UNIVERSITY OF OSLO
Department of Informatics

Safe Commits for Transactional Featherweight Java[1]

Research Report No. 392

Martin Steffen and Thi Mai Thuong Tran

ISBN 82-7368-353-2
ISSN 0806-3036

October 2009

Safe Commits for Transactional Featherweight Java[†]

Martin Steffen and Thi Mai Thuong Tran

30th. October 2009

Abstract

Transactions are a high-level alternative for taming concurrency and more low-level mechanisms such as locks, semaphores, monitors, etc. One recent proposal for integrating transactional features into programming languages is *Transactional Featherweight Java* (TFJ) with syntactic support for nested and multi-threaded transactions. The transactional and concurrency systems based on this calculus model *nested* and *multi-threaded* transactions with flexible programming constructs. In this paper, we introduce a type and effect system for safe use of transactions in these systems which prevents potential errors due to free usage of constructs for starting and finishing transactions; We prove the soundness of our type system.

1 Introduction

Transactions, a well-known and successful concept originating from database systems [20, 11], have recently attracted interest to be incorporated directly into programming languages. They are advocated as a high-level, declarative alternative to more low-level mechanisms such as locks, monitors, etc.

A recent proposal for integrating transactional features into programming languages is *Transactional Featherweight Java* (TFJ) [15] with syntactic support for nested and multi-threaded transactions. The transactional model of TFJ is quite general, supporting *nested* transactions (a transaction can contain one or more child transactions) and *multi-threaded* transactions (one transaction can contain internal concurrency). Furthermore, the calculus allows to freely start and commit transactions (the respective constructs are called onacid and commit).

The flexibility 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

[*]The work has been partly supported by the EU-projects IST-33826 *Credo* (Modeling and analysis of evolutionary structures for distributed services) and FP7-231620 HATS (Highly Adaptable and Trustworthy Software using Formal Methods).

[†]The work has been partly supported by the EU-projects IST-33826 *Credo* (Modeling and analysis of evolutionary structures for distributed services) and FP7-231620 HATS (Highly Adaptable and Trustworthy Software using Formal Methods).

inside a transaction. In this paper, we introduce a static type and effect system to prevent these errors by keeping track of starting and committing transactions, formulated as an 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.

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 this paper, we choose the design which allow to put explicitly commit commands into the code of child threads, and that means that they can continue to run after committing. The primary contribution of this paper is to give a particular static type system to avoid these errors by keeping track of onacid and commit expressions in the program to guarantee that the onacid and commit commands are placed in the right place. Soundness of our type checker is given in terms of a standard subject reduction proof.

The paper is organized as follows. After Section 2 which recapitulates the syntax of the calculus and related issues, section 3 will introduce rules of the type checker and some new notations in order to keep track of onacide and commit expressions aiming to prevent *commit error* situations due to using onacid and commit commands arbitrarily. The proof of the type checker is also described 4. Section 5 raises some discussions and future works.

2 An object-oriented calculus with transactions

In this section, we present the object-oriented core calculus we use for the results. It is, with slight modifications, taken from [15] and is a variant of Featherweight Java (FJ) [13] extended with *transactions* and a construct for thread creation. We first present the syntax, and afterwards sketch a type system. The type system is fairly standard, and we include it in the technical report for completeness sake, only.

2.1 Syntax

Featherweight Java was introduced originally to study typing issues related to Java, such as inheritance, subtype polymorphism, type casts, etc., while ignoring other issues. For instance, the original proposal in [13] ignored imperative features. FJ inspired quite a number of variants and generalizations, concentrating on different languagues features of object-oriented language, and the term Featherweight Java is more understood as a generic term for Java-related core-calculi. Like in [15] and compared to the original FJ proposal, we ignore inheritance, subtyping, and type casts, as they are orthogonal to the issues at hand. On the other hand, the calculus supports imperative features, i.e., destructive field updates, furthermore concurrency and support for transactions.

Table 1 shows the abstract syntax of transactional Featherweight Java (cf. also [15]). A program consists of zero or 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 vector $\vec{f}$ represents a list of fields, under the silent assumption that all f_i's are different. The vector notation is used analogously for other entities, as well; $\vec{T}$ represents

P	::=	$0 \mid P \parallel P \mid t\langle e\rangle$	process
L	::=	$class\ C\{\vec{f}:\vec{T};K;\vec{M}\}$	class definition
K	::=	$C(\vec{f}:\vec{T})\{\mathsf{this}.\vec{f}:=\vec{f}\}$	contructor
M	::=	$m(\vec{x}{:}\vec{T})\{e\}:T$	method
e	::=	$v \mid v.f \mid v.f:=v \mid \mathsf{if}\ v\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \mathsf{let}\ x:T=e\ \mathsf{in}\ e \mid v.m(\vec{v})$	expression
	$\mid$	$\mathsf{new}\ C(\vec{v}) \mid \mathsf{spawn}\ e \mid \mathsf{onacid} \mid \mathsf{commit} \mid \mathsf{null}$	
v	::=	$r \mid x$	values

