Safe Typing for Transactional vs. Lock-Based Concurrency in Multi-threaded Java

Thi Mai Thuong Tran and Olaf Owe and Martin Steffen
Department of Informatics
University of Oslo
Oslo, Norway
{*tmtran,olaf,msteffen*}*@ifi.uio.no*

***Abstract*—Many concurrency models have been developed for high-level programming languages such as Java. A trend here is towards more flexible concurrency control protocols, going beyond the original Java multi-threading treatment based on lexically-scoped concurrency control mechanism. Two proposals supporting flexible, non-lexical concurrency control are the lock-handling via the `Lock`-classes in Java 5 and *Transactional Featherweight Java* (TFJ), an extension of Featherweight Java by transactions. Even if these two take quite different approaches towards dealing with concurrency —"pessimistic" or lock-based vs. "optimistic" or based on transactions— the added flexibility of non-lexical use of the corresponding concurrency operators comes at a similar price: improper usage leads to run-time exceptions and unwanted behavior. This is in contrast with the more disciplined use under a lexically scoped regime, where each entrance to a critical region is syntactically accompanied by a corresponding exit (as e.g. with traditional `synchronized` methods or as with so-called atomic blocks).**

To assure safe use of locking, resp. transactions in these settings, we present in this paper abstractions in the form of two static type and effect systems, which make sure that for instance, no lock is released by a thread which does not hold it, resp., that no commit is executed outside any transaction. We furthermore compare the two mentioned approaches to concurrency control on the basis of these type abstractions.

I. INTRODUCTION

With the advent of multiprocessor and multi-core architectures and distributed web-based programs, effective parallel programming models and suitable language support are becoming main stream. This includes Java as the current "default" object-oriented language, where much research and development concerning support for concurrency is done.

How to syntactically represent corresponding mechanisms in the language may, of course, vary. One option is lexical scoping, for instance based on `synchronized`-methods/blocks for lock handling in Java, or using an *atomic* keyword designating protected regions, or similar approaches. One trend, however, is towards more flexible ways going beyond more traditional lexically-scoped concurrent control where the region protected against unwanted interference can be started and finished freely.

In this paper, we are concerned with two typical recent proposals in that field, one is the lock-handling as introduced in Java 5, and the other extends Java by transactions, namely *Transactional Featherweight Java* (TFJ) [1]. Where Java 5 uses lock and unlock for acquiring and releasing re-entrant locks, TFJ uses onacid and commit as keywords to start, resp. terminate a transaction. Even if these two take quite different approaches towards dealing with concurrency —"pessimistic" or lock-based vs. "optimistic" or based on transactions— the added flexibility of non-lexical use of the corresponding concurrency operators comes at a similar price: improper use leads to run-time exceptions and unwanted behaviors. This is in contrast with the more disciplined use under a lexically scoped regime, where each entering a critical region is syntactically accompanied by a corresponding exit of it (as for instance with traditional `synchronized` methods). In this paper we compare the two approaches, in particular we present and compare two type-and-effect based abstractions for both concurrent models that prevent unsafe uses of the concurrency operators.

The paper is organized as follows. After comparing the two languages in Section II, we sketch corresponding type and effect systems for each in the Section III. Section IV gives the conclusion by discussing related and future work.

II. TRANSACTIONAL VS. LOCK-BASED CONCURRENCY CONTROL

In this section we shortly highlight aspects of transactional Java resp. of lock handling in Java 5, relevant for our abstractions. Concerning details for safe commits in TFJ, we refer to [2].

Transactional Featherweight Java: Transactions, a well-known and successful concept originating from database systems [3], [4], have recently been proposed to be directly integrated into *programming languages.* As mechanism for concurrent control, they can be seen as a high-level, more abstract, and more compositional alternative to more conventional means for concurrent control, such as locks, semaphores, monitors, etc.

