollback Predicate This predicate is responsible for calling the wrapper predicate and handling the validation result. It is also responsible for handling requests for retries and for handling any exceptions the atomic goal might raise.

The Top-Level Predicate This predicate simply invokes the rollback predicate. It provides the abstraction that the entire transaction is a single I/O action.

The wrapper predicate is the centerpiece of the transformation as it contains the actual atomic subgoal. It is invoked by the rollback predicate through a specialised

version of the try predicate, which is used to catch any exceptions the wrapper predicate might throw. The arguments of the wrapper predicate include the input variables and the initial transaction log, which is created by the rollback predicate shortly before the wrapper predicate is invoked. These two arguments correspond to the inner parameter of the original atomic scope. The wrapper predicate also is responsible for returning the final transaction log as this is needed by the rollback predicate when an exception is raised.

The purpose of the wrapper predicate is to execute the main atomic subgoal and validate the final transaction log. If the resulting transaction log is valid, the wrapper predicate commits it and exits successfully. If the resulting transaction log is invalid, the wrapper predicate discards it as a raises a rollback exception. This exception is caught by the try predicate and is handled by the rollback predicate. The validation and the act of committing a transaction must be protected by concurrently running threads. The wrapper predicate does so by acquiring a global lock before it validates and commits the transaction. The wrapper predicate is also responsible for releasing the lock after committing or before indicating a rollback.

Reliance on the exception system has a number of disadvantages: the main one being that the closure passed to the specialised try predicate can only have a single output argument. To get around this restriction, we have used a packing and unpacking technique, using Mercury tuples. For cases where the main atomic subgoal does not have any output variables, a reserved dummy value is returned.

Pseudo code of the wrapper predicate is provided in figure 5.

```
wrapper_pred(InputVars, SingleOutputVar, Log0, Log) :-
    execute the atomic goal,        % transaction log goes from Log0 to Log
    build the output variable producing SingleOutputVar,
    acquire the global lock,
    validate transaction log Log,
    ( log is valid ->
        commit transaction log Log,
        release the global lock
    ;
        release the global lock
        throw(rollback exception)
    ).
```

Figure 5: Pseudo Mercury code of the wrapper predicate

The rollback predicate can be seen as the controller of the wrapper predicate. It is responsible for creating the transaction log and calling the wrapper predicate. It is also responsible for unpaking the output variables if the wrapper predicate succeeds and for handling any exceptions the wrapper predicate might throw. The wrapper predicate has handling code for the following exceptions:

Rollback Exception Thrown by the wrapper predicate when a rollback is required. When received, the rollback predicate discards the current transaction log and recursively calls itself.

Retry Exception Thrown by the retry predicate. When received, the rollback predicate first validates the transaction log. If the log is valid, the rollback predicate blocks the thread on all the transaction variables it has seen so far. After the thread has been woken up, or if the log was invalid, the wrapper predicate discards the transaction log and recursively calls itself.

Other Exceptions Any other exceptions are also caught by the rollback predicate. When caught, the rollback predicate first validates the transaction log. If the log is valid, the exception is passed to the next active handler. If the log is invalid, the wrapper predicate discards the transaction log and recursively calls itself.

The rollback predicates accepts the set of input variables as in arguments and is responsible for passing back the set of output variables as out arguments. The rollback predicate passes the input arguments to the wrapper predicate and the wrapper predicate returns the packaged output arguments. The output variables are unpacked by the rollback predicate and passed to the top level predicate. Since the input and output variables differ for every atomic scope, one rollback predicate is created for each atomic scope (in a similar way to the wrapper predicate).

A high level description of the rollback predicate is provided in Figure 6.

```
rollback_pred(InputVars, OutputVars) :-
    create transaction log,
    try(wrapper predicate),
    ( wrapper predicate succeeded ->
        unpack output and return
    ; rollback exception called ->
        discard log and recursively call self
    ; retry exception called ->
        validate log,
        ( log is valid ->
            block thread on transaction variables seen so far
            discard log and recursively call self
        ;
            discard log and recursively call self
        ),
        recursively call self
    ; % any other exception
        validate log,
        ( log is valid ->
            rethrow exception
        ;
            discard log and recursively call self
        )
    )
```

Figure 6: Pseudo Mercury code of the rollback predicate

The original top level atomic goal is replaced with a call to the top level predicate. This call takes the set of input variables and returns values for the set of output variables.

