MUTS: Native Scala Constructs for Software Transactional Memory

MUTS: Native Scala Constructs for Software Transactional Memory

Daniel Goodman Behram Khan Salman Khan
Chris Kirkham Mikel Luján Ian Watson
The University of Manchester
Daniel.Goodman@cs.man.ac.uk

Abstract

In this paper we argue that the current approaches to implementing transactional memory in Scala, while very clean, adversely affect the programmability, readability and maintainability of transactional code. These problems occur out of a desire to avoid making modifications to the Scala compiler. As an alternative we introduce Manchester University Transactions for Scala (MUTS), which instead adds keywords to the Scala compiler to allow for the implementation of transactions through traditional block syntax such as that used in "while" statements. This allows for transactions that do not require a change of syntax style and do not restrict their granularity to whole classes or methods. While implementing MUTS does require some changes to the compiler's parser, no further changes are required to the compiler. This is achieved by the parser describing the transactions in terms of existing constructs of the abstract syntax tree, and the use of Java Agents to rewrite to resulting class files once the compiler has completed. In addition to being an effective way of implementing transactional memory, this technique has the potential to be used as a light-weight way of adding support for additional Scala functionality to the Scala compiler.

1. Introduction

In this paper we will describe how, through a combination of modifications to just the compiler parser and bytecode rewriting, we have added native support for Software Transactional Memory (STM) [7] to the Scala programming language [10]. As motivation for this we argue that existing implementations of STM either have limitations on the granularity of the transactions, or require a syntax change between transactional and non-transactional code. This, we feel, results in fragmentation of the code, adversely affecting its programmability, readability and maintainability. These limitations occur because these implementations restrict themselves to not making any changes to the compiler. We instead created Manchester University Transactions for Scala (MUTS), in which we restrict ourselves to making the minimal set of changes required to implement transactional memory such that MUTS does not place limitations on the granularity of transactions or require changes to the syntax. The changes to the compiler turn out to be a relatively small set of changes to the parser, in order to detect transactions and encode the required logic using existing components of the abstract syntax tree. Having added these elements to the tree, the process of adding transactions is completed by a byte-code rewrite, so preventing the need to make any further modifications to the compiler, and leaving all but the first compilation step unchanged. In addition to being an effective way of implementing transactional memory, this technique has the potential to be used as a light-weight way of adding support for additional Scala functionality to the Scala compiler.

We will now introduce in more detail both Transactional Memory and Scala. Then we will introduce some of the existing STMs for Scala and Java before detailing our implementation.

1.1 Transactional Memory

To ensure that correct results are generated when handling the concurrent access to shared memory it is necessary to control access to the shared state. Without such controls errors can occur when threads interleave reads and writes. For example, consider a program that contains a counter to keep track of the number of completed threads. In this example, each thread at the end of its execution performs the assignment `counter = counter + 1` to increment the counter. Unfortunately, if thread A reads the value of `counter`, then Thread B reads the value of `counter` and increments it, when thread A writes back its result, because it read `counter` before B incremented it, the value of `counter` will be one less than it should be and the computation may no longer behave correctly.

Traditionally this issue is handled by some form of locking that restricts the access to the shared state to a single thread at a time. This approach has several complications:

1. The composition of functions that require a lock needs to break the encapsulation of the implementation of the composed functions. For example if we have functions to

deposit and withdraw money from bank accounts, each of these functions will require a lock to ensure that the balances remain correct. To now construct a function that atomically transfers money from one account to another so that all money can remain accounted for at all time, it is necessary to get the locks of both the sending and the receiving account before the functions can be invoked. To achieve this, the locking mechanisms for the individual functions and therefore information about their internal data structures has to be made available to the programmer. This breach of encapsulation affects both programmability and maintainability.

2. Code that requires multiple locks to be obtained is prone to dead-lock or live-lock as competing functions may attempt to gain the same set of locks in a different order. If the set of locks required is known in advance, then a total order can be applied to the locks to overcome this, but for a large collection of interesting problems this is not possible. For example, exploring a graph in order to make modifications.
3. Locking is pessimistic, it assumes that there will be a conflict and so restricts access on the basis of this. This pessimistic nature of the locking means that opportunities for concurrency are missed. As the world is forced to use parallel processing due to the limits of single core processors, this is becoming a serious issue.

One solution to these problems is transactional memory. Here, instead of taking locks, the code executes and records sufficient information such that conflicts can be detected and one of the conflicting transactions can be rolled back and restarted when a conflict occurs. The underlying system is constructed such that when the transaction has completed, it will attempt to commit, and if it succeeds its changes to the system will appear atomically, otherwise the transaction will roll back such that, as far as all other processes are aware, it never executed. This means that the semantics of transactions are equivalent to having a single global lock that the transaction takes, while it executes, but because of the underlying implementation the possible concurrency is far higher.

