Safe Commits for Transactional Featherweight Java*

Thi Mai Thuong Tran and Martin Steffen
Department of Informatics, University of Oslo, Norway

Abstract. Transactions are a high-level alternative for low-level concurrency-control mechanisms such as locks, semaphores, monitors. A recent proposal for integrating transactional features into programming languages is *Transactional Featherweight Java* (TFJ), extending Featherweight Java by adding transactions. With support for *nested* and *multi-threaded* transactions, its transactional model is rather expressive. In particular, the constructs governing transactions —to start and to commit a transaction— can be used freely with a *non-lexical* scope. On the downside, this flexibility also allows for an incorrect use of these constructs, e.g., trying to perform a commit outside any transaction. To catch those kinds of errors, we introduce a static type and effect system for the safe use of transactions for TFJ. We prove the soundness of our type system by subject reduction.

1 Introduction

With CPU speeds and memory capacities ever increasing, and especially with the advent of multiprocessor and multi-core architectures, effective parallel programming models and suitable language support are in need to take full advantage of the architectural advances. Transactions, a well-known and successful concept originating from database systems, have recently been proposed to be directly integrated into *programming languages*. As known from databases, transactions offer valuable safety and failure guarantees: atomicity, consistency, isolation, and durability, or ACID for short. Atomicity means that the code inside a transaction is executed completely or not at all, consistency that all transactions have the same "view" on shared data, isolation says that when a transaction is running, other transactions cannot interfere, and durability states successfully committed changes are persistent. One characteristic difference of transactions compared to locks is a non-blocking behavior. All threads/transactions may run in parallel provided that they guarantee the mentioned ACID properties. As a result, transactional programming languages may make better use of parallelism and resources in concurrent systems, and may avoid also deadlock situations.

As mechanism for concurrency control, they can be seen as a high-level, more abstract, and more compositional alternative to more conventional means for concurrency control, such as locks, semaphores, monitors, etc. How to syntactically capture transactional programming in the language may vary. One option is lexical scoping, e.g., using an *atomic* keyword, similar to the *synchronized* keyword in Java for lock-handling. More flexible is non-lexical scoping, where transactions can be started and finished (i.e., committed) freely. One proposal supporting non-lexical scoping of transaction handling

* The work has been partly supported by the EU-project FP7-231620 HATS (Highly Adaptable and Trustworthy Software using Formal Methods).

is *Transactional Featherweight Java* (TFJ) [15]. In the free use of the transactional constructs, it resembles also the way Java 5.0 allows for lock handling (using the lock and unlock methods via the `Lock`-interface). The start of a transaction in TFJ programs is marked by the onacid keyword and the end by the commit keyword. The transactional model of TFJ is quite general and 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. To commit an entire transaction, all child transaction must have committed and the child threads and the thread itself must commit at the same time. The flexibility of non-lexical use of onacid and commit comes at a cost: not all usages of starting and committing transactions "make sense". In particular, it is an error to perform a commit without being inside a transaction; we call such an error a *commit error*. In this paper, we introduce a static type and effect system to prevent these errors by keeping track of starting and committing transactions. The static analysis is formulated as a type and effect system [18]. We concentrate on the effect part, as the part dealing with the ordinary types works in a standard manner and is straightforward. See [20] for details.

The paper is organized as follows. After Section 2, which recapitulates the syntax and the operational semantics of the calculus, Section 3 defines the effect system to prevent *commit errors*. The soundness of the type system relative to the given semantics is shown in Section 4. Section 5 concludes with related and future work. In particular we draw some parallel to the lock handling in Java 5.

2 An object-oriented calculus with transactions

Next we present the syntax and semantics of TFJ. It is, with some adaptations, taken from [15] and a variant of Featherweight Java (FJ) [13] extended with *transactions* and a construct for thread creation. The main adaptations are: we added standard constructs such as sequential composition (in the form of the let-construct) and conditionals. Besides that, we did not use evaluation-context based rules for the operational semantics.

2.1 Syntax

FJ is a core language originally introduced to study typing issues related to Java, such as inheritance, subtype polymorphism, type casts. A number of extensions have been developed for other language features, so FJ is today a generic name for Java-related core calculi. Following [15] we include imperative features such as destructive field updates, further concurrency and support for transactions. Table 1 shows the abstract syntax of TFJ. A program consists of a number of processes/threads $t\langle e\rangle$ running in parallel, where t is the thread's identifier and e is the expression being executed.. The syntactic category L captures class definitions. In absence of inheritance, a class $\mathsf{class}\ C\{\vec{f}:\vec{T};K;\vec{M}\}$ consists of a name C, a list of fields $\vec{f}$ with corresponding type declarations $\vec{T}$ (assuming that all f_i's are different), a constructor K, and a list $\vec{M}$ of method definitions. A constructor $C(\vec{f}{:}\vec{T})\{\mathsf{this}.\vec{f} := \vec{f}\}$ of the corresponding class C initializes the fields of

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

Table 1. Abstract syntax

instances of that class, these fields are mentioned as the formal parameters of the constructor. We assume that each class has exactly one constructor; i.e., we do not allow constructor overloading. Similarly, we do not allow method overloading by assuming that all methods defined in a class have a different name; likewise for fields. A method definition $m(\vec{x}{:}\vec{T})\{e\} : T$ consists of the name m of the method, the formal parameters $\vec{x}$ with their types $\vec{T}$, the method body e, and finally the return type T of the method.

In the syntax, v stands for values, i.e., expressions that can no longer be evaluated. In the core calculus, we leave unspecified standard values like booleans, integers, ..., so values can be object references r, variables x or null. The expressions $v.f$ and $v_1.f := v_2$ represent field access and field update respectively. Method calls are written $v.m(\vec{v})$ and object instantiation is new $C(\vec{v})$. The next two expressions deal with the basic, sequential control structures: if v then e_1 else e_2 represents conditions, and the let-construct let $x : T = e_1$ in e_2 represents sequential composition: first e_1 is evaluated, and afterwards e_2, where the eventual value of e_1 is bound to the local variable x. Consequently, standard sequential composition $e_1; e_2$ is syntactic sugar for let $x : T = e_1$ in e_2 where the variable x does not occur free in e_2. The language is multi-threaded: spawn e starts a new thread of activity which evaluates e in parallel with the spawning thread. Specific for TFJ are the two constructs onacid and commit, two dual operations dealing with transactions. The expression onacid starts a new transaction and executing commit successfully terminates a transaction. The syntax is restricted concerning where to use general expressions e. E.g., Table 1 does not allow field updates $e_1.f := e_2$, where the object whose field is being updated and the value used in the right-hand side are represented by general expressions. It would be straightforward to relax the abstract syntax that way. We have chosen this presentation, as it slightly simplifies the operational semantics and the (presentation of the) type and effect system later. Of course, this is not a real restriction in expressivity.