5.3.2 Nested Atomic Goals

There are two approaches to handling nested atomic goals. The first approach is to create a nested transaction log for the nested atomic goal. The nested transaction log is passed to the atomic subgoal of the nested atomic goal. Once the nested atomic subgoal is complete, the final version of the nested transaction log is *merged* with the parent log. Merging involves taking all log entries from the nested transaction log and merging it with the parent which, in this case, is the transaction log specified in the `outer` parameters of the atomic scope.

The second approach is to simply give the outer transaction log to the nested atomic subgoal. In this approach, all read and write operations are recorded to the outer transaction log directly. Once the nested atomic subgoal has finished, the new version of the transaction log is passed back to the outer scope.

We have selected the second approach for nested atomic goals with no or_elses. The reasons for this are:

1. The resulting parent log after being merged with the child log would be the same as if all changes in the nested atomic subgoal where logged to the parent log. Therefore, the creation of the nested transaction log is an unnecessary expense.

2. We cannot commit the changes of a nested transaction as the transaction has not yet completed (the transaction is only fully completed when execution reaches the end of the top level atomic goal).

3. All exceptions would need to be passed up to the top level atomic goal as the entire transaction would need to be re-executed if the log is invalid.

When a nested atomic goal with no or_else alternatives is encountered, the transform algorithm simply replaces the atomic goal with the atomic subgoal. The atomic subgoal is also bound by two assignment statements. These assignment statements connect the outer and inner transaction logs before and after the nested atomic subgoal. This avoids the need to rename the variables within the nested atomic subgoal. Since transaction logs are passed around as references, using this approach lets the programmer use the nested atomic goal at virtually no cost.

5.3.3 The Or_Else Alternatives

The final case the transformation algorithm handles is atomic goals which contain or_else alternatives. The way we have approached the implementation of the or_else operator is fairly involved due to is complex functional requirements. Despite the differences between top level atomic goals and nested atomic goals, the approach we have taken for handling the or_else operator is the same for both of these types of goals.

The code generated for the or_else operator must go through each alternative subgoal until it has found one that does not throw a retry. Each alternative is given a nested transaction log so that any change the or_else alternative makes has no effect on the parent

```
forall (or_else alternative subgoal) do
    create nested transaction log
    execute the alternative subgoal with nested transaction log
    if (subgoal does not retry)
        merge nested transaction log and parent transaction log
        exit successfully
    end if
end for

% All or_else alternatives have retried
forall (created nested transaction log)
    validate log
    if (log is invalid)
        raise invalid rollback exception
    end if
end for

% All nested transaction logs are valid
forall (created nested transaction log)
    merge nested transaction log with parent
end for

retry transaction
```

Figure 7: Imperative description of the execution algorithm of the or_else operator

```
or_else_pred(InputVars, OutputVars, ParentLog0, ParentLog) :-
    create nested transaction log with parent log ParentLog0
    try(simple wrapper predicate 1),
    ( wrapper predicate succeeded ->
        unpackage output variables into OutputVars
        merge nested transaction log producing ParentLog
        discard all nested transaction logs
    ; non-retry exception called ->
        discard all nested transaction logs
        rethrow caught exception
    ; retry exception called ->
        create new nested transaction log with parent log ParentLog0
        try(simple wrapper predicate 2),
        ( wrapper predicate succeeded ->
            unpackage output into OutputVars
            merge nested transaction log producing ParentLog
            discard all nested transaction logs
        ; non-retry exception called ->
            discard all nested transaction logs
            rethrow caught exception
        ; retry exception called ->
            % If there are more or_else alternatives, the code to handle
            % them would go here.

            ( both nested transaction logs are valid ->
                % The merge operation of all nested transaction logs is
                % performed here so that the transaction waits on the union
                % of all transaction variables seen by each of the
                % alternatives. The local values in ``ParentLog'' here are
                % not used.

                merge all nested transaction logs with ParentLog0
                        producing ParentLog
                discard all nested transaction logs
                retry on ParentLog
            ;
                discards nested transaction log
                throw invalid transaction exception
            )
        )
    )
```

Figure 8: Pseudo Mercury code of the or_else predicate with two alternatives

transaction log until that alternative has successfully completed. If every alternative has retried, and the nested transaction logs created for each alternative are still valid, the parent transaction log is then retried. A description of the or_else execution algorithm is provided in figure 7.