A recent proposal for integrating transactional features into programming languages is TFJ. The start of a transaction in TFJ programs is marked by onacid keyword and the end by commit keyword. The transactional model of TFJ is quite general. Transactions can be started and committed with *non-lexical* scope. It supports *nested* transactions which means a transaction can contain one or more child transactions, which is very useful for composability and partial rollback. Furthermore, TFJ supports *multi-threaded* transactions, i.e., one transaction can contain internal concurrency. TFJ threads in a parent transaction can execute concurrently with threads in nested transactions. To commit an entire parent transaction, all its child threads must join (via a commit), in other words all threads in the parent transaction including the parent thread must commit at the same time. Table I sketches the abstract syntax used in our analysis, a variant of Featherweight Java.

$$
\begin{array}{rcll}
P & ::= & \mathbf{0} \mid P \parallel P \mid t\langle e\rangle & \text{processes/threads} \\
L & ::= & \mathsf{class}\ C\{\vec{f} : \vec{T}; K; \vec{M}\} & \text{class definitions} \\
K & ::= & C(\vec{f} : \vec{T})\{\mathsf{this}.\vec{f} := \vec{f}\} & \text{contructors} \\
M & ::= & m(\vec{x}{:}\vec{T})\{e\} : T & \text{methods} \\
e & ::= & v \mid v.f \mid v.f := v & \text{expressions} \\
 & \mid & \mathsf{if}\, v\, \mathsf{then}\, e\, \mathsf{else}\, e \mid e;\ e \mid v.m(\vec{v}) & \text{expressions} \\
 & \mid & \mathsf{new}\ C(\vec{v}) \mid \mathsf{spawn}\ e & \\
 & \mid & \mathsf{onacid} \mid \mathsf{commit} & \\
v & ::= & r \mid x \mid \mathsf{null} & \text{values}
\end{array}
$$

Table I
ABSTRACT SYNTAX

Java 5 locks: The built-in support for concurrent control in Java is lock-based; each object comes equipped with a (re-entrant) lock, which can be used to specify synchronized blocks and, as a special case, synchronized methods. The lock can achieve mutual exclusion between threads that compete for the lock before doing something critical. Thus, the built-in, lock-based (i.e., "pessimistic") concurrent control in Java offers *lexically scoped* protection based on mutual exclusion. While offering basic concurrent control, the scheme has been criticized as too rigid, and consequently, Java 5 now supports explicit locks with *non-lexical* scope. The `ReentrantLock` class and the `Lock` interface allow more freedom. Thus, on a purely syntactical level, Java 5 allows to handle locks similar as TFJ handles transactions: instead of onacid and commit in TFJ, the methods lock and unlock are used. Basically, the syntax therefore is the one of Table I with the mentioned replacement (and additionally, we consider exceptions, which are good practice when programming with locks.) Apart from those syntactical similarities, there are of course differences especially wrt. failure and progress properties. See e.g., [5] for a discussion of such differences.

Comparison

Besides the more behavioral differences, such as different progress guarantees, deadlocking behavior etc., the lock handling in Java 5 and the transactional model of TFJ differ in the following aspects, which are relevant for type analysis. The differences are summarized in Table II.

One basic difference is that we proposed a *static* scheme to catch commit errors, whereas in Java, improper use of locking and unlocking is checked at *run-time.* In Section III-B, we sketch a static type and effect system to avoid such run-time checks. Both languages, as mentioned, have all the flexibility of non-lexical scoping. The rest of Table II deals with the structure of protected areas (the transaction or the execution protected by a lock) and the connection to the threading model.

	Java 5.0	TFJ
when?	run-time	compile time
non-lexical scope	yes	yes
program level identity	yes	no
re-entrance	yes	no
nested transactions (critical sections)	no	yes
internal multi-threading	no	yes