2.2 Semantics

This section describes the operational semantics of TFJ with some adaptations at two different levels: a local and a global semantics. The local semantics is given in Table 2. These local rules deal with the evaluation of *one* single *expression/thread* and reduce configurations of the form $E \vdash e$. Thus, local transitions are of the form $E \vdash e \rightarrow E' \vdash e'$,

where e is one expression and E a *local environment.* At the local level, the relevant commands only concern the current thread.

Definition 1. *A* local environment *E of type LEnv is a finite sequence of the form $l_1{:}\rho_1, \ldots l_k{:}\rho_k$, i.e., of pairs of transaction labels l_i and a corresponding* log *ρ_i. We write $|E|$ for the size of the local environment (number of pairs $l{:}\rho$ in the local environment).*

Transactions are identified by labels l, and as transactions can be nested, a thread can execute "inside" a number of transactions. So, the E in the above definition is ordered, with e.g. l_k to the right refers to the inner-most transaction, i.e., the one most recently started and commiting removes bindings from right to left. The number $|E|$ of a thread represents the nesting depth of the thread, i.e., how many transactions the thread has started but not yet committed. The corresponding *logs* ρ_i can, in a first approximation, be thought of as "local copies" of the heap including bindings from references to objects. The *log* ρ_i keeps track of changes of the threads actions concerning transaction l_i. The exact structure of such environments and the logs have no influence on our static analysis, and indeed, the environments may be realized in different ways (e.g., [15] gives two different flavors, a "pessimistic", lock-based one and an "optimistic" one). Relevant for our type and effect system will be only a number of *abstract properties* of the environments, formulated in Definition 3 later. As the local rules in Table 2 are pretty standard, and correspond to the ones of [15]. The first four rules deal straightforwardly with the basic, sequential control flow. Unlike the first four rules, the remaining ones do access the heap. Thus, the local environment E is consulted to look up object references and then *changed* in the step. The access and update of E is given abstractly by corresponding access functions *read*, *write*, and *extend* (which look-up a reference on the heap, update a reference, resp. allocate an entry for a new reference on the heap). The details can be found in [15] but note that also the *read*-function used in the rules actually *changes* the environment from E to E' in the step. The reason is that in a transaction-based implementation, read-access to a variable may be *logged*, i.e., remembered appropriately, to be able to detect conflicts and to do a roll-back if the transaction fails. This logging may change the local environment. The premises assume the class table is given implicitly where $\mathit{fields}(C)$ looks up fields of class C and $\mathit{mbody}(m,C)$ looks up the method m of class C. Otherwise, the rules for field lookup, field update, method calls, and object instantiation are standard.

The five rules of the *global* semantics are given in Table 3. The semantics works on configurations of the form $\Gamma \vdash P$, where P is a *program* and Γ is a global environment. Besides that, we need a special configuration *error* representing an error state. Basically, a program P consists of a number of threads evaluated in parallel (cf. Table 1), where each thread corresponds to one expression, whose evaluation is described by the local rules. Now that we describe the behavior of a number of (labeled) threads $t\langle e\rangle$, we need one E for each thread t. This means, Γ is a "sequence" (or rather a set) of $t{:}E$ bindings where t is the name of a thread and E is its corresponding local environment.

Definition 2. *A* global environment *Γ of type GEnv is a finite mapping, written as $t_1{:}E_1, \ldots t_k{:}E_k$, from threads names t_i to local environments E_i (the order of bindings does not play a role, and each thread name can occur at most once).*

So global steps are of the form $\Gamma \vdash P \Longrightarrow \Gamma' \vdash P'$ or $\Gamma \vdash P \Longrightarrow \mathit{error}$. As for the local rules, the formulation of the global steps makes use of a number of functions accessing and changing the (this time global) environment. As before, those functions are left abstract and only later we will formalize abstract properties that Γ and E considered as abstract data types must satisfy. Rule G-PLAIN simply *lifts* a local step to the global level, using the reflect-operation, which roughly makes local updates of a thread globally visible. Rule G-SPAWN deals with starting a thread. The next three rules treat the two central commands of the calculus, those dealing directly with the transactions. The first one G-TRANS covers onacid, which starts a transaction. The *start* function creates a new label l in the local environment E of thread t. The two rules G-COMM and G-COMM-ERROR formalize the successful commit resp. the failed attempt to commit a transaction. In G-COMM, the label of the transaction l to be committed is found (right-most) in the local context E. Furthermore, the function $\mathit{intranse}(l,\Gamma)$ finds the identities $t_1 \ldots t_k$ of all concurrent threads in the transaction l and which all join in the commit. In the erroneous case of G-COMM-ERROR, the local environment E is empty; i.e., the thread executes outside of any transactions, which constitutes an error.

The next section continues with the effect system, as part of a "type and effect" system. The underlying types T include names C of classes, basic types B (natural numbers, booleans, etc.) and Void for typing side-effect-only expressions. The corresponding type system for judgements of the form $\Gamma \vdash e : T$ ("under type assumptions Γ, expression e has type T") is standard and omitted here (cf. the technical report [20]).