Since the or_else alternatives are quite complex and are dependent on the number of alternatives, our transformations creates a new predicate (called the *or_else* predicate) for each atomic goal which contains or_else alternatives. Since Mercury does not contain any iterative constructs, we had to find a way to run each alternative in turn using other methods. Recursion is a possibility but would be too difficult to implement. Therefore, we have settled upon a brute force approach and have unrolled the Pseudo code for the generated or_else predicate with two alternatives is available in figure 8.

The or_else predicate takes, as arguments, the common set of input variables (which is the union of each set of input variables from each alternative subgoal) and produces the common set of output variables (which Mercury requires to be the same for each alternative subgoal). Since each alternative subgoal needs to create a nested transaction log, the or_else predicate also take the current parent transaction log and produces the new parent transaction log. This permits the use of the or_else operator in nested atomic goals as well as for top level atomic goals.

Along with the or_else predicate, we need to create a separate predicate for each or_else alternative. Since retries are indicated by throwing an exception, we must catch any exceptions thrown by each alternative subgoal. As stated before, the only way to do this in Mercury is using the `try` predicate which takes a closure. The predicate created for each subgoal, called a *simple wrapper* predicate, simply contains the or_else alternative subgoal along with any output packing that is required. The clause of these predicates are similar to the *wrapper* predicate of top level atomic scopes but do not contain the code which validates and commits the transaction log.

The merging of each nested transaction log (that has been produced by each or_else alternative) with the parent transaction log produces a merged log containing the union of all transaction variables seen in each nested transaction log. This property is used by the or_else predicate to determine the set of transaction variables the transaction will wait on when all alternatives retry. The local values of the transaction variables in the log do not matter as the transaction will be retried anyway.

The final thing the transformation algorithm must do is include the proper call to the or_else predicate. For nested atomic goals, the atomic goal is replaced with a call to the or_else predicate. This call takes the input variables as in arguments and produces the output variables as out arguments. It also takes the outer transaction log and produces a new version of the outer transaction log after the return.

The call to the or_else predicate within a top level atomic goal is made within the wrapper predicate. This is because the final version of the transaction log must be validated and committed. Therefore, the wrapper predicate only contains the call to the or_else predicate, along with the code to validate and commit the transaction log. Since the or_else predicate returns the output variables as out arguments, these must be packed before they can be sent up to the rollback predicate.

5.4 Library and Runtime

We have also added a new library module into the standard Mercury library. This library module contains the predicates which can be used by the user and the primitive operations used by the transformations.

5.4.1 Transaction Variables

Transaction Variables are heap allocated data structures which contain two things:

- The current value of the transaction variable.
- A list of *wait entries*.

Since it is possible to represent any Mercury value as a single word of memory, the value of the transaction variable is stored within the transaction variable as a word. The transaction variable also keeps a list of "wait entries". Each "wait entry" corresponds to a thread which has been blocked until the value of one of the transaction variables the thread has seen is changed. Wait entries are added and removed by the predicate which block the transaction.

5.4.2 Transaction Logs

During a transaction, a thread local transaction log is built-up with the recording of the reads and tentative writes that the transaction has performed. The transaction log is a heap allocated object and is created and returned to user code when the thread enters the atomic sub-goal.

The transaction log contains one entry for each transaction variable the transaction has accessed. Each entry contain the following attributes:

- The transaction variable this entry refers to.
- The value of the transaction variable when the entry was created (the "old value").
- The locally updated value of the transaction variable used by the transaction log (the "new value"). If the transaction log is committed, the new value of the transaction variable will be set to the value of this attribute.

Entries are added by calls to `read_stm_var` and `write_stm_var`. When a request to read or write to a transaction variable is made, the transaction log is first scanned in search for an existing log entry for that transaction variable. If an entry is found, the predicate will use the associated "new value" to represent the value of the transaction variable (i.e. a read will return the "new value" and a write will modify the "new value"). If no entry exists for that transaction variable, a new entry is created with an "old value" and a "new value" equal to the current value of the transaction variable. Because the transaction log is local to a transaction, there is no possibility that reads or updates performed by the transaction can be viewed by concurrently running transactions which ensure "isolation" between these transactions. Furthermore, since reads and writes consult the log first, the actual values of transaction variables are changed only when the transaction commits. Hence, any changes by the log can be implicitly undone by simply removing the transaction log.