Table II
TRANSACTIONAL FEATHERWEIGHT JAVA AND EXPLICIT LOCKS OF JAVA

One difference is that locks have an identity available at the program level, whereas transactions have not. Furthermore, locks and monitors in Java are *re-entrant,* i.e., one particular thread holding a lock can recursively reenter a critical section or monitor. Re-entrance is not an issue in TFJ: a thread leaves a transaction by committing it (which terminates the transaction), hence re-entrance into the same transaction makes no sense. Transactions in TFJ can be nested. Of course, in Java, a thread can hold more than one lock at a time; however, the critical sections protected by locks do not follow a first-in-last-out discipline, and the sections are not nested as they are independent. For nested transactions in contrast, a commit to a child transaction is propagated to the surrounding parent transaction, but not immediately further, until that parent commits its changes in turn. Finally, TFJ allows concurrency within a transaction (supporting multi-threaded transactions), whereas monitors and locks in Java are meant to ensure mutual exclusion. In particular, if an activity inside a monitor spawns a new thread, the new thread starts executing *outside* any monitor, in other words, a new thread holds *no* locks.

III. STATIC ABSTRACTIONS FOR SAFE CONCURRENCY CONTROL

The flexibility of non-lexical use of onacid and commit comes at a cost: not all usages "make sense". In particular, it is an error to perform a commit without being inside a transaction. This similarly happens in Java 5 when releasing a lock, i.e., calling the `unlock` method on a lock without actually owning it.

In Section III-A and III-B, we sketch two type and effect systems which statically allow to prevent such errors. The two effect systems share some similarities: The basic idea in both is to abstractly keep track of the number of onacids and commit, resp. of lock and unlock. The differences in the analysis, on the other hand, reflect the differences discussed earlier and summarised in Table II.

A. Type and effects for transaction handling

The purpose of our formal system is to determine correct usage of starting and committing transactions, in particular to avoid committing when one is not inside a transaction. We call such erroneous situations *commit errors*. To prevent them, we basically keep track per thread of the number of onacids minus the number of commits encountered (which we call the *balance*) at a given point in the code. The general form of a judgment for a single expression (i.e., inside one thread) is of the form:

$$\Gamma;\ n_1 \vdash e : T \ \&\ n_2, S \tag{1}$$

The judgment is read as "under the assumption Γ, the expression e has the type T and evaluating e which starts with a balance of n_1 will lead to a balance of n_2". The multi-set S of integers mentioned in the post-condition takes care of the balance of *new* threads spawned by e.

The situation is slightly more involved, as TFJ supports *nested* and *multi-threaded* transactions. For instance, to commit a transaction, all threads inside must *join* to commit at the same time. To adequately take care of this form of multi-threading inside a transaction, the multi-set S of equation (1) is needed, which calculates the balance for potentially all threads concerned, i.e, all threads (potentially) spawned during that execution.

Table III sketches four typical rules for expressions, concentrating on the aspects of transaction handling and multi-threading.

$$\frac{}{\Gamma;\ n \vdash \mathsf{onacid} : \mathsf{Void}\ \&\ n+1, \emptyset}\ \text{T-ONACID}$$

$$\frac{n \geq 1}{\Gamma;\ n \vdash \mathsf{commit} : \mathsf{Void}\ \&\ n-1, \emptyset}\ \text{T-COMMIT}$$

$$\frac{\Gamma;\ n_0 \vdash e_1 : T_1\ \&\ n_1, S_1 \qquad \Gamma;\ n_1 \vdash e_2 : T_2\ \&\ n_2, S_2}{\Gamma;\ n_0 \vdash e_1;\ e_2 : T_2\ \&\ n_2, S_1 \cup S_2}\ \text{T-SEQ}$$

$$\frac{\Gamma;\ n \vdash e : T\ \&\ n', S}{\Gamma;\ n \vdash \mathsf{spawn}\ \ e : \mathsf{Void}\ \&\ n, S \cup \{n'\}}\ \text{T-SPAWN}$$

Table III
TYPE AND EFFECTS FOR TFJ

The first basic two rules (cf. rule T-ONACID and T-COMMIT) are to start and commit a transaction. The dual two commands of onacid and commit simply increase, resp. decrease the balance by 1. In a sequential composition (cf. rule T-SEQ), the effects are accumulated. Creating a new thread by executing spawn e has the type of Void and does not change the balance of the executing thread (cf. rule T-SPAWN). The spawned expression e in the new thread is analyzed starting with the same balance n in its pre-state.