3 The effect system

The effect system assures that starting and committing transactions is done "properly", in particular to avoid committing when outside a transaction, which we call *commit errors*. To catch commit errors at compile time, the system keeps track of onacids and

---

$$E \vdash \mathsf{let}\, x : T = v \;\mathsf{in}\; e \to E \vdash e[v/x] \qquad \text{R-RED}$$

$$E \vdash \mathsf{let}\, x_2 : T_2 = (\mathsf{let}\, x_1 : T_1 = e_1 \;\mathsf{in}\; e) \;\mathsf{in}\; e' \to E \vdash \mathsf{let}\, x_1 : T_1 = e_1 \;\mathsf{in}\; (\mathsf{let}\, x_2 : T_2 = e \;\mathsf{in}\; e') \qquad \text{R-LET}$$

$$E \vdash \mathsf{let}\, x : T = (\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2) \;\mathsf{in}\; e \to E \vdash \mathsf{let}\, x : T = e_1 \;\mathsf{in}\; e \qquad \text{R-COND}_1$$

$$E \vdash \mathsf{let}\, x : T = (\mathsf{if}\ \mathsf{false}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2) \;\mathsf{in}\; e \to E \vdash \mathsf{let}\, x : T = e_2 \;\mathsf{in}\; e \qquad \text{R-COND}_2$$

$$\frac{\mathit{read}(r,E) = E', C(\vec{u}) \quad \mathit{fields}(C) = \vec{f}}{E \vdash \mathsf{let}\, x{:}T = r.f_i \;\mathsf{in}\; e \to E' \vdash \mathsf{let}\, x{:}T = u_i \;\mathsf{in}\; e} \text{R-LOOKUP} \qquad \frac{\mathit{read}(r,E) = E', C(\vec{r}) \quad \mathit{write}(r \mapsto C(\vec{r}) \downarrow_i^{r'}, E') = E''}{E \vdash \mathsf{let}\, x{:}T = r.f_i := r' \;\mathsf{in}\; e \to E'' \vdash \mathsf{let}\, x{:}T = r' \;\mathsf{in}\; e} \text{R-UPD}$$

$$\frac{\mathit{read}(r,E) = E', C(\vec{r}) \quad \mathit{mbody}(m,C) = (\vec{x}, e)}{E \vdash \mathsf{let}\, x{:}T = r.m(\vec{r}) \;\mathsf{in}\; e' \to E' \vdash \mathsf{let}\, x : T = e[\vec{r}/\vec{x}][r/\mathsf{this}] \;\mathsf{in}\; e'} \text{R-CALL}$$

$$\frac{r \;\mathit{fresh} \quad E' = \mathit{extend}(r \mapsto C(\vec{\mathsf{null}}), E)}{E \vdash \mathsf{let}\, x{:}T = \mathsf{new} C() \;\mathsf{in}\; e \to E' \vdash \mathsf{let}\, x = r \;\mathsf{in}\; e} \text{R-NEW}$$

Table 2. Semantics (local)

$$\frac{E \vdash e \rightarrow E' \vdash e' \qquad \Gamma \vdash t : E \qquad \mathit{reflect}(t, E', \Gamma) = \Gamma'}{\Gamma \vdash P \parallel t\langle e \rangle \Longrightarrow \Gamma' \vdash P \parallel t\langle e' \rangle} \text{ G-PLAIN}$$

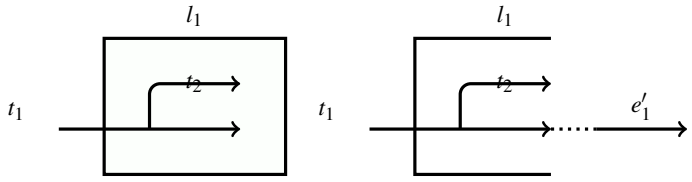
$$\frac{t' \textit{ fresh} \qquad \mathit{spawn}(t, t', \Gamma) = \Gamma'}{\Gamma \vdash P \parallel t\langle \mathsf{let}\, x : T = \mathsf{spawn}\, e_1 \;\mathsf{in}\; e_2 \rangle \Longrightarrow \Gamma' \vdash P \parallel t\langle \mathsf{let}\, x : T = \mathsf{null}\;\mathsf{in}\; e_2 \rangle \parallel t'\langle e_1 \rangle} \text{ G-SPAWN}$$

$$\frac{l \textit{ fresh} \qquad \mathit{start}(l, t, \Gamma) = \Gamma'}{\Gamma \vdash P \parallel t\langle \mathsf{let}\, x : T = \mathsf{onacid}\;\mathsf{in}\; e \rangle \Longrightarrow \Gamma' \vdash P \parallel t\langle \mathsf{let}\, x : T = \mathsf{null}\;\mathsf{in}\; e \rangle} \text{ G-TRANS}$$

$$\frac{\Gamma = \Gamma'', t{:}E \qquad E = E', l{:}\rho \qquad \mathit{intranse}(l, \Gamma) = \vec{t} = t_1 \ldots t_k \qquad \mathit{commit}(\vec{t}, \vec{E}, \Gamma) = \Gamma' \quad t_1{:}E_1, t_2{:}E_2, \ldots t_k{:}E_k \in \Gamma \quad \vec{E} = E_1, E_2, \ldots, E_k}{\Gamma \vdash P \parallel \ldots \parallel t_i\langle \mathsf{let}\, x : T_i = \mathsf{commit}\;\mathsf{in}\; e_i \rangle \parallel \ldots \Longrightarrow \Gamma' \vdash P \parallel \ldots \parallel t_i\langle \mathsf{let}\, x : T_i = \mathsf{null}\;\mathsf{in}\; e_i \rangle \parallel \ldots} \text{ G-COMM}$$

$$\frac{\Gamma = \Gamma'', t{:}E \qquad E = \emptyset}{\Gamma \vdash P \parallel t\langle \mathsf{let}\, x : T = \mathsf{commit}\;\mathsf{in}\; e \rangle \Longrightarrow \mathit{error}} \text{ G-COMM-ERROR}$$

Table 3. Semantics (global)