5.4.3 Nested Transaction Logs

Nested transaction logs are transaction logs which are based upon a parent transaction log. They are included so that any updates performed by an or_else alternative subgoal does not affect the main transaction log until the nested transaction has successfully completed (i.e. does not throw a retry).

When a request to read a transaction variable occurs within an or_else alternative subgoal, the read first consults the nested transaction log. If no entry for the transaction variable exists within the nested transaction log, the read then consults the parent transaction log. When a request to write a transaction variable occurs, the entry within the nested transaction log is updated. No changes at all, including the addition of entries for transaction variables seen for the first time, are made to the parent log throughout the or_else alternative subgoal.

If the alternative subgoal completes successfully, the nested transaction log is *merged* with the parent transaction log. This involves going through each entry in the nested transaction log and checking whether or not the parent log also contains an entry for the associated transaction variable. If so, the "new value" of the entry in the parent's transaction log is changed to the "new value" of the entry in the nested transaction log. If not, the entry in the nested transaction log is added to the parent transaction log. After the merge, the parent of the transaction log contains all the changes made by the nested transaction log.

5.4.4 Validation and Commit

Once execution has reached the end of the atomic sub-goal, the final version of the transaction log is validated. Validation involves going through each entry in the transaction log and comparing the "old value" with the value of the associated transaction variable. Since the "old value" and the value of the transaction variable are both words. we simply determine equality by comparing the bit sequence of the two values. This test suffices as it is extremely rare that the bit sequences of the two words values differ if the terms they refer to are the same (it is, of course, impossible for the bit sequence to be the same if the two terms are different). If the value of all transaction variables are equal, the log is deemed valid.

The act of committing a transaction log comes after validation. Committing a transaction log involves going through all entries of the transaction log and changing the value of the associated transaction variable to the "new value" of the transaction log.

Validation and the act of committing a transaction log must be atomic. When validating a transaction log, we cannot allow other threads to change the values of the transaction variables as doing so might cause an incorrect validation result. When committing a transaction log, we cannot allow any other threads to change the value of any transaction variables as doing so would leave the shared memory in an inconsistent state. In our implementation, we protect the validation and the act of committing a transaction log using a single global lock. This lock must be acquired by any thread that wants to validate or commit a transaction and must be released as soon as possible afterwards so that other transactions have a chance to validate and possibly commit their transaction logs.

5.4.5 Thread Blocking

When a retry occurs, the transaction is blocked until one of the transaction variables seen in the log has been changed by the act of committing another transaction. Unfortunately, we cannot associate condition variables to transaction variables as we cannot block the thread on more than one condition variable. Therefore, we associate the condition variable to threads. Condition variables are provided by the standard POSIX thread library and can be used to suspend the thread until the call to "signal" it is made.

Before suspending the thread, a *wait entry* is added to the list of wait entries for every transaction variable seen by the log. The wait entry contains the reference to the condition variable that this thread has been blocked on. When a transaction is changing the value of a transaction variable (while it is being committed), it traverses the list of wait entries. Every thread that has a wait entry in this list is signalled. When a thread is signalled, the thread removes its associated wait entry from the transaction variables it has added a wait entry to. It is now free to discard the current log and restart.

6 Discussion

6.1 Limitations

Currently, from the tests we have executed, we have reasonable confidence that the software transactional memory constructs work correctly in the following situations:

- Top level atomic scopes with a number of input variables and zero, one or more output variables.
- Nested atomic scopes with a number input variables and zero, one or more output variables.
- Top level atomic scopes which retries or throws an exception.
- Top level atomic scopes which calls a predicate which retries or throws an exception.
- Atomic scopes with or_else alternatives.
- Nested atomic scopes with or_else alternatives.

The following situations are not fully operational:

Atomic scopes which only contain a retry Atomic scopes which only contain a retry cause the compiler to raise an exception when compiling. The reason for this is that the current implementation relies on the introduction of dummy predicates to enforce uniqueness of the outer and inner variables. Since the retry exception has the mode "erroneous", the compiler removes all goals that appear after the retry predicate, including the dummy predicates. Ultimately, we need to fully implement mode checking for atomic goals so that these dummy predicates are not needed.