The type and effect system is not only concerned with checking expressions, the declarations of methods are generalized, as well. We do not require that method bodies are balanced (which would correspond to a lexically scoped discipline for transactions): a method may perfectly well be used to implement code for committing a transaction. To ensure, however, that this flexibility does not lead to commit errors, the declaration of a method does not only contains the expected balance of the method body, but also a requirement on where that method can be used as a form of precondition. So the *specification* of a method, as far as its effects are concerned, is of the form

$$m(\vec{x} : \vec{T})\{e\} : n_1 \rightarrow n_2\ ,$$

where n_1 is the balance after evaluating the previous expression before calling the method m and n_2 is the balance after evaluating the body e of the method. The corresponding rule looks as follows:

$$\frac{\vec{x}{:}\vec{T},\ \mathsf{this}{:}C;\ n_1 \vdash e : T\ \&\ n_2, \{0, \ldots\}}{\vdash m(\vec{x} : \vec{T})\{e\} : T\ \&\ n_1 \rightarrow n_2, \{0, \ldots\}}\ \text{T-METH}$$

In this rule, we require that all spawned threads in the method body must have the balance 0 after evaluating the expression e, that the balance of the method itself has the form $n_1 \rightarrow n_2$ where n_1 is interpreted as pre-condition, i.e., it is safe to call the method *only* in a state where the balance is at least n_1. The number n_2 as the post-condition corresponds to the balance after exiting the method, when called with balance n_1 as pre-condition. The precondition n_1 is needed to assure that at the call-sites the method is only used where the execution of the method body does not lead to a negative balance.

B. Type and effects for lock handing

As mentioned, in comparison to earlier Java versions, Java 5 introduced a more flexible way to use re-entrant locks. The types we use in our calculus are given in equation (2).

$$T \quad ::= \quad C \mid B \mid \mathsf{Void} \mid L \tag{2}$$

In a nominal type system, class names C serve as types. In addition, B represents basic types (left unspecified) such as booleans, integers etc. Void expresses the absence of a value, i.e., it is used for expressions evaluated for their side-effect, only. Specially, the distinguished type L is used for *ReentrantLock* objects. In order to capture effects related to locks, we use locks environment Δ as an additional part to the type environment for building the type and effect system. The Δ's are the abstractions of balances of the locks, i.e., they can be seen as mapping of type $L \rightarrow \mathsf{Nat}$ and defined in (3).

$$\Delta \quad ::= \quad \emptyset \mid \{l_1 : n_1,\ l_2 : n_2 \ldots\} \tag{3}$$

Here $\Delta = \emptyset$ says that there is no lock taken by some thread; it can be an empty set or a set of free locks $l[0]$. $(l : n)$ says that the lock l is taken n times by some thread.

Next we present a generalization of the type and effect system to deal with locks. The most important difference is that locks have an *identity* and thus can be shared between threads. Indeed, locks are *meant* to be shared between threads and one could say, a crucial advantage of transactions over lock-based concurrency from the perspective of the user is that the user can obtain non-interference *without* the need of identifying individual locks.

In our setting, we assume in the analysis that the locks are statically known. For example, in concrete Java, the locks are public static fields of classes in the system. Under this assumption the generalization from the transactional setting is straightforward: it is no longer enough to use the *nesting depth* inside transactions, we need to keep track of the "balance" of a thread individually *per lock*. So the type judgments for expressions are of the form:

$$\Gamma;\Delta_1 \vdash e : T \,\&\, \Delta_2 \tag{4}$$

It is read as ("under lock and type assumptions Γ, expression e has type T and some effect which changes Δ_1 into Δ_2").