We will now describe how this alternative approach addresses the specific points raised about locks.

1. Because transactions can be nested within one another, and the collection of data and the handling of collision detection is all dealt with by the runtime, the user does not need to be aware of any of the internal locking information of functions, so the composition of functions does not require the breaking of encapsulation.
2. As there are no specific locks, just logged information, there is no possibility of dead-locking. Live-lock is possible with some collision detection mechanisms, but the use of live lock free collision management resolves this.
3. Transactions do not have locks that prevent code from executing concurrently. As a result they are optimistic and concurrency is not restricted.

1.2 Scala

Scala [10] is a general purpose programming language designed to smoothly integrate features of object-oriented [12] and functional languages [2]. By design it supports seamless integration with Java and existing Java code including libraries and the compiler generates Java byte-code [9]. This means that you can call Scala from Java and you can call Java from Scala. Scala is a pure object-oriented language in the sense that every value is an object. Types and behaviour of objects are described by classes and traits, and classes are extended by sub-classing and a flexible mixin-based composition mechanism as a clean replacement for multiple inheritance. However, Scala is also a functional language in the sense that every function is a value. This power is furthered through the provision of a lightweight syntax for defining anonymous functions, support for higher-order functions, the nesting of functions, and support for currying. Scalas case classes and its built-in support for pattern matching model algebraic types are equivalent to those used in many functional programming languages.

For typing, Scala is statically typed and equipped with a type system that statically enforces that abstractions are used in a safe and coherent manner. A local type inference mechanism means that the user is not required to annotate the program with redundant type information.

Finally Scala provides a combination of language mechanisms that make it easy to smoothly add new language constructs in the form of libraries. Specifically, any method may be used as an infix or postfix operator, and closures are constructed automatically depending on the expected type (target typing). A joint use of both features facilitates the definition of new statements without extending the syntax and without using macro-like meta-programming facilities.

2. Related Work

There are already a large number of transactional memories for different languages implemented in software and many proposals for both hardware and hybrid systems [7]. Here we will briefly look at some of the existing Scala STM's and explain why we feel that they do not meet our needs. We will also introduce Deuce STM which is an STM targeted at Java and modified as part of the work described in this paper.

2.1 Scala STMs

Existing Scala STM's that we are aware of fall into two categories, both of which aim to add transactions to Scala without modification to the Scala compiler:

1. **Libraries that require the user to add method calls for every read and write to variables within a transaction. [11]** This technique adds a huge amount of addi-

tional code even in Scala code which is for the most part functional. This in turn makes writing, reading and maintaining programs challenging. In addition, in languages such as Scala, ensuring that all relevant variables are instrumented is challenging as the introduction of implicit variables is easy. For these reasons this style of STM is rare.

2. **Libraries which take advantage of Scala's treatment of functions as first class variables. [3, 11]** This allows for the creation of a function `atomic` that takes a function as its argument. Because Scala will implicitly convert a set of statements into a function that can be passed into this library call, the definition of a transaction can be achieved as though a new keyword has been added to the language. The library calls for the individual reads and writes can then be added by the creation of an encapsulating object for each piece of transactional data and utilising Scalas ability to override accessors. This creates code that looks very much like a new set of keywords in the language. However, there are some changes between the code that appears inside the atomic library call and outside the call, for example in CCSTM = has to be replaced with := and the variables have to be appropriately wrapped. These changes are required because in Scala it is not possible to overload = to handle the instrumentation of transactional code, or for the method := to be implemented in a single location without the enclosing wrapper class. These changes create opportunities for errors to slip into the system, and make changing the regions marked as atomic labour intensive. As such we feel that while this is a very neat solution, these problems mean alternatives should be investigated.

2.2 Deuce

An interesting alternative approach is provided in Java by the Deuce STM library [8]. With this library the user adds Java annotations to the declaration of a method. This method is then turned into a transaction in its entirety. The transformation of this method is achieved through the use of a Java Agent to perform byte-code rewriting. Java Agents are Java programs that are specified as parameters to the JVM when it is initialised. These programs rewrite Java class files when they are loaded into the JVM, allowing them to seamlessly augment existing class files with additional functionality, in this case adding transactional semantics to methods starting with the `@atomic` annotation. The rewriting uses the ASM framework [4] for rewriting class files. This framework consists of an object for reading class files and a set of objects that implement the visitor pattern [5]. These visitors are then linked together into a pipeline, such that each visitor performs an action on the code they are called on which is provided by the call from the previous visitor. The final visitor writes the modified output back into byte-code that can be written to file, or in this case passed to the class loader.