Nested atomic scopes with or_else in polymorphic predicates Using a nested atomic scope with or_else alternatives within a predicate which uses Mercury's parametric polymorphism system currently causes the compiler to raise an exception. This is because some aspects of the Run Time Type Information associated with the created predicate are not currently handled by the transformations of the or_else predicate.

Along with the above limitations, the following aspects of this implementation are not complete:

Support for the low level C backend Some aspects of the software transactional memory are backend specific — particularly the thread control. Blocking the thread in the low level C backend requires explicit dealing with contexts. Currently, the STM runtime does not have this code. Since the low level C backend is one of the two most important backends, this will eventually need to be completed.

6.2 Future Work

Along with the limitations, a number of enhancements can be made to improve the usability of the STM constructs. The following is a brief discussion on a number of improvements that can be made to the implementation of the STM system.

Parallel Commits

The current implementation uses a single global lock to protect the validation and the committing of transaction logs from other threads which may be trying to commit at the same time. For transactions which access or modify the same set of transaction variables, this is necessary. But for two transactions which do not share a common set of transaction variables, this is overkill. A way to prevent the serialisation of commits is to lock individual transaction variables instead of having a global lock. One approach of doing this has been developed by Tim Harris, et. al. for a STM implementation in Haskell[5]. Their approach involves modifying the act of validating and committing a transaction log into a three step process:

Acquiring the transaction variables This involves going through the transaction log and attempting to acquire the lock of every transaction variable that corresponds to an update entry. The implementation uses the CAS (compare-and-swap) instruction to check if a transaction variable is currently locked and to check if the transaction variable holds the value expected by the transaction. If a transaction variable is locked, all locks held by the transaction are released and the transaction retries. Transaction variables corresponding to a read entry will have their version number recorded.

Checking the reads This involves going through the transaction log again checking the new version of transaction variables that correspond to a read entry. If the transaction variable's version number differs from the version number recorded in the previous step, all locks are released and the transaction retries.

Make updates This involves going through the transaction log once more and modifying the value of transaction variables that correspond to an update entry. Once a transaction variable has been modified, the lock on the transaction variable is released.

This three step process ensures atomicity whilst allowing the parallel committing of transaction logs that do not share transaction variables.

Invariants

Another possible extension is support for *invariants*[4]. This method was implemented by Tim Harris for the Concurrent Haskell implementation of software transactional memory. Invariants promises to be an approach which further improves the consistency of software transactional memory by providing a method of explicitly stating what constitutes a consistent view of memory.

Essentially, an invariant is a software transactional memory goal which, when defined, must be satisfied by every user transaction until the program terminates. Any transaction that does not satisfy an invariant is rolled back. The invariant goal is itself a memory transaction. Before a transaction log is committed, each invariant goal is executed with a nested transaction log (based upon the transaction log being committed). The invariant cannot make any changes to the transaction log being committed. Once execution of the invariant has finished, the transaction log of the invariant is discarded.

Implementation Improvements

Along with the addition of functional features, other improvements which would be beneficial would be improvements to the implementation itself. The following changes to the implementation could potentially increase the performance and the ease of use of the software transactional memory constructs:

Remove the reliance on the exception system The current implementation makes heavy use of closures, which is reasonably inefficient. As we would like the software transactional memory system to execute as fast as possible, we would like to remove the reliance on the exception system to indicate rollbacks and retries. This may involve generating code later on in the compiler but would potentially result in significantly faster code.

Use of state variables within the outer and inner parameters Currently, the outer and inner parameter only accepts a pair of variables but requiring the programmer to use this notation in large predicates would be an extreme nuisance. Mercury supports the use of state variables which makes it much easier to write code which needs to pass state around (e.g. code which performs I/O). We should support this notation for the outer and inner parameters as well so that code which makes use of them can be modified easily.

General optimisation over atomic goals The current implementation does not perform any optimisations over the atomic goal. We will need to provide some so that the code generated would be reasonably efficient. For example: an atomic goal

which does not call any predicates which take STM state does not need the transaction log. Such code can be taken out of the atomic scope so that the generated code does not incur the cost of creating and validating a transaction log.

7 Conclusion

Software developers will eventually need to realise that concurrent programming is important for the development of tomorrows high performance programs. But using locks to synchronise data, which is very important in concurrent programs, is notoriously difficult. However, software transactional memory is promising to be a much easier method of synchronisation and supports the development of large, concurrent programs.