The *type environment* Γ keeps the type assumptions for local variables, basically the formal parameters of a method body and the fields. Environments Γ are of the form $x_1{:}T_1,\ldots,x_n{:}T_n$, where we silently assume the x_i's are all different. This way, Γ is also considered as a finite mapping from variables to types. By $dom(\Gamma)$ we refer to the domain of that mapping and write $\Gamma(x)$ for the type of variable x in Γ. Furthermore, we write $\Gamma,x{:}T$ for extending Γ with the binding $x{:}T$, assuming that $x \notin dom(\Gamma)$.

The *lock environment* Δ keeps the lock assumptions for locks, whether a lock is free (denoted by 0) or taken, where the natural number indicates how many times it is taken, which captures re-entrance. Since the locks assure mutual exclusion, a lock can be taken only by one thread at a time, which means the static analysis does not need to take into account interference between concurrent threads when analysing these balances. Another difference between the judgements for lock handling and the one for transactions is that the *multi-set* in the post-environment is not needed here (cf. again equation (1)).

This highlights another difference between transactions and locks from Table II, namely transactions in TFJ allow internal non-determinism whereas locks assure mutual exclusion: A newly spawned thread in Java does *not* "inherit" the locks of its spawning thread, whereas a new thread in TFJ starts executing *inside the same* transactions as its spawner. The latter is the reason why one needs the multi-set S of balances of spawned threads in equation (1).

Typical rules of the type and effect system are given inductively in Table IV. The first 2 rules deal with acquiring and releasing a lock. Here each lock has a name; therefore, the dual two method invocations on the lock object l simply increase, resp. decrease the balance of that specific lock by 1, leaving all others in Γ unchanged. The first rule T-LOCK deals with acquiring a lock, which increases the balance of the lock by 1 in the post-configuration. The next rule T-UNLOCK deals with unlocking. Note, of course, there is no rule that allows unlocking a free lock, i.e., such a situation does not type check and represents a *lock error*. Rule T-SEQL is the standard rule for sequential composition, where the post-condition of the first expression is taken as the precondition of the second. As for the typing part, the type of the sequential composition corresponds to the type of the second expression, but the effect is accumulated. This rule assumes that no exception is raised in e_1; we

$$\frac{\Gamma \vdash l : L}{\Gamma;\Delta,l{:}n \vdash l.\mathsf{lock} : \mathsf{Void} \,\&\, \Delta,l{:}(n+1)}\ \text{T-LOCK}$$

$$\frac{\Gamma \vdash l : L \quad n \geq 1}{\Gamma;\Delta,l{:}n \vdash l.\mathsf{unlock} : \mathsf{Void} \,\&\, \Delta,l{:}(n-1)}\ \text{T-UNLOCK}$$

$$\frac{\Gamma;\Delta \vdash e_1 : T_1 \,\&\, \Delta_1 \quad \Gamma;\Delta_1 \vdash e_2 : T_2 \,\&\, \Delta_2}{\Gamma;\Delta \vdash e_1;\ e_2 : T_2 \,\&\, \Delta_2}\ \text{T-SEQL}$$

$$\frac{\Gamma;\emptyset \vdash e : T \,\&\, \emptyset}{\Gamma;\Delta \vdash \mathsf{spawn}\ e : \mathsf{Void} \,\&\, \Delta}\ \text{T-SPAWNL}$$

$$\frac{\vec{x}{:}\vec{S},\mathsf{this}{:}C;\Delta_1 \vdash e : T \,\&\, \Delta_2}{\vdash m(\vec{x}:\vec{S})\{e\} : \vec{S} \to T \,\&\, \Delta_1 \to \Delta_2}\ \text{T-METHL}$$

Table IV
TYPE AND EFFECTS FOR JAVA 5

omit exceptions here for simplicity. Creating a new thread by $\mathsf{spawn}\,e$ does not change the balance of the lock in the executing thread (cf. rule T-SPAWNL). The spawned expression e in the new thread is analyzed starting from the abstraction where all locks are assumed to be free, i.e., more precisely, from the perspective of the thread that executes the spawned e, no lock has been taken yet. Note again the contrast to the corresponding rule T-SPAWN for the transactional setting. The final rule T-METHL deals with method declaration. Note that the effect on the lock is assumed to be part of the *interface specification* of the method, in the same way as the input/output types are.