(a) Multi-threaded transaction (b) Sequential composition

Fig. 1. Transactions and multi-threading

commits; we refer to the number of onacids minus the number of commits as the *balance*. E.g., for an expression $e = \mathsf{onacid};\ e_1;\ \mathsf{commit};\ \mathsf{commit}$, the *balance* equals $1-2=-1$. An execution of a thread is *balanced,* if there are no pending transactions, i.e., if the balance is 0. The situation gets slightly more involved when dealing with multi-threading. TFJ supports not only nested transactions, but *multi-threaded* transactions: inside one transaction there may be more than one thread active at a time. Due to this internal concurrency, the effect of a transaction may be non-deterministic. Figure 1 shows a simple situation with two threads t_1 and t_2, where t_1 starts a transaction with the label l_1 and spawns a new thread t_2 inside the transaction. An example expression resulting in the depicted behavior of Figure 1(b) is $e_1 = \mathsf{onacid}; \mathsf{spawn}\, e_2; e'_1$, where e_1 is the expression evaluated by thread t_1, and e_2 by the freshly created t_2. In TFJ's concurrency model, to terminate the parent transaction l_1, both t_1 and t_2 must *join* via a common commit. To keep track we must take into account that e_2 and the rest e'_1 of the original thread are executed in parallel, and furthermore, that when executing e_2 in the new thread t_2, one onacid has already been executed by t_1, namely before the spawn-operation. Hence, we need to keep track of the balance not just for the thread under consideration, but take into account the balance of the newly created threads, as well. Even if a spawning thread and a spawned thread run in parallel, the situation wrt.

the analysis is not symmetric. Considering the balance for the left of $\mathsf{onacid};\mathsf{spawn}\,e_2$, the balance for both "threads" after execution amounts to $+1$, i.e., both threads are executing inside one enclosing transaction. When calculating the combined effect for $\mathsf{onacid};\mathsf{spawn}\,e_2;e_1'$, the balance value of onacid is treated differently from the one of e_2, as the control flow of the sequential composition connects the trailing e_1' with onacid, but not with the thread of e_2

To sum up: to determine the effect in terms of the balance, we need to calculate the balance for *all* threads potentially concerned, which means for the thread executing the expression being analysed plus all threads (potentially) spawned during that execution. From all threads, the one which carries the expression being evaluated plays a special role, and is treated specially. Therefore, we choose a pair of an integer n and a (finite) multi-set S of integers to represent the effect after evaluating an expression as follows:

$$n, S : \mathsf{Int} \times (\mathsf{Int} \to \mathsf{Nat}) \; . \tag{1}$$

The integer n represents the balance of the thread of the given expression, the multi-set the balance numbers for the threads potentially spawned by the expression. We write $\emptyset$ for the empty multi-set, $\cup$ for the multi-set union. The multi-set can be seen as a function of type $\mathsf{Int} \to \mathsf{Nat}$ (the multi-set's characteristic function), and we write $dom(S)$ for the set of elements of S, ignoring their multiplicity. As an example: we use also the set-like notation $\{-3,1,1,2\}$ to represent the finite mapping $-3 \mapsto 1, 1 \mapsto 2, 2 \mapsto 1$ (and all other integers to 0). As a further operation, we use "addition" and "substraction" of such multisets and integers illustrated on a small example: $\{-3,1,1,2\}+5$ gives $\{2,6,6,7\}$. Based on S, we know how many newly created threads with their corresponding balances in the current expression, including threads with the same balance. The judgements of the analysis are thus of the following form:

$$n_1 \vdash e :: n_2, S \; , \tag{2}$$

which reads as: starting with a balance of n_1, executing e results in a balance of n_2 and the balances for new threads spawned by e are captured by S. The balance for the new threads in S is calculated *cumulatively*; i.e., their balance includes n_1, the contribution of e before the thread is spawned, plus the contribution of the new thread itself.

The effect system is given in Table 4. For clarity, we do not integrate the effect system with the underlying type system. Instead, we concentrate on the effects in isolation. Variables, the null-expression, field lookup, and object creation have no effect (cf. T-VAR, T-NULL, T-LOOKUP, and T-NEW in Table 4). A field update has no effect (cf. T-UPD), as we require that the left- and the right-hand side of the assignment are already evaluated. In contrast, the two dual commands of onacid and commit have the expected effect: they simply increase, resp. decrease the balance by one (cf. T-ONACID and T-COMMIT). A class declaration (cf. T-CLASS) has no effect and no newly created threads, therefore the balance is zero and the multiset of balances equals $\emptyset$. Rule T-METH deals with method declarations. 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 $n_1 \to n_2$ 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

$$\frac{}{n \vdash x :: n,\emptyset}\ \text{T-VAR} \qquad \frac{}{n \vdash \mathsf{null} :: n,\emptyset}\ \text{T-NULL} \qquad \frac{}{n \vdash v.f :: n,\emptyset}\ \text{T-LOOKUP} \qquad \frac{}{n \vdash \mathsf{new}\, C :: n,\emptyset}\ \text{T-NEW}$$

$$\frac{n \vdash v_1 :: n,\emptyset \qquad n \vdash v_2 :: n,\emptyset}{n \vdash v_1.f_i := v_2 :: n,\emptyset}\ \text{T-UPD} \qquad \frac{}{n \vdash \mathsf{onacid} :: n+1,\emptyset}\ \text{T-ONACID} \qquad \frac{n \geq 1}{n \vdash \mathsf{commit} :: n-1,\emptyset}\ \text{T-COMMIT}$$

$$\frac{K = C(\vec{f} : \vec{T})\{\mathsf{this}.\vec{f} := \vec{f}\} \qquad \vdash \vec{M} :: \vec{n}_1 \to \vec{n}_2, \vec{S}}{\vdash \mathsf{class}\ C\{\vec{f} : \vec{T}; K; \vec{M}\} :: 0,\emptyset}\ \text{T-CLASS} \qquad \frac{n_1 \vdash e :: n_2, \{0,\ldots\}}{\vdash m(\vec{x} : \vec{T})\{e\} :: n_1 \to n_2, \{0,\ldots\}}\ \text{T-METH}$$

$$\frac{n \vdash e :: n', S_1 \qquad S_1 \leq S_2}{n \vdash e :: n', S_2}\ \text{T-SUB} \qquad \frac{n_0 \vdash e_1 :: n_1, S_1 \qquad n_1 \vdash e_2 :: n_2, S_2}{n_0 \vdash \mathsf{let}\ x : T = e_1\ \mathsf{in}\ e_2 :: n_2, S_1 \cup S_2}\ \text{T-LET}$$