In this thesis, we have introduced our implementation of a software transactional memory system in Mercury. We have added the notion of transaction variables that can be used to synchronised data within memory transactions. We have also introduced two new language constructs, the atomic scope and the or_else operator, which can be used by the programmer to create transactions and alternative transactions. We have also added the "retry" predicate which blocks a thread until the value of any transaction variable seen by the thread has been changed. Whilst a few areas of the implementation need to be completed, our implementation shows that the fundamentals are working.

8 Acknowledgements

Thanks to the guys in the Mercury lab who have helped with this thesis. Special thanks to Julien Fischer for providing direction, helping with the runtime system and for answering my almost continuous stream of emails. Also, special thanks to my supervisor, Zoltan Somogyi, for helping in any way possible. All your efforts are much appreciated.

A Example Source To Source Transformation

This section shows the source to source transformations generated for the transfer predicate in section 4.2.

```
:- pred transfer(int::in, stm_var(account)::in, stm_var(account)::in,
    io::di, io::uo) is cc_multi.

transfer(Amount, AccountA, AccountB, IO0, IO) :-
    'StmExpanded_toplevel_transfer_5_1_2'(Amount, AccountA, AccountB,
    IO0, IO).


:- pred 'StmExpanded_toplevel_transfer_5_1_2'(stm_var(account)::in,
    stm_var(account)::in, int::in, io::di, io::uo) is cc_multi.

'StmExpanded_toplevel_transfer_5_1_2'(AccountA, AccountB, Amount,
        IO0, IO) :-
```

```
    'StmExpanded_rollback_transfer_3_1_0'(AccountA, AccountB, Amount),
    IO = IO0.


:- pred 'StmExpanded_rollback_transfer_3_1_0'(int::in,
    stm_var(account)::in, stm_var(account)::in) is cc_multi.

'StmExpanded_rollback_transfer_3_1_0'(Amount, AccountA, AccountB) :-
    promise_pure (
        impure stm_create_transaction_log(STM0_Aux_1),
        Closure_Aux_3 = 'StmExpanded_wrapper_transfer_6_1_1'(Amount,
            AccountA, AccountB),
        unsafe_try_stm(Closure_Aux_3, ExceptionResult_Aux_4,
            STM0_Aux_1, STM_Aux_2),
        (
            ExceptionResult_Aux_4 = exception(ExceptUnivVar_Aux_6),
            (
                RollbackExcpt_Aux_12 = rollback_invalid_transaction,
                type_to_univ(UnivPayload_Aux_11, ExceptUnivVar_Aux_6),
                UnivPayload_Aux_11 = RollbackExcpt_Aux_12
            ->
                impure stm_discard_transaction_log(STM_Aux_2),
                'StmExpanded_rollback_transfer_3_1_0'(Amount, AccountA,
                    AccountB)
            ;
                (
                    RollbackExcpt_Aux_10 = rollback_retry,
                    type_to_univ(UnivPayload_Aux_9, ExceptUnivVar_Aux_6),
                    UnivPayload_Aux_9 = RollbackExcpt_Aux_10
                ->
                    impure stm_lock,
                    impure stm_validate(STM_Aux_2, ValidResult_Aux_8),
                    (
                        ValidResult_Aux_8 = stm_transaction_valid,
                        impure stm_block(STM_Aux_2)
                    ;
                        ValidResult_Aux_8 = stm_transaction_invalid,
                        impure stm_unlock
                    ),
                    impure stm_discard_transaction_log(STM_Aux_2),
                    'StmExpanded_rollback_transfer_3_1_0'(Amount,
                        AccountA, AccountB)
                ;
                    impure stm_lock,
                    impure stm_validate(STM_Aux_2, ValidResult_Aux_7),
                    impure stm_unlock,
```

```
                    (
                        ValidResult_Aux_7 = stm_transaction_valid,
                        rethrow(ExceptionResult_Aux_4)
                    ;
                        ValidResult_Aux_8 = stm_transaction_invalid,
                        'StmExpanded_rollback_transfer_3_1_0'(Amount,
                            AccountA, AccountB)
                    )
                )
            )
        ;
            ExceptionResult_Aux_4 = succeeded(DummyResult_Aux_5)
        )
    ).


:- pred 'StmExpanded_wrapper_transfer_6_1_1'(int::in,
    stm_var(account)::in, stm_var(account)::in, stm_dummy_output::out,
    stm::di, stm::uo) is cc_multi.

'StmExpanded_wrapper_transfer_6_1_1'(Amount, AccountA, AccountB,
        Stm_ResultVar, STM0, STM) :-
    withdraw(Amount, AccountA, STM0, V_17),
    deposit(Amount, AccountB, V_17, STM),
    Stm_ResultVar = stm_dummy_output,
    promise_pure (
        impure stm_lock,
        impure stm_validate(STM, Stm_Expand_IsValid_Aux_0),
        (
            Stm_Expand_IsValid_Aux_0 = stm_transaction_valid,
            impure stm_commit(STM),
            impure stm_unlock
        ;
            Stm_Expand_IsValid_Aux_0 = stm_transaction_invalid,
            impure stm_unlock,
            Stm_Expand_Rollback_Aux_1 = rollback_invalid_transaction,
            throw(Stm_Expand_Rollback_Aux_1)
        )
    ).
```