IV. CONCLUSION

In this paper, we have presented a comparison between TFJ, an extended version of Featherweight Java with support for transaction-based concurrent control, and Java 5, an extended version of Java for lock-based concurrent control. Both languages provide users syntactic constructs to deal with transactions and locks with non-lexical scope. TFJ paper [1] is not concerned with static analysis, but develops and investigates two different operational semantics for TFJ that assure transactional guarantees.

As mentioned, however, the flexibility of TJF may lead to run-time errors when executing a commit outside any transaction; we called such situations *commit-errors*. Java 5.0 also introduces a similar problem when it allows users to explicitly use operations *acquiring* and *releasing* lock objects with non-lexical scope. It is always possible that users can incorrectly use these operations, and hence cause unexpected situations. Java 5 deals with this by throwing an exception which interrupts the whole system. Therefore, in this paper, we sketched some rules of our type and effect systems for both TFJ and Java 5 to statically prevent such errors. We proved the soundness of our type and effect system for TFJ in [2] resp. the corresponding technical report. Further considerations about exceptions related to incorrect usage of locks which might happen in

Java 5 will be discussed in our future work.

Future work

The work presented here can be extended to deal with more complex language features, for instance when dealing with higher-order functions. In that setting, the effect part and its connection to the type system become challenging. Furthermore, we plan to adopt the results for a different language design, more precisely to the language Creol [6], which is based on asynchronously communicating, active objects, in contrast to Java, whose concurrency is based on multi-threading. We plan to use generalize the techniques described for Java locking to handle *dynamic* creation of locks, as well. That will complicate the account quite a bit, as one has to deal with aliasing and passing around identities of newly created locks. Interesting and of practical relevance is also to extend the system from considering type and effect checking to type *inference*, potentially along the lines [7].

Related work

There has been a number of further proposals for integrating transactional features into programming languages, e.g. AtomCaml [8], X10 [9], Fortress [10], Chapel [11]. [12] proposes to reconcile transactions and locking in the context of Java monitors. The goal there is to "get the best of both worlds", i.e. pessimistic, lock-based concurrency in high-contention situations and using transactional computing when there is little contention. The formal development to assure sound co-existence of lock-based and transaction-based implementation is based on a formal calculus similar to the one used here (CJ "Classic Java" [13]). The paper [14] presents the AME calculus, a calculus for *automatic mutual conclusion*, a concept proposed in [15]. The sequential core is a λ-calculus with references and imperative update, extended by the possibility to create asynchronous threads and means for atomic execution. Unlike other approaches, where the user is required to mark parts of the code intended for atomic execution, in AME, atomic execution is the default. For code parts where transactional behavior is not intended or possible (for instance, legacy code from libraries) can be marked as unprotected. A calculus and a proof method (implemented in the tool QED) for atomic actions is presented in [16].

For transactional languages, lexical scope for transactions, so-called atomic blocks, have been proposed, using e.g., an `atomic`-construct or similar. Examples are Atomos [17], the AME calculus [14], and many proposals for software transactional memory [18], [19], [11]. Besides, many early language designs, especially for data base programming, supported non-lexical scoping of the transactional constructs, cf. e.g. CICS [20], Camelot [21], Argus [22]. A recent proposal to integrate software transactional memory into a full-fledged general purpose language is Clojure [23], an extension of Lisp.