Using this framework the process of adding transactions is achieved through two steps:

1. An additional copy of each method in each class file is created. These additional methods take an extra parameter in the form of a transactional context, and have all instructions to load and store fields replaced with method calls to this context. In conjunction with these instrumenting calls, all method calls within these methods also have the context added as an extra parameter. This effectively creates two separate code bases, one running as a transaction, the other as normal code without the instrumentation and associated overhead.
2. Methods annotated as atomic are replaced with a method that constructs a transactional context and, using this context, calls the duplicate method constructed from the original replaced method. This provides the means for the control flow to pass back and forth between the uninstrumented non-transactional methods and the instrumented transactional methods. In addition to constructing an instance of a transactional context, this new method also contains all the control logic required to both commit and rollback transactions.

While this approach is very effective and removes the need for the user to adjust their code to use different syntax, it does force transactions to occur at the method level. The net effect of this when adding transactions is that bits of code have to be re-factored out into separate methods, and if the user wants to change the scope of their transactions they have to re-factor these methods again. This in our experience makes programs harder to read and to maintain.

3. Design and Implementation of MUTS

As we feel that none of the solutions discussed in Section 2 are satisfactory for a languages in which transactions are accessible and maintainable to all users, we decided to look at how we could add transactions using traditional block syntax and extra keywords, without the need for different syntactical elements to be added. This is clearly not achievable without modification to the Scala compiler, however we wanted to ensure that these modifications remained as small as possible, and in this section we will detail the design and implementation we used to achieve this.

3.1 Choice of constructs

If the compiler is to be modified to accept transactions, we first need to define what the syntax and semantics of these transactions should be. In the literature [6, 7] there are three basic types of transactional construct, and currently we support all three. These are as follows, complete with the syntax we use to represent them:

- Transactions that will not retry in the event of failure.

```
atomic(Test)
{
  Body
}
```

- Transactions that will retry in the event of failure.

```
atomic(Test)
{
  Body
} retry;
```

- Transactions that will run a different piece of code in the event of failure.

```
atomic(Test)
{
  Body
} orElse
{
  ElseBody
}
```

All of these blocks support the optional inclusion of a set of test conditions that must be met before the transaction can start to execute. This range of constructs and functionality has been included in order to test the completeness of our system, however in a production system it may be decided to reduce this scope to improve ease of use. Library calls could be used to allow user code to abort transactions. These have not been included as it is not clear that user code should have the ability to interact with the transaction in this manner. For example if an abort was added to a function, then that function could only ever be called from within a transaction. However, transactions are intended as a non invasive way of protecting code against race conditions. The inclusion of the information that the function has to be called from a transaction at a system level would undermine this transparency. In our implementation the keywords are prefixed with "TF" to prevent collisions with existing code, but this has been removed within this paper to improve clarity. We will now describe how these constructs are implemented.

4. Scala Compiler Architecture and Modifications

In this section we will give a brief overview of the structure of the compiler, before describing the minimal modifications that we have made in order to support the required block structures as described in Section 3.1.

The Scala compiler is made up of a pipeline consisting of 21 phases. This pipeline starts with the parser that converts a textual input into a tokenised abstract syntax tree. The tree is then transformed by the remaining phases, with each transformation moving it closer to the required output. When the final phase passes over the tree JVM byte-code is produced. The pipeline can be extended by the addition of user defined phases to add functionality, however, this requires knowledge of the complex relationships of the different data structures within the tree. This situation is made more complex by the insertion and use of implicit methods. These are added in several of the phases, and any phase that added transactions would have to occur after all of these. However, the more transformations that have occurred to the tree, the further the data structures are from the initial Scala input and the greater level of understanding that is required of the internal data structures of the compiler. This means making any changes to add functionality at this level requires a high level of understanding, and makes any work extremely vulnerable to changes in the compiler architecture. As a result the only change made directly to the Scala compiler is to the parser, and functionality that cannot be added to this first phase is instead added through a byte-code rewrite when we once again have a strong and guaranteed stable specification of the structures involved. To do this we constructed a Java Agent based on the one produced as part of the Deuce STM [8]. The modifications to the parser will now be discussed in detail, before describing how the implantation of transactions was completed with the Java Agent.

4.1 Parser Modifications

The modifications to the parser consist of adjusting it to accept the three new keywords "atomic", "retry" and "orElse", and interpreting these into the blocks described by the three types of transaction. These modifications take advantage of the same functionality as "if" and "while" blocks. Once these have been interpreted, they are mapped onto the abstract syntax tree using existing Scala functionality to provide the required control flow. These transformations can be seen in Figures 1, 2 and 3. Currently the type of these transformations is Unit, so no information about the transaction escapes the transaction. This could be trivially extended to return the value of the encased user code if required for better integration with Scala.