$$\frac{n \vdash e :: n', S}{n \vdash \mathsf{spawn}\, e :: n, S \cup \{n'\}}\ \text{T-SPAWN} \qquad \frac{n \vdash v :: n,\emptyset \qquad n \vdash e_1 :: n', S_1 \qquad n \vdash e_2 :: n', S_2}{n \vdash \mathsf{if}\ v\ \mathsf{then}\, e_1\ \mathsf{else}\ e_2 :: n', S_1 \cup S_2}\ \text{T-COND}$$

$$\frac{n \vdash v :: n,\emptyset \qquad n \vdash v_i :: n,\emptyset \qquad \mathit{mtype}(C,m) :: n'_1 \to n'_2, S \qquad n = n'_1}{n \vdash v.m(\vec{v}) :: n'_2 - n'_1 + n, S - n'_1 + n}\ \text{T-CALL}_1$$

$$\frac{n \vdash v :: n,\emptyset \qquad n \vdash v_i :: n,\emptyset \qquad \mathit{mtype}(C,m) :: n'_1 \to n'_2,\emptyset \qquad n > n'_1}{n \vdash v.m(\vec{v}) :: n'_2 - n'_1 + n,\emptyset}\ \text{T-CALL}_2$$

$$\frac{|E| \vdash e :: 0, \{0,0,\ldots\}}{t{:}E \vdash t\langle e \rangle : ok}\ \text{T-THREAD} \qquad \frac{\Gamma_1 \vdash P_1 : ok \qquad \Gamma_2 \vdash P_2 : ok}{\Gamma_1, \Gamma_2 \vdash P_1 \parallel P_2 : ok}\ \text{T-PAR}$$

Table 4. Effect system

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 (see also the T-CALL-rules below). Rule T-SUB captures a notion of *subsumption* where by $S_1 \leq S_2$ we mean the subset relation on multi-sets.[1] In a let-expression (cf. T-LET), representing sequential composition, the effects are accumulated. Creating a new thread by executing $\mathsf{spawn}\, e$ does not change the balance of the executing thread (cf. T-SPAWN). The spawned expression e in the new thread is analyzed starting with the same balance n in its pre-state. The resulting balance n' of the new thread is given back in the conclusion as part of the balances of the spawned threads, i.e., as part of the multi-set. For conditionals $\mathsf{if}\, v\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2$ (cf. T-COND), the boolean condition v does not change the balance, and the rule insists that the two branches e_1 and e_2 agree on a balance n'.

For method calls, we distinguish two situations (cf. T-CALL$_1$ and T-CALL$_2$), depending on whether the method being called creates new threads or not. In the latter case, the multi-set of balances for method m in class C is required to be empty by the third premise of the rule. In that situation, the precondition of the method can be interpreted in a "loose" manner: the current balance n in the state before the call must be *at least* as big as the pre-condition n'_1. If, however, the method may spawn a new

[1] The non-structural rule of subsumption makes the system non syntax-directed. To turn it to an algorithm, one would have to disallow subsumption and derive a minimal multiset instead.

thread (cf. T-CALL$_1$), the pre-condition is interpreted *strictly*, i.e., we require $n = n'_1$ (with this equality, T-CALL$_1$ could be simplified; we chose this representation to stress the connection with T-CALL$_2$, where $n > n'_1$). Allowing the loose interpretation also in that situation would make the method callable in different levels of nestings at the caller side; however, only exactly *one* level actually is appropriate, as with concurrent threads inside a transaction, all threads must join in a commit to terminate the transaction. A thread $t\langle e\rangle$ is well-typed (cf. T-THREAD), if the expression has balance 0 after termination, starting with a balance corresponding to the length $|E|$ of the local environment E. We use *ok* to indicate that the thread is well-typed, i.e., without commit-error. This balance in the pre-state corresponds to the level of nesting inside transactions, the thread $t\langle e\rangle$ currently executes in. A program is well typed, if all threads in the system are well-typed (cf. T-PAR). We illustrate the system with the following two examples:

Example 1. The following derivation applies the effect system to the expression e_1; $\mathsf{spawn}(e_2;\mathsf{spawn}\,e_3)$; $e_4 :: n_4, \{n_2, n_3\}$, when starting with a balance of 0.

$$\dfrac{\dfrac{}{0 \vdash e_1 :: n'_1, \{\}} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{n_2 \vdash e_3 :: n_3, \{\}}}{n'_1 \vdash e_2 :: n_2, \{\} \quad n_2 \vdash \mathsf{spawn}\,e_3 :: n_2, \{n_3\}}}{n'_1 \vdash (e_2;\mathsf{spawn}\,e_3) :: n_2, \{n_3\}}}{n'_1 \vdash \mathsf{spawn}(e_2;\mathsf{spawn}\,e_3) :: n'_1, \{n_2, n_3\} \quad n'_1 \vdash e_4 :: n_4, \{\}}}{n'_1 \vdash \mathsf{spawn}(e_2;\mathsf{spawn}\,e_3); e_4 :: n_4, \{n_2, n_3\}}}{0 \vdash e_1; \mathsf{spawn}(e_2;\mathsf{spawn}\,e_3); e_4 :: n_4, \{n_2, n_3\}}$$

The derivation demonstrates sequential composition and thread creation with a starting balance of 0 for simplicity. Remember that sequential composition $e_1;e_2$ is syntactic sugar for $\mathsf{let}\,x{:}T = e_1\ \mathsf{in}\ e_2$, where x does not occur free in e_2; i.e., assume that the expressions $e_1,\ldots e_4$ themselves have the following balances $0 \vdash e_i :: n'_i, \{\}$, which implies $n'_1 \vdash e_2 :: n'_1 + n'_2 = n_2, \{\}$, $n_2 \vdash e_3 :: n_2 + n'_3 = n_3, \{\}$, and $n'_1 \vdash e_4 :: n'_1 + n'_4 = n_4, \{\}$. □

Example 2. Assume the following code fragment:

```
...
void n(){ onacid; m(10); }

void m(i){     commit;
               if    (i ≤ 0)
               then  onacid;
               else  ...; onacid; this.m(i−1); }
void main(){ n();  commit; }
```