Table 1: Abstract syntax

a sequence of types, $\vec{x}$ stands for a sequence of variables, etc. When writing $\vec{x}:\vec{T}$ we silently assume that the length of $\vec{x}$ correspond to the length of $\vec{T}$, and we refer by $x_i:T_i$ to the i's pair of variable/type. We do not make explicit or formalize such kind of assumptions, when they are clear from the context; it would not be hard, but would clutter the rules and definitions without adding insight.

The syntactic category L captures class definitions. In absence of inheritance, a class $class\ C\{\vec{f}:\vec{T};K;\vec{M}\}$ consists of a name name C, a list of fields $\vec{f}$ with corresponding type declarations, a constructor K, and a list $\vec{M}$ of method definitons. A constructor $C(\vec{f}{:}\vec{T})\{\mathsf{this}.f:=\vec{f}\}$ mentions the name of the corresponding class C and its only purpose is to initialize the fields of that class, which 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 with their types, the method body, and finally the return type T of the method.

For expressions e, v stands for values, i.e., expressions that can no longer be evaluated. When leaving out in the core calculus some standard values such as integers, booleans, and the like, the only values are variables x and object references r. The expressions field access $v.f$ 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 $\mathsf{new}\ C(\vec{v})$. The next two expressions deal with the basic, sequential control structures: $\mathsf{if}\ v\,\mathsf{then}\,e_1\,\mathsf{else}\,e_2$ represents conditions, and the let-contruct $\mathsf{let}\ x:T=e_1\,\mathsf{in}\ e_2$ introduces a local variable x, evaluation e_1 before e_2. So sequential composition $e_1;e_2$ is syntactic sugar for $\mathsf{let}\ x:T=e_1\,\mathsf{in}\ e_2$ where the variable x does not occur free in e_2. The let-construct, as usual, binds x in e_2. We write $fv(e)$ for the free variables of e, defined in the standard way. The language is multi-threaded and a new thread of activity is spawned by the expression $\mathsf{spawn}\,e$, which will run in parallel with the spawning threads. 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 termininates a transaction by committing its effect. The expression null represents the null value.

A note on the form of expressions and the use of values may be in order. The

syntax is restricted concerning where to use general expressions e. For instance, Table 1 does not allow to use method 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 that need to be evaluated first. It would be easy to relax the syntax that way and indeed the proposal of TFJ from [15] allows such more general expressions. We have chosen this more restricted syntax, as it slightly simplifies the operational semantics later: [15] specify the operational semantics using so-called evaluation contexts, which fixes the order of evaluation in such more complex expressions. With that slightly restricted representation, we can get away with a semantics without evaluation contexts, using simple rewriting rules (and the let-syntax). Of course, this is not a real restriction in expressivity. For instance, the mentioned expression $e_1.f := e_2$ can easily and in a semantically equivalent manner expressed by $\mathsf{let}\, x_1 = e_1 \,\mathsf{in}(\mathsf{let}\, x_2 = e_2 \,\mathsf{in}\, x_1.f := x_2)$, making the evaluation order explicit. The transformation from a program in the more general syntax to the one of Table 1 is standard.

2.2 The underlying type system

We first describe the *underlying* type system, i.e., the standard type system for the object-oriented language that assures that actual parameters of a method call match the expected types for that method, that an object can handle an invoked method. The available types are given in equation (1). In a nominal type system, class names C serve a types. In addition, B represents basic types (left unspecified) such as booleans, integers etc. Finally, Void expresses the absence of a value, i.e., it is used for expressions that are evaluated for their side-effect, only.

$$T \quad ::= \quad C \mid B \mid \mathsf{Void} \tag{1}$$

The type system is given inductively in Table 2. For expressions, the type judgments are of the form $\Gamma \vdash e : T$ ("under type assumptions Γ, expression e has type T"). The *type environment* Γ keeps the type assumptions for local variables, basically the formal parameters of a method body and the fields. Type environments Γ are of the form $x_1{:}T_1, \ldots, x_n{:}T_1$, where we silently assume the x_i's are all different. This way, Γ is considered also as a finite mapping from variables to types. By $dom(\Gamma)$ we refer to the domain of that mapping and we 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 $x \notin dom(\Gamma)$.