As can be seen in these Figures, this stage of the transformation adds a context that will be used to store the information required to check for conflicts and commit the computation if there are none. The calls to initiate the conflict check and commit the transaction are also added at this stage. The information to be stored in the context will be added by calls to the context when each read and write is performed; however, it is not yet possible to add in these calls as the implicit methods are yet to be added by the compiler. This instead will be handled by the byte-code rewrite. Leaving this until later not only removes the need for further interference with the compiler, but also has the advantage of not weakening the compile time type checking.

When using library calls, it is necessary to provide the call with sufficient information to update the value that the call relates to if required by the commit. For method variables, this is not possible as the structure of the JVM is deliber-

ately such that one method cannot interfere with the method variables belonging to another method. An alternative technique was chosen that takes advantage of the type inference provided by the Scala type system. With this technique, the set of method variables that are updated, but not declared within a transaction is detected. In Figures 1, 2 and 3, this set is called *transaction_Variables*. These values are then copied to a set of variables labelled *transaction_Variables_Backup*. Having done this the method variables can be updated without requiring any additional instrumentation, as these values are only accessible to the method. If the transaction fails to commit, then the stored values will be copied back over any changes that may have occurred. It may be beneficial to nullify the object values in these copies to facilitate garbage collection, but due to the structure of Scala's type system this involves first determining which of these variables is of type AnyRef and which is of type AnyVal. This cannot be achieved until a later phase in the compiler. However, adding it into a later pass would incur the complications described earlier and thereby breach the minimum changes remit of this work. As a result this optimisation would have to be added during the byte-code rewrite if it is to be included. However, as the lifetime of these values is at most the lifetime of the method call, for the time being this is not being addressed.

All of the transformations include a "try catch" block around the user code. These perform three functions:

1. **To catch any exceptions thrown by the user code and to re-throw these if the transaction commits.** If the transaction fails to commit then the exception will be caught as the semantics of the transaction are that nothing should appear to happen unless the transaction successfully commits.
2. **To catch any exceptions thrown by the transaction mechanism.** This is currently used so that transactions which fail as a result of a commit by another transaction are able to jump out of their current function back to their encasing transaction block. This is vulnerable to user "try catch" blocks that catch all exceptions, but do not re-throw them. This can be overcome in almost all cases simply by throwing a new exception if an access to the context is made again before the exception propagates to the correct "try catch" block. The unsupported case is where no reads or writes are performed before re-entering the "try catch" block that caught the last exception. For example if it is encapsulated by a "while" block in which the guard is simply the value true and no further code is present. Such cases should not appear in sensible code within a transaction, or for that matter within code that forms a component part of an application.
3. **To provide a marker in the byte-code, describing the sections of the code which are transactional.** If byte-code rewriting is to be used to add in library calls for transactions once the Scala compiler has completed, it is important that the correct segments of byte-code can be identified. This is achieved through the capture of the exception class TransactionArea. This exception class is created for this purpose, is never thrown, and its capture is removed during the byte-code rewrite. The effect of its presence is to mark all regions of the byte-code output of the compiler that form part of a transactional block. This is required as transactional code cannot be identified simply by looking for the call to initialise the context and the call to commit, as the compilation may have reordered the byte-code. However, as any reordering will be applied to the scope of the exception handlers as well, a record of such transformations is maintained. So the lines of code that are covered by the handler for this exception are the lines that the byte-code rewrite must treat as transactional.

4.2 Byte-code Rewrite

Having added in the control logic and the handling of method variables in the parser, the remaining functionality to handle the accesses to class variables is added by rewriting the original byte-code. To do this we modified the existing Deuce STM rather than implement a new STM from scratch. Deuce was chosen as it provides a working implementation that rewrites byte-code and is well tested. It also, instead of implementing a single STM backend implementation, provides an interface that is capable of interacting with a range of backend STMs. As described in Section 2.2, Deuce uses the ASM [4] framework to perform the class rewriting. Using this framework it performs two functions:

1. Add duplicate methods to the compiled classes. These duplicate methods have the following properties:
   - They take a context as an additional parameter.
   - They replace all field accesses with method calls to the context which record the reads and writes.
   - They replace all method calls with method calls that use this extra context parameter.
2. Replace any method annotated as atomic with a method that performs the following actions:
   - Creates a context to hold the transactional information.
   - Calls this method's corresponding duplicate using the newly created context as the extra parameter.
   - Manages commits, retries and exceptions of the transaction started through these actions.