First observe that the program shows no commit-errors during run-time. Method *m* calls itself recursively and the two branches of the conditional in its body both execute one onacid each. Especially, method *m* is called (in this fragment) only via method *n*, especially after *n* has performed an onacid, i.e., *m* is called *inside* one transaction. If *m* were called *outside* a transaction it would result in an error, as the body of *m* starts by executing a commit-statement. In our effect system, method *m* can be declared as of effect $1 \to 1$, which expresses not only that the body of *m* does not change the balance, but that as a precondition, it must be called *only* from call-sites where the balance is ≥ 1, as is the case in the body of *n* (cf. also T-METH and T-CALL). So the

declarations of the two shown methods are of the form $n() : \mathsf{Void} \to \mathsf{Void}, 0 \to 1$ and $m(i) : \mathsf{Int} \to \mathsf{Void}, 1 \to 1$. For recursive calls, an effect like $1 \to 1$ can be interpreted as *loop invariant*: the body of the method must not change the balance to be well-typed. However, not every method needs to be balanced; the non-recursive method n is one example which (together with the call to m) has a net-balance of 1. □

4 Soundness of the type and effect system

Next we prove that the type and effect system does what it is designed to do, namely absence of commit errors.

Lemma 1 (Subject reduction (local)). *Let $n = |E|$. If $n \vdash e :: n', S'$ and $E \vdash e \to E' \vdash e'$, then $|E'| = n$ and $n \vdash e' :: n', S'$.*

Proof. By straightforward induction on the rules of Table 2, observing that by the properties of *read*, *write*, and *extend*, $|E| = |E'|$. □

The global semantics accesses and changes the global environments Γ. These manipulations are captured in various functions, which are kept "abstract" in this semantics (as in [15]). To perform the subject reduction proof, however, we need to impose certain requirements on those functions:

Definition 3. *The properties of the abstract functions are specified as follows:*

1. *The function reflect satisfies the following condition: if $reflect(t,E,\Gamma) = \Gamma'$ and $\Gamma = t_1{:}E_1, \ldots, t_k{:}E_k$, then $\Gamma' = t_1{:}E'_1, \ldots, t_k{:}E'_k$ with $|E_i| = |E'_i|$ (for all i).*
2. *The function spawn satisfies the following condition: Assume $\Gamma = t : E, \Gamma''$ and $t' \notin \Gamma$ and $spawn(t,t',\Gamma) = \Gamma'$, then $\Gamma' = \Gamma, t'{:}E'$ s.t. $|E| = |E'|$.*
3. *The function start satisfies the following condition: if $start(l,t_i,\Gamma) = \Gamma'$ for a $\Gamma = t_1{:}E_1, \ldots, t_i{:}E_i, \ldots, t_k{:}E_k$ and for a fresh l, then $\Gamma' = t_1{:}E_1, \ldots, t_i{:}E'_i, \ldots, t_k{:}E_k$, with $|E'_i| = |E_i| + 1$.*
4. *The function intranse satisfies the following condition: Assume $\Gamma = \Gamma'', t{:}E$ s.t. $E = E', l{:}\rho$ and $intranse(l,\Gamma) = \vec{t}$, then*
   (a) $t \in \vec{t}$ and
   (b) for all $t_i \in \vec{t}$ we have $\Gamma = \ldots, t_i : (E'_i, l{:}\rho_i), \ldots$.
   (c) for all threads t' with $t' \notin \vec{t}$ and where $\Gamma = \ldots, t'{:}(E', l'{:}\rho'), \ldots$, we have $l' \neq l$.
5. *The function commit satisfies the following condition: if $commit(\vec{t},\vec{E},\Gamma) = \Gamma'$ for a $\Gamma = \Gamma'', t{:}(E, l{:}\rho)$ and for a $\vec{t} = intranse(l,\Gamma)$ then $\Gamma' = \ldots, t_j{:}E'_j, \ldots, t_i{:}E'_i, \ldots$ where $t_i \in \vec{t}, t_j \notin \vec{t}$, $t_j{:}E_j \in \Gamma$, with $|E'_j| = |E_j|$ and $|E'_i| = |E_i| - 1$.*

Lemma 2 (Subject reduction). *If $\Gamma \vdash P : ok$ and $\Gamma \vdash P \Longrightarrow \Gamma' \vdash P'$, then $\Gamma' \vdash P' : ok$.*

Proof. Proceed by case analysis on the rules of the operational semantics from Table 3 (except rule G-CommError for commit errors). For simplicity (and concentrating on the effect, not the values of expressions) we use ; for sequential composition in the proof, and not the more general let-construct.

Case: G-PLAIN
From the premises of the rule, we get for the form of the program that $P = P'' \parallel t\langle e\rangle$, furthermore for t's local environment $\Gamma \vdash t : E$ and $E \vdash e \to E' \vdash e'$ as a local step. Well-typedness $\Gamma \vdash P : ok$ implies $n \vdash e :: n', S'$ for some n' and S', where $n = |E|$. By subject reduction for the local steps (Lemma 1) $n \vdash e' :: n', S'$. By the properties of the *reflect*-operation, $|E'| = n$, so we derive for the thread t

$$\frac{n \vdash e' :: 0, \{0,\ldots\}}{\Gamma', t{:}E' \vdash t\langle e'\rangle : ok}$$

from which the result $\Gamma' \vdash P'' \parallel t\langle e'\rangle : ok$ follows (using T-PAR and the properties of *reflect* from Definition 3.1).