Static analysis is a well-established method to assure desired properties ranging from resource consumption (e.g., concerning memory, time ...), absence of deadlocks and race conditions. When dealing with concurrency, most authors (e.g., [24], [25] ...) focus on avoiding data races and deadlocks, especially for multi-threaded Java programs. Static type systems have also been used to impose restrictions assuring transactional semantics, for instance in [26], [14], [15]. A type system for *atomicity* is presented in [27], [28]. Also the Rcc/Java type system tries to keep track of which locks are held (in an approximate manner), noting which field is guarded by which lock, and which locks must be held when calling a method. [29] present a Hoare-logic, more precisely a separation logic, for re-entrant locks, but without exceptions. The treatment of the locks, resp. the transactions in this paper is related also to type systems governing *resource usage* where the possession of the lock or the having started a transaction is corresponds to the resource in question. There have been quite a number type-based approaches to assure proper usage of resources of different kinds (for instance file access, i.e., to govern the opening and closing of files). See[7] for a recent, rather general formalization for, what the authors call, the resource usage analysis problem (see the paper also for further pointer to the literature for approaches to safe resource usage). Unlike the type system explained here, [7] consider also type *inference* (or type reconstruction). Their language, a variant of the λ-calculus, however, is sequential. The approach is applied in [30] in a concurrent setting for the π-calculus. [31], [32] present a type system for statically assuring proper lock handling for the JVM, i.e., on the level of byte code. Their system assures what is known as *structured locking*, i.e., (in our terminology), each method body is balanced as far as the locks are concerned, and at no point, the balance reaches below 0. Since the work does not consider non-lexical locking as in Java 5, the conditions apply *per method* only.

Acknowledgements

We thank the anonymous reviewers for their helpful comments and suggestions. The work has been partly supported by the EU-project and FP7-231620 HATS (Highly Adaptable and Trustworthy Software using Formal Methods).

References

[1] S. Jagannathan, J. Vitek, A. Welc, and A. Hosking, "A transactional object calculus," *Science of Computer Programming*, vol. 57, no. 2, pp. 164–186, August 2005.

[2] T. Mai Thuong Tran and M. Steffen, "Safe commits for Transactional Featherweight Java," in *Proceedings of the 8th International Conference on Integrated Formal Methods (iFM 2010)*, ser. Lecture Notes in Computer Science, D. Méry and S. Merz, Eds. Springer-Verlag, Oct. 2010, accepted for publication. An earlier an longer version has appeared as UiO, Dept. of Comp. Science Technical Report 392, Oct. 2009.

[3] G. Weikum and G. Vossen, *Fundamentals of Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2001.

[4] J. Gray and A. Reuter, *Transaction Processing. Concepts and Techniques*. Morgan Kaufmann, 1993.

[5] C. Blundell, E. C. Lewis, and M. K. Martin, “Subtleties of transactional memory atomicity semantics,” *IEEE Computer Architecture Letters*, vol. 5, no. 2, 2006.

[6] E. B. Johnsen, O. Owe, and I. C. Yu, “Creol: A type-safe object-oriented model for distributed concurrent systems,” *Theoretical Computer Science*, vol. 365, no. 1–2, pp. 23–66, Nov. 2006.

[7] A. Igarashi and N. Kobayashi, “Resource usage analysis,” *ACM Transactions on Programming Languages and Systems*, vol. 27, no. 2, pp. 264–313, 2005.

[8] M. F. Ringenburg and D. Grossman, “AtomCaml: First-class atomicity via rollback,” in *ACM International Conference on Functional Programming*. ACM, 2005, pp. 92–104, in *SIGPLAN Notices*.

[9] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, “X10: An object-oriented approach to non-uniform cluster computing,” in *Twentieth Object Oriented Programming: Systems, Languages, and Applications (OOPSLA) ’05*. ACM, 2005, pp. 519–538, in *SIGPLAN Notices*.

[10] E. Allen, D. Chase, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt, “The Fortress language specification,” Sun Microsystems, 2005.

[11] Cray, “Chapel specification,” Feb. 2005.

[12] A. Welc, A. L. Hosking, and S. Jagannathan, “Transparently reconciling transactions with locking for Java synchronization,” in *European Conference on Object-Oriented Programming (ECOOP 2006)*, ser. Lecture Notes in Computer Science, D. Thomas, Ed., vol. 4067. Springer-Verlag, 2006, pp. 148–173.

[13] M. Flatt, S. Krishnamurthi, and M. Felleisen, “Classes and mixins,” in *Proceedings of POPL ’98*. ACM, 1998, pp. 171–183.