While Deuce is highly tested with Java, it is incompatible with Scala in its current form as it instruments all class variables. This is a problem because, unlike Java, Scala byte-code uses the "this" parameter to set class fields representing object values before it calls the constructor for the super

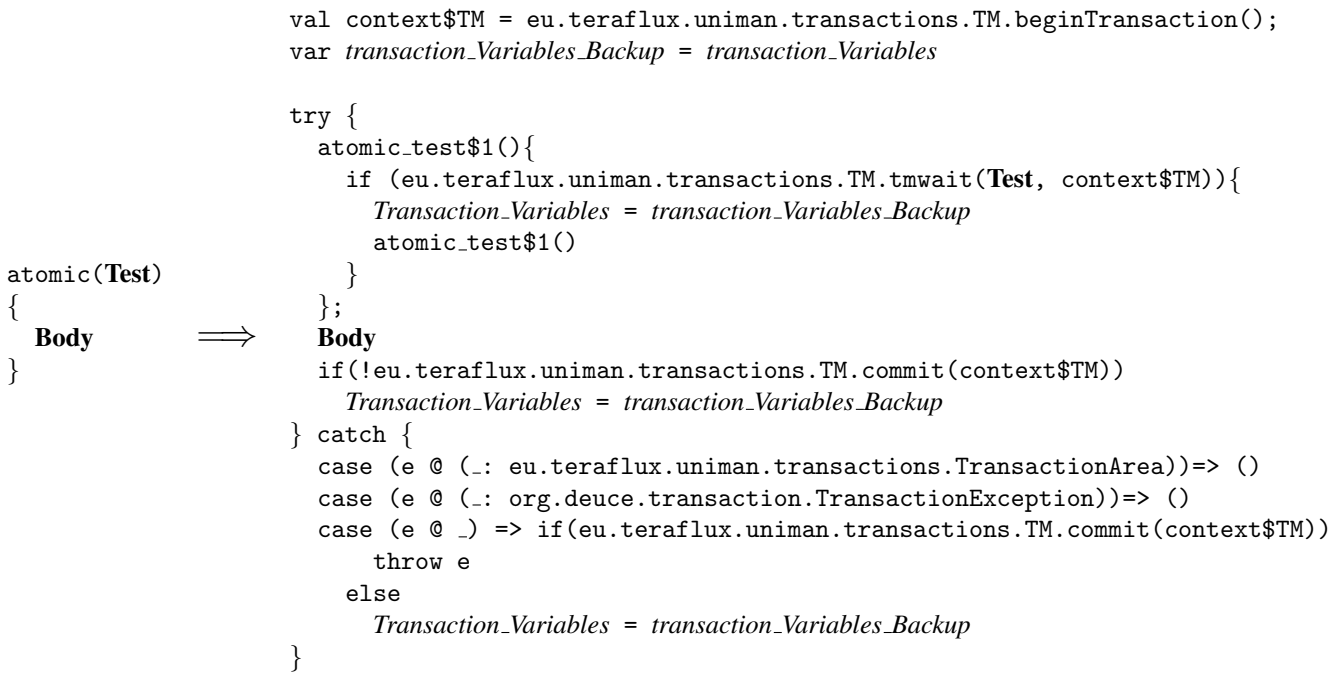

Figure 1. Transformation of a basic transaction block into existing Scala abstract syntax tree constructs. On the left the user code, on the right the parser generated code.

class. This is perfectly legal byte-code, but to change these assignments to calls on the transactional context that take "this" as a parameter is not valid as the class is not yet fully defined and as such cannot yet be passed into methods. The result is that using Deuce on such Scala class files results in the JVM detecting a type error and aborting. However, as these fields are only values, not variables, it is safe to modify the rewrite to ignore them as they will never change so do not need instrumenting.

As well as changing Deuce to prevent it adding illegal method calls, the other change required is to the behaviour when moving in and out of a transaction. Now instead of detecting an annotation before visiting a method and choosing the method visitor accordingly, it is necessary to detect the code within a method that is transactional and instrument it appropriately. As discussed in Section 4.1 this is achieved through the addition in the parser of a "try catch" block which in turn creates an exception handler for the required region. Any field access that appears within the scope of this exception handler should be instrumented and any method calls should be augmented with the context as an extra parameter. These changes require the presence of a context, however, unlike in the initial Deuce implementation, the code to create the context is added in the parser, not in the byte-code rewrite. This means that the Scala compiler, not the byte-code rewrite, decides the location where the context is to be stored. As such it is now also necessary to detect the call to construct the context and record where the result from this action is stored in order to gain all the required information for the instrumentation.