Table 2 shows the rules of the underlying type system. The rules are straightforward and similar to the ones found for other variants of FJ. To define the rules, we need two additional auxiliary functions. We assume that the definitions of all classes are given. As this information is static, we do not mention the corresponding "class table" explicitly in the rules which contains the definitions of $\mathit{fields}(C)$ or $\mathit{mtype}(C,m)$

Variables are typed according to the binding found in the type environment (rule T-VAR). The null-value has the type Void (cf. rule T-NULL). A conditional expression is well-typed with type T, if both branches carry that type and if the conditional expressions is a boolean expression (cf. rule T-COND). To determine the type of a field access in rule T-FIELD, we use the *fields*-function to look up the types of the fields of appropriate class. Similarly for methods calls in rule T-METH, where *mtype* yields

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T}\ \text{T-VAR} \qquad \frac{}{\Gamma \vdash \mathsf{null} : \mathsf{Void}}\ \text{T-NULL}$$

$$\frac{\Gamma \vdash e : \mathsf{Bool} \qquad \Gamma \vdash e_1 : T \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash \mathsf{if}\, e\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 : T}\ \text{T-COND}$$

$$\frac{\Gamma \vdash e : C \qquad \mathit{fields}(C) = \vec{f} : \vec{T}}{\Gamma \vdash e.f_i : T_i}\ \text{T-FIELD}$$

$$\frac{\Gamma \vdash e : C \qquad \mathit{mtype}(C,m) : \vec{S} \to T \qquad \Gamma \vdash \vec{e} : \vec{S}}{\Gamma \vdash e.m(\vec{e}) : T}\ \text{T-CALL}$$

$$\frac{\Gamma \vdash e_1 : C \qquad \mathit{fields}(C) = \vec{f} : \vec{T} \qquad \Gamma \vdash e_2 : T_i}{\Gamma \vdash e_1.f_i := e_2 : T_i}\ \text{T-ASSGN}$$

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma, x{:}T_1 \vdash e_2 : T_2}{\Gamma \vdash \mathsf{let}\, x : T_1 = e_1\, \mathsf{in}\, e_2 : T_2}\ \text{T-LET} \qquad \frac{C \in \Gamma}{\Gamma \vdash \mathsf{new}\, C : C}\ \text{T-NEW}$$

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash \mathsf{spawn}\, e : \mathsf{Void}}\ \text{T-SPAWN}$$

$$\frac{}{\Gamma \vdash \mathsf{onacid} : \mathsf{Void}}\ \text{T-ONACID} \qquad \frac{}{\Gamma \vdash \mathsf{commit} : \mathsf{Void}}\ \text{T-COMMIT}$$

$$\frac{\vec{x}{:}\vec{S}, \mathsf{this}{:}C \vdash e : T}{\vdash m(\vec{x} : \vec{S})\{e\} : T : \mathit{ok}\ \mathit{in}\ C}\ \text{T-METH}$$

$$\frac{K = C(\vec{f} : \vec{T})\{\mathsf{this}.\vec{f} := \vec{f}\} \qquad \vdash \vec{M} : \mathit{ok}\ \mathit{in}\ C}{\vdash \mathit{class}\ C\{\vec{f} : \vec{T}; K; \vec{M}\} : \mathit{ok}}\ \text{T-CLASS}$$

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

Table 2: The underlying type system

the type of the method as found in the concerned class. For assignments $e_1.f := e_2$, in rule T-ASSGN, the type of the appropriate field is determined using *fields* as for field access, and furthermore checked that the type of e_2 coincides with it. The type of a sequential composition $e_1; e_2$ is the type of the last expression e_2 (cf. rule T-SEQ). A freshly instantiated object carries the class it instantiates as type, and spawning a new thread has a side-effect, only, but returns no value, hence spawn carries type Void (cf. rules T-NEW and T-SPAWN). Similarly, the two operations for starting and ending a

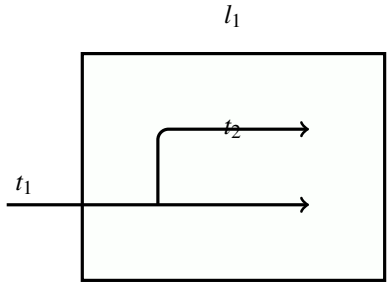

Figure 1: Multi-threaded transaction

transaction, onacid and commit, are evaluated for their effect, only, and carry both the type Void (cf. rules T-ONACID and T-COMMIT).