[14] M. Abadi, A. Birell, T. Harris, and M. Isard, “Semantics of transactional memory and automatic mutual exclusion,” in *Proceedings of POPL ’08*. ACM, Jan. 2008.

[15] M. Isard and A. Birell, “Automatic mutual exclusion,” in *Proceedings of the 11th Workshop on Hot Topics in Operating Systems*, 2007.

[16] T. Elmas, S. Qadeer, and S. Tasiran, “A calculus of atomic actions,” in *Proceedings of POPL ’09*. ACM, Jan. 2009.

[17] B. D. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. C. Minh, C. Kozyrakis, and K. Oluktun, “The ATOMOΣ transactional programming language,” in *ACM Conference on Programming Language Design and Implementation (Ottawa, Ontario, Canada)*. ACM, Jun. 2006.

[18] T. Harris and K. Fraser, “Language support for lightweight transactions,” in *Eighteenth Object Oriented Programming: Systems, Languages, and Applications (OOPSLA) ’03*. ACM, 2003, in *SIGPLAN Notices*.

[19] A. Welc, S. Jagannathan, and A. Hosking, “Transactional monitors for concurrent objects,” in *18th European Conference on Object-Oriented Programming (ECOOP 2004)*, ser. Lecture Notes in Computer Science, M. Odersky, Ed., vol. 3086. Springer-Verlag, 2004, pp. 519–542.

[20] P. Helland, “Transaction monitoring facility,” *Database Engineering*, vol. 8, no. 1, pp. 9–18, Jun. 1988.

[21] J. L. Eppinger, L. B. Mummert, and A. Z. Spector, *Camelot and Avalon: A Distributed Transaction Facility*. Morgan Kaufmann, 1991.

[22] B. Liskov, D. Curtis, P. Johnson, and R. Scheifler, “The implementation of Argus,” in *Proceedings of SOSP’87: Symposium on Operating Systems Principles*, 1987, pp. 111–122.

[23] R. Hickey, “The Clojure language home page,” 2010. [Online]. Available: http://clojure.org

[24] C. Boyapati, R. Lee, and M. Rinard, “A type system for preventing data races and deadlocks in Java programs,” in *In Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, 2002, pp. 211–230.

[25] M. Abadi, C. Flanagan, and S. N. Freund, “Types for safe locking: Static race detection for Java,” *ACM Trans. Program. Lang. Syst.*, vol. 28, no. 2, pp. 207–255, 2006.

[26] T. Harris, S. M. S. Peyton Jones, and M. Herlihy, “Composable memory transactions,” in *PPoPP’05: 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Jun. 2005, pp. 48–60.

[27] C. Flanagan and S. Quadeer, “A type and effect system for atomicity,” in *ACM Conference on Programming Language Design and Implementation (San Diego, California)*. ACM, Jun. 2003.

[28] C. Flanagan and S. Freund, “Atomizer: A dynamic atomicity checker for multithreaded programs,” in *Proceedings of POPL ’04*. ACM, Jan. 2004, pp. 256–267.

[29] C. Haack, M. Huisman, and C. Hurlin, “Reasoning about Java’s reentrant locks,” in *APLAS 2008*, ser. Lecture Notes in Computer Science, G. Ramalingam, Ed., vol. 5356. Springer-Verlag, 2008, pp. 171–187.

[30] N. Kobayashi, K. Suenaga, and L. Wischik, “Resource usage analysis for the π-calculus,” in *Proceedings of VMCAI 2006*, ser. Lecture Notes in Computer Science, E. A. Emerson and K. S. Namjoshi, Eds., vol. 3855. Springer-Verlag, 2006, pp. 298–312.

[31] G. Bigliardi and C. Laneve, “A type system for JVM threads,” in *In Proceedings of 3rd ACM SIGPLAN Workshop on Types in Compilation (TIC2000*, 2000, p. 2003.

[32] C. Laneve, “A type system for JVM threads,” *Theoretical Computer Science*, vol. 290, no. 1, pp. 741 – 778, 2003.