4.3 Nesting

Nesting is handled at two levels. At the parser level the parser is aware of the nesting of transactions within the same method. When this occurs, it changes the calls to `beginTransaction()` to take the context of the encompassing transaction as a parameter. This allows this method to encode into the newly returned context the required information about its encasing transactions and their contexts.

If the nesting occurs outside of the method that created the encasing transaction it will be met when the byte-code is rewritten. At this stage the same effect as the parser transformation can be achieved by allowing the byte-code rewrite to add the context parameter to the beginTransaction method when it appears within a method that is being instrumented. As this method already exists and we prevent the rewrite from rewriting the STM libraries, the net effect is that we call the beginTransaction method that takes a context as an argument.

Some of this complexity could be removed by maintaining a single context per thread stored and accessed through the use of a statically stored ThreadLocal object. However, this would add considerable additional overhead as an additional method call would be required to retrieve the context for every transactional read and write. This would be in addition to the accesses that are subsequently made to the context.

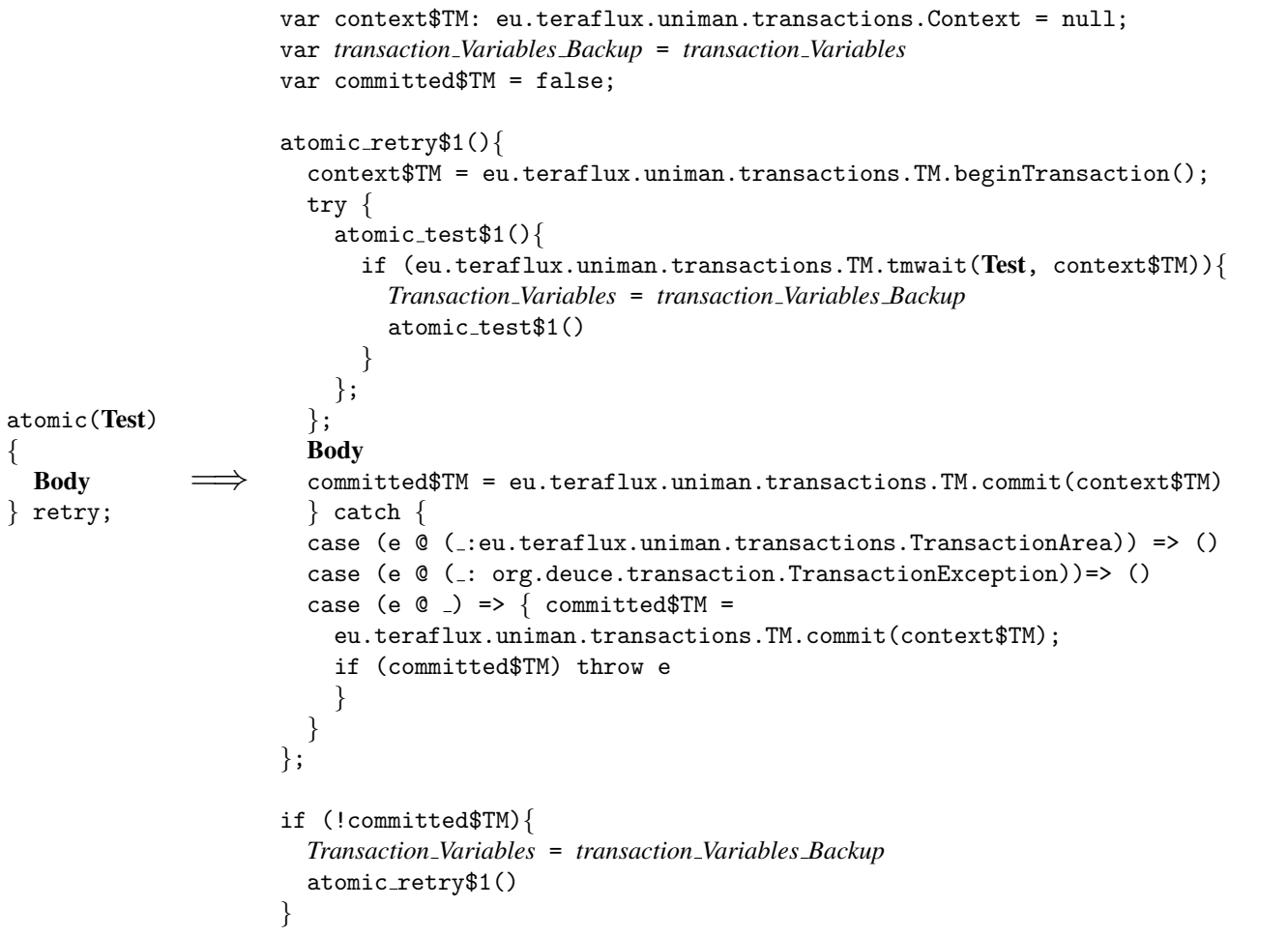

Figure 2. Transformation of a transaction block with retries into existing Scala abstract syntax tree constructs. On the left the user code, on the right the parser generated code.