Case: G-SPAWN
In this case, $P = P'' \parallel t\langle \mathsf{spawn}\, e_1; e_2\rangle$ and $P' = P'' \parallel t\langle \mathsf{null}; e_2\rangle \parallel t'\langle e_1\rangle$ (from the premises of G-SPAWN). The well-typedness assumption $\Gamma \vdash P : ok$ implies the following sub-derivation:

$$\dfrac{\dfrac{\dfrac{n \vdash e_1 :\ 0, S_1}{n \vdash \mathsf{spawn}\, e_1 :\ n, S_1 \cup \{0\}} \quad n \vdash e_2 :\ 0, S_2}{n \vdash \mathsf{spawn}\, e_1; e_2 :\ 0, \{0,\ldots\}}}{t{:}E \vdash t\langle \mathsf{spawn}\, e_1; e_2\rangle : ok} \tag{3}$$

with $S_1 = \{0,\ldots\}$ and $S_2 = \{0,\ldots\}$. By the properties of *reflect*, the global environment Γ' after the reduction step is of the form $\Gamma, t'{:}E'$ where t' is fresh and $|E'| = |E|$ (see Definition 3.2). So we can derive

$$\dfrac{t{:}E \vdash t\langle \mathsf{null}; e_2\rangle : ok \qquad \dfrac{n \vdash e_1 : 0, \{0,\ldots\}}{t'{:}E' \vdash t'\langle e_1\rangle : ok}}{t{:}E, t'{:}E' \vdash t\langle \mathsf{null}; e_2\rangle \parallel t'\langle e_1\rangle : ok}$$

The left sub-goal follows from T-THREAD,T-SEQ, T-NULL, and the right sub-goal of the previous derivation (3). The right open sub-goal directly corresponds to the left sub-goal of derivation (3).

Case: G-TRANS
In this case, $P = P'' \parallel t\langle \mathsf{onacid}; e\rangle$ and $P' = P'' \parallel t\langle \mathsf{null}; e\rangle$. The well-typedness assumption $\Gamma \vdash P : ok$ implies the following sub-derivation (assume that $|E| = n$):

$$\dfrac{\dfrac{n \vdash \mathsf{onacid} :: n+1, \emptyset \quad n+1 \vdash e :: 0, \{0,\ldots\}}{n \vdash \mathsf{onacid}; e :: 0, \{0,\ldots\}}}{t{:}E \vdash t\langle \mathsf{onacid}; e\rangle : ok} \tag{4}$$

For the global environment Γ' after the step, we are given $\Gamma' = \mathit{start}(l, t, \Gamma)$ from the premise of rule G-TRANS. By the properties of *start* from Definition 3.3, we have $\Gamma' = \Gamma'', t{:}E'$ with $|E'| = n+1$. So with the help of right sub-goal of the previous derivation (4), we can derive for thread t after the step:

$$\frac{n+1 \vdash e :: 0, \{0,\ldots\}}{t{:}E' \vdash t\langle e\rangle : ok}$$

Since furthermore the local environments of all other threads remain unchanged (cf. again Definition 3.3), the required $\Gamma' \vdash P' : ok$ can be derived, using T-PAR.

Case: G-COMM
In this case, $P = P'' \parallel \vec{t}\langle \mathsf{commit}; \vec{e} \rangle$ and $P' = P'' \parallel \vec{t}\langle \vec{e} \rangle$. The well-typedness assumption $\Gamma \vdash P : ok$ implies the following sub-derivation for thread t:

$$\dfrac{\dfrac{n \vdash \mathsf{commit} :: n-1, \emptyset \quad n-1 \vdash e_i : \ 0, \{0, \ldots\}}{n \vdash \mathsf{commit}; e_i : \ 0, \{0, \ldots\}}}{t_i{:}E_i \vdash t_i\langle \mathsf{commit}; e_i \rangle : ok} \tag{5}$$

For the global environment Γ' after the step, we are given $\Gamma' = commit(\vec{t}, \vec{E}, \Gamma)$ from the premise of rule G-TRANS, where $\vec{t} = intranse(l, \Gamma)$ and $\vec{E}$ are the corresponding local environments. By the properties of *commit* from Definition 3.5, we have for the local environments $\vec{E'}$ of threads $\vec{t}$ after the step that $|E'_i| = n - 1$. So we obtain by T-THREAD, using the right sub-goal of derivation (5):

$$\dfrac{n-1 \vdash e_i :: 0, \{0, \ldots\}}{t_i{:}E'_i \vdash t_i\langle e_i \rangle : ok}$$

For the threads $t_j\langle e_j \rangle$ different from $\vec{t}$, according to the Definition 3.5, we have $|E'_j| = |E_j|$ so $t_j{:}E'_j \vdash t_j\langle e'_j \rangle : ok$ straightforwardly. As a result, we have $\Gamma' \vdash P' : ok$. □

Lemma 3. *If $\Gamma \vdash P : ok$ then it is not the case that $\Gamma \vdash P \Longrightarrow error$.*

Proof. Let $\Gamma \vdash P : ok$ and assume for a contradiction that $\Gamma \vdash P \rightarrow error$. From the rules of the operational semantics it follows that $P = t\langle \mathsf{commit}; e \rangle \parallel P'$ for some thread t, where the step $\Gamma \vdash P \rightarrow error$ is done by t (executing the commit-command). Furthermore, the local environment E for the thread t is empty:

$$\dfrac{E = \emptyset}{\Gamma', t{:}E \vdash t\langle \mathsf{commit}; e \rangle \parallel P' \rightarrow error}\ \text{G-COMM}$$

To be well-typed, i.e., for the judgment $\Gamma \vdash t\langle \mathsf{commit}; e \rangle \parallel P' : ok$ to be derivable, it is easy to see that the derivation must contain $\Gamma', t{:}\emptyset \vdash t\langle \mathsf{commit}; e \rangle : n, S$ as sub-derivation (for some n and S). By inverting rule T-THREAD, we get that $0 \vdash \mathsf{let}\ \mathsf{commit}\ \mathsf{in}\ \ e : 0, \{0, 0, \ldots\}$ is derivable (since $|E| = 0$). This is a contradiction, as the balance after commit would be negative (inverting rules T-LET and T-COMMIT). □

Corollary 1 (Well-typed programs are commit-error free). *If $\Gamma \vdash P : ok$ then it is not the case that $\Gamma \vdash P \Longrightarrow^* error$,*

Proof. A direct consequence of the subject reduction Lemma 2 and Lemma 3. □

5 Conclusion

This work took the TFJ language design from [15] as starting point. That paper 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 the language may lead to run-time errors when executing a commit outside any transaction; we called such situations *commit-errors*. To statically prevent commit-errors, we presented a static type and effect system, which keeps track of the commands for starting and finishing transactions. We proved soundness of the type system.