References

[1] Livermore Computing. POSIX thread programming. Available at www.llnl.gov/computing/tutorials/pthreads/, July 2007.

[2] Thomas Conway. *Towards Parallel Mercury.* Ph.D. Thesis, Department of Computer Science and Software Engineering, University of Melbourne, 2002.

[3] Tim Harris and Keir Fraser. Language support for lightweight transactions. *Proceedings of the ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2003.

[4] Tim Harris and Simon Peyton Jones. Transactional memory with data invariants. *First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT'06)*, June 2006.

[5] Tim Harris, Simon Marlow, and Simon Peyton Jones. Haskell on a shared-memory multiprocessor. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell*, pages 49–61, New York, NY, 2005.

[6] 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, 2005.

[7] Fergus Henderson, Thomas Conway, and Zoltan Somogyi. Mercury, an efficient purely declarative logic programming language. pages 499–512, Glenelg, Australia, 1995.

[8] Fergus Henderson, Thomas Conway, Zoltan Somogyi, and David Jeffery. The Mercury language reference manual. Technical Report 96/10, Department of Computer Science, University of Melbourne, Melbourne, Australia, 2006.

[9] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the Twentieth Annual International Symposium on Computer Architecture*, 1993.

[10] Simon Peyton Jones. *Beautiful Code*, chapter 24. Beautiful Concurrency. O'Reilly, June 2007.

[11] Richard Kelsey, Jonathan Rees, and Mike Sperber. The incomplete Scheme 48 reference manual release 1.3. April 2005.

[12] Kenneth C. Louden. *Programming Languages: Principles and Practices*, chapter 6.9. Explicit Polymorphism, page 244. Thomson Brooks/Cole, CA, USA, 2nd edition, 2003.

[13] Ulrich Muller. Introducing the atomic keyword into C/C++ using assembler code instrumentation and software transactional memory. 2006.

[14] Yang Ni, Vijay S. Menon, Ali-Reza Adl-Tabatabai, Antony L. Hosking, Richard L. Hudson, J. Eliot B. Moss, Bratin Saha, and Tatiana Shpeisman. Open nesting in software transactional memory. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on i Principles and practice of parallel programming*, pages 68–78, New York, NY, USA, 2007. ACM.

[15] Kunle Olukotun and Lance Hammond. The future of microprocessors. *Queue*, 3(7):26–29, 2005.

[16] Shaz Qadeer and Dinghao Wu. Kiss: Keep it simple and sequential. *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, 2004.

[17] Herb Sutter. The free lunch is over: a fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 2005.

[18] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.

[19] Andrew Tanenbaum. *Modern Operating Systems*, chapter 2.3. Interprocess Communication. Prentice Hill, New Jersey, 2001.

[20] The GHC Team. *The Glorious Glasgow Haskell Compilation System User Guide*, chapter 7.15. Concurrent and Parallel Haskell. April 2007.

[21] The Mercury Developement Team. Notes on the design of the mercury compiler. Available at http://www.cs.mu.oz.au/research/mercury/information/doc-latest/compiler_design.html, June 2007.

[22] Peter Wang. *Parallel Mercury.* Honours Thesis, Department of Computer Science and Software Engineering, 2006.