4.3.1 Integration with the Scala Compiler

The byte-code rewrite is intended to be added as a Java Agent at runtime, and in general there is no reason not to do this. However, there is also nothing to prevent this transformation being performed immediately after the files have been constructed at compile time. Such a compilation strategy would require all existing class files being used by the compiled code to already be instrumented. If they were compiled with this new compilation strategy this would be the case, but when integrating with Java code or existing Scala class files they would not be instrumented. However the instrumentation of existing class files can be achieved by a one off code transformation on the required extra classes. This transformation would be exactly the same as the automated transformation at the end of the compilation script.

5. Conclusion

In this paper we have argued that the current approaches to implementing transactional memory fragments the code through restrictions on the syntax or granularity of the areas that can be covered by these techniques. This in turn adversely affects accessibility of this functionality and the construction, readability, and maintainability of codes using it. This position is reached as a result of the implementers of transactional memory choosing implementation strategies that require no changes to the underlying compiler. This is done for the very valid reason that it means that when the compiler changes the STM does not have to change too. We have instead decided to make the minimum changes to the compiler to implement transactional memory in Scala such that there would be no syntax changes between transactions and standard code, and no restriction on the granularity of the transactions. This aim has been achieved by modifying the parsing of the initial input to accept three new keywords that it uses to identify areas of code that are transactional. Having identified these it encodes them into the abstract syntax tree using existing constructs of the abstract syntax tree. As these changes are restricted to just the parser, the bulk of the compiler remains untouched. This encoding in the parser includes the control logic for the transaction, the handling of