A comparison with explicit locks of Java The built-in support for concurrency 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") concurrency control in Java offers *lexically scoped* protection based on mutual exclusion. While offering basic concurrency control, the scheme has been criticized as too rigid, and consequently, Java 5 now additionally supports explicit locks with non-lexical scope. The `ReentrantLock` class and the `Lock` interface allow more freedom, offering explicit `lock` and `unlock` operations. Locking and unlocking can be compared, to some extent, to starting and committing a transaction, even if there are differences especially wrt. failure and progress properties. See e.g., [3] for a discussion of such differences. Besides the more behavioral differences, such as different progress guarantees or deadlocking behavior, the lock handling in Java 5 and the transactional model of TFJ differ in the following aspects, as relevant for the type analysis (cf. Table 5).

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.* Both schemes, as mentioned, have all the flexibility of non-lexical scoping. The rest of Table 5 deals with the structure of protected areas (the transaction or the execution protected by a lock) and the connection to the threading model. 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 re-enter 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 section 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 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. In [17], we discuss the differences and similarities in more depth by comparing the analysis developed here with a corresponding one that deals with the safe use of a statically allocated number of locks.

Related work There have been a number of further proposals for integrating transactional features into programming languages. 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 [4], the AME calculus [1], and many proposals for software transactional memory [8, 21], but none of them deals with assuring statically proper use of the corresponding constructs. When dealing with concurrency, most static analyses 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, e.g. in [9, 1, 14]. A type system for *atomicity* is presented in [7, 6]. [2] develops 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. 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. Especially *safe lock* analysis, supported e.g. by the Indus tool [19] as part of Bandera, is a static analysis that checks whether a lock is held indefinitely (in the context of multi-threaded Java). Software model checking is a prominent, alternative way to assure quality of software. By using some form of abstraction (typically ignoring data parts and working on an abstract, automata-based representation), model checking can be used as a form of static analysis of concrete programs, as well. The Blast analyzer [11] allows automatic verification for checking temporal safety properties of C programs (using counter-example guided abstraction refinement), and has been extended to deal with concurrent programs, as well [5]. Similarly, Java PathFinder is an automatic, model-checking tool (based on Spin) to analyze Java programs [10].

Future work The work presented here can be extended to deal with more complex language features, e.g. when dealing with higher-order functions. In that setting, the effect part and its connection to the type system become more challenging. Furthermore, we plan to adopt the results for a different language design, more precisely to the language Creol [16], which is based on asynchronously communicating, active objects, in contrast to Java, whose concurrency is based on multi-threading. As discussed, there are similarities between lock-handling in Java 5 and the transactions as treated here. We plan to use similar techniques as explored here to give static guarantees for lock-based concurrency, as well. Of practical relevance is to extend the system from type *checking* to type *inference*, potentially along the lines [12].

	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 5. Transactional Featherweight Java and explicit locks of Java

Acknowledgements We thank the anonymous reviewers for their helpful suggestions.

References

1. M. Abadi, A. Birell, T. Harris, and M. Isard. Semantics of transactional memory and automatic mutual exclusion. In *Proceedings of POPL '08*. ACM, 2008.
2. G. Bigliardi and C. Laneve. A type system for JVM threads. In *In Proceedings of 3rd ACM SIGPLAN Workshop on Types in Compilation (TIC2000*, page 2003, 2000.
3. C. Blundell, E. C. Lewis, and M. K. Martin. Subtleties of transactional memory atomicity semantics. *IEEE Computer Architecture Letters*, 5(2), 2006.
4. B. D. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. C. Minh, C. Kozyrakis, and K. Olukotun. The ATOMOΣ transactional programming language. In *ACM Conference on Programming Language Design and Implementation (Ottawa, Ontario, Canada)*. ACM, 2006.
5. A. Davare. Concurrent BLAST, 2003. Internal Report, EECS Berkely. Mentors: Mentors Rupak Majumdar and Ranjit Jhala.
6. C. Flanagan and S. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Proceedings of POPL '04*, pages 256–267. ACM, 2004.
7. 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, 2003.
8. T. Harris and K. Fraser. Language support for lightweight transactions. In *Eighteenth OOPSLA '03*. ACM, 2003. In *SIGPLAN Notices*.
9. 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*, pages 48–60, June 2005.
10. K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.
11. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In T. Ball and S. K. Rajamani, editors, *Proc. SPIN2003*, volume 2648 of *LNCS*, pages 235–239. Springer, 2003.
12. A. Igarashi and N. Kobayashi. Resource usage analysis. *ACM Transactions on Programming Languages and Systems*, 27(2):264–313, 2005.
13. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *OOPSLA '99*, pages 132–146. ACM, 1999. In *SIGPLAN Notices*.
14. M. Isard and A. Birell. Automatic mutual exclusion. In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems*, 2007.
15. S. Jagannathan, J. Vitek, A. Welc, and A. Hosking. A transactional object calculus. *Science of Computer Programming*, 57(2):164–186, 2005.
16. E. B. Johnsen, O. Owe, and I. C. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 365(1–2):23–66, 2006.
17. T. Mai Thuong Tran, O. Owe, and M. Steffen. Safe typing for transactional vs. lock-based concurrency in multi-threaded Java. In *Proceedings of the Second International Conference on Knowledge and Systems Engineering, KSE 2010*, 2010. Accepted for publication.
18. F. Nielson, H.-R. Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer, 1999.
19. V. P. Ranganath and J. Hatcliff. Slicing concurrent Java programs using Indus and Kaveri. *International Journal of Software Tools and Technology Transfer*, 9(5):489–504, 2007.
20. M. Steffen and T. M. T. Tran. Safe commits for Transactional Featherweight Java. Technical Report 392, University of Oslo, Dept. of Computer Science, Oct. 2009.
21. A. Welc, S. Jagannathan, and A. Hosking. Transactional monitors for concurrent objects. In M. Odersky, editor, *18th European Conference on Object-Oriented Programming (ECOOP 2004)*, volume 3086 of *LNCS*, pages 519–542. Springer, 2004.