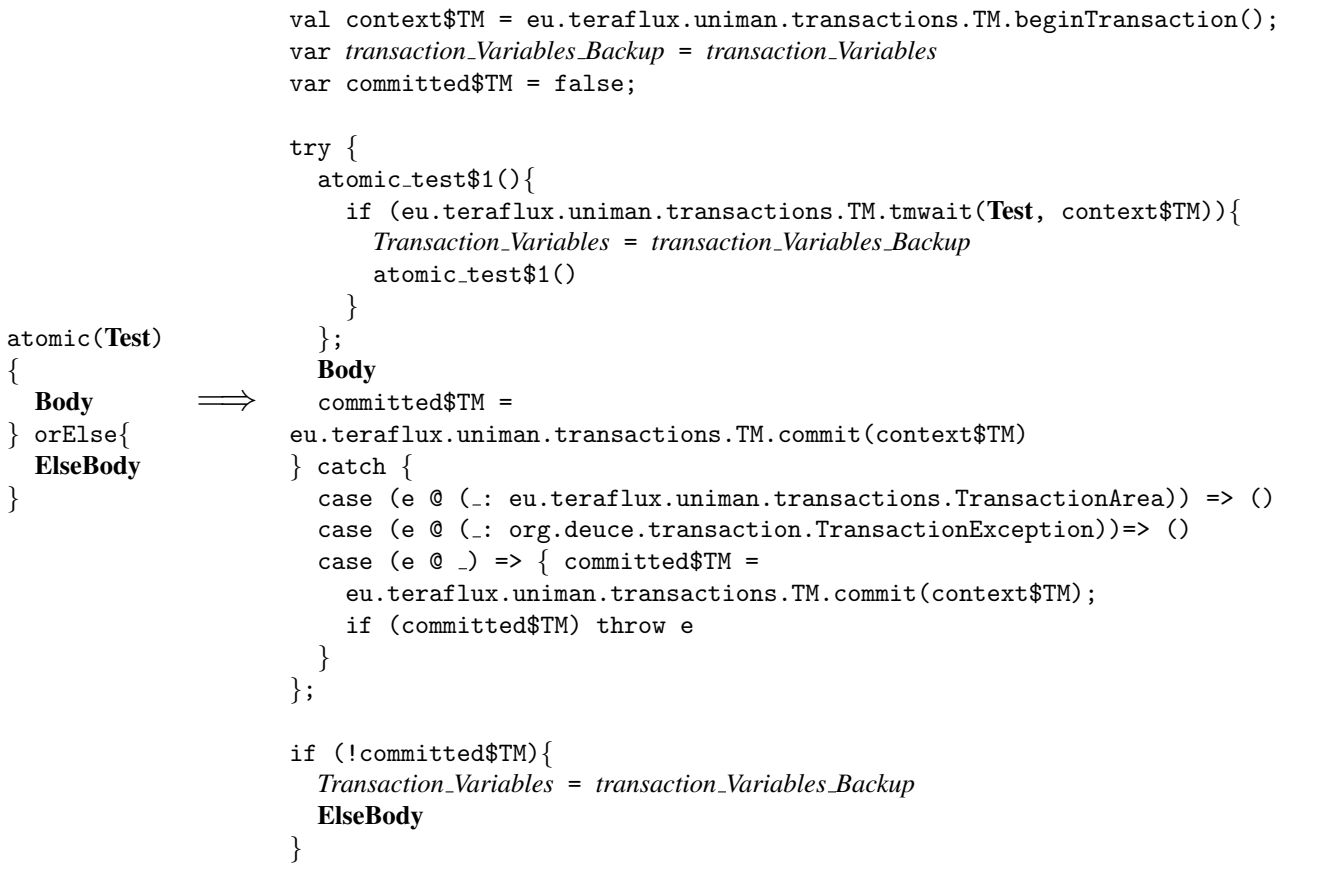

Figure 3. Transformation of a transaction block with an alternative execution in the event of failure into existing Scala abstract syntax tree constructs. On the left the user code, on the right the parser generated code.

method variables and the identification of the body of the transaction such that the remaining changes can be added later. The process of adding the remaining instrumentation of the transaction is then performed by a byte-code rewrite once the Scala compiler has finished. This ensures that all implicit elements have been added before this commences and guarantees that this step is independent of the compiler design. The byte-code rewrite is performed using the ASM framework and a modified version of the Deuce STM.

5.1 Strong vs Weak Isolation

The construction of transactional blocks from existing Scala constructs means that reference objects cannot be used to hold transactional data. This means that, unlike CCSTM, a different syntax is not required for assignment inside and outside of transactions. However, this also means that, like in DeuceSTM, the type system cannot currently be used to ensure strong isolation of transactional data. An alternative to using the type system to enforce strong isolation through object types is to adjust the memory model to create different types of memory, including memory that is defined as transactional. Such memory could be distinguished by its address space [1], or through additional record keeping by the runtime environment, however such work is outside of the scope of this paper.

5.2 Performance

As this implementation of transactional memory uses the same backend code as Deuce and only changes the method by which it is integrated into the class files, the performance is dependent on the particular implementation of the backend being used. It does have the opportunity to outperform Deuce through a lack of the overhead created by the need to fragment code with additional method calls to move between the transactional and non-transactional sections. However, with a method call being made for every read and write within the transaction, this performance gain will generally not be observable. The functional side of Scala provides many opportunities to insert optimisations, and in the future we hope to consider these further, while still maintaining the mandate of making the minimal possible changes to the compiler.

Acknowledgments

The authors would like to thank the European Communitys Seventh Framework Programme (FP7/2007-2013)

for funding this work under grant agreement no 249013 (TERAFLUX-project). Dr. Luján is supported by a Royal Society University Research Fellowship.

References

[1] M. Abadi, T. Harris, and M. Mehrara. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. *SIGPLAN Not.*, 44: 185–196, February 2009. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1594835.1504203. URL `http://doi.acm.org/10.1145/1594835.1504203`.

[2] R. Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, second edition, 1998.

[3] N. G. Bronson, H. Chafi, and K. Olukotun. CCSTM : A library-based STM for Scala. In *The First Annual Scala Workshop at Scala Days 2010*, April 2010.

[4] E. Bruneton, R. Lenglet, and T. Coupaye. Asm: A code manipulation tool to implement adaptable systems. In *Adaptable and extensible component systems*, 2002.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, 1995. ISBN 0201633612.

[6] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM. ISBN 1-59593-080-9. doi: 10.1145/1065944.1065952. URL `http://dx.doi.org/10.1145/1065944.1065952`.

[7] T. Harris, J. R. Larus, and R. Rajwar. *Transactional Memory, 2nd edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2010.

[8] G. Korland, N. Shavit, and P. Felber. Noninvasive concurrency with Java STM. In *Third Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG-3)*, Pisa, Italy, January 2010.

[9] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999. ISBN 0201432943.

[10] M. Odersky, L. Spoon, and B. Venners. Programming in scala: [a comprehensive step-by-step guide], 2008.

[11] Scalable Solutions AB. Akka project, February 2011. URL `http://akka.io/`.

[12] C. T. Wu. *An Introduction to Object-Oriented Programming with Java 2nd Edition*. McGraw-Hill, Inc., New York, NY, USA, 2nd edition, 2000. ISBN 0072396849.