Rule T-METH deals with method declarations, explicitly mentioning the class C which contains the declaration. The body e of the method is type checked, under an type environment extended by appropriate assumptions for the formal parameters x and by assuming type C for the self-parameter this. A class definition $class\ C\{\vec{f}{:}\vec{T}; K; \vec{M}\}$ is well-typed (cf. rule T-CLASS), if all methods $\vec{M}$ are well-typed in C (the premise $\vdash \vec{M}$ $:ok\ in\ c$ of the rule is meant as iterating over all of the class's methods, using T-METH for each individual one). A thread $t\langle e\rangle$ is well-typed, if the expression e it evaluates is (cf. rule T-THREAD). Rule T-EMPTY and T-PAR assure that a program is well-typed if all its processes are.

3 The Effect system

In our setting, the purpose of the effect system is to determine correct use of the starting and committing transactions, in particular to avoid committing when one is not inside a transaction. Such a situation constitutes an error, we call it a *commit error*. To avoid such erroneous situations, the effect system keeps track of onacid and commit in the code; we refer to the number of onacid minus the number of commits as the *balance*. An execution of a thread is *balanced,* if all begun transactions are committed, i.e., if the balance equals 0.

The situation gets slightly more involved when dealing with multi-threading. TFJ supports not only nested transactions, but *multi-threaded* transactions. I.e., inside one transaction there might be more than one thread active at a time, which, for instance, can yield the effect of the transaction, upon commit, non-determinstic due to internal concurrency. Figure 1 shows a simple situation with two threads t_1 and t_2, where t_1 starts a transaction, and spawns a new thread t_2 inside the transaction. So an example expression resulting in the depicted behavior is $e_1 = \mathsf{onacid}; \mathsf{spawn}\, e_2; e_1'$, where e_1 is the expression evaluated by thread t_1, and e_1 by the freshly created t_2.

Now, both t_1 and t_2 must *join* in a common commit, in order to terminate the transaction. So in order to keep track over the number of open and yet uncommitted transaction, it's we must take into account that e_2 and the rest e_1' of the original thread are

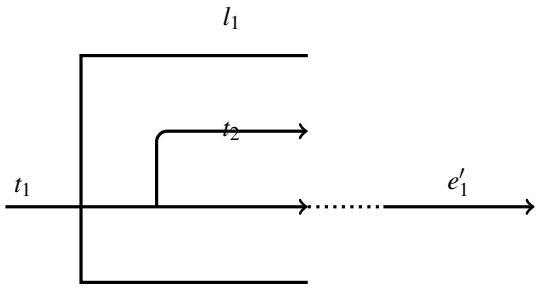

Figure 2: Sequential composition

executed in parallel, and furthermore, that when executing e_2 in the new thread t_2, already one onacid has been executed by t_1, namely before the spawn-operation. A consequence for the effect system is that we need to keep track of the balance not just for the thread/expression 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, without preference or priority to either one (indeed, in the semantics, the information which threads is the father of which other thread is not maintained), the situation wrt. the analysis is not symmetric. More precisely, the current thread of control plays a specific role when it comes to sequential composition of expressions. Consider the expression $\mathsf{onacid};\mathsf{spawn}\,e_2$ followed by e'_1. The first expression is depicted left of Figure 2, where the "open box" represents the transaction started by the first onacid. Considering the balance for the left $\mathsf{onacid};\mathsf{spawn}\,e_2$, the balance for both "threads" after execution amouts to 1, i.e., both threads are execution inside one enclosing transaction (assuming that e_2 itself does not do start or end any transactions). When calculating the combined effect in the analysis for $\mathsf{onacid};\mathsf{spawn}\,e_2$ followed sequentially by e'_1, the balance value of onacid is treated different from the one of e_2, since the control flow of the sequential composition connects the trailing e'_1 with onacid, but not with the thread of e_2 (indicated by the dotted line in Figure 2). This means, for the analysis of the sequentially composed expression, the *sum* of the balances for onacid and of e'_1 must be calculated.

To sum up: to determine the effect in terms of the balance, we need to calculate the balance for potentially *all* threads 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 evaluate plays a special role, and is treated specially. The effect after evaluating an expression is therefore represented as a pair of an integer n and a (finite) multi-set S of integers:

$$n, S : \mathsf{Int} \times (\mathsf{Int} \to \mathsf{Nat}) \tag{2}$$

We write $\emptyset$ for the empty multi-set, $\cup$ for the multi-set union. Alternatively, the multi-set can be seen as a function of type $\mathsf{Int} \to \mathsf{Nat}$, and we write $dom(S)$ for the set of elements of S, ignoring their multiplicity. The integer represents the balance of the thread of the given expression, the multi-set the balance numbers for the threads

spawned by the expression. The judgements of the analysis are thus of the following form:

$$n_1 \vdash e :: n_2, S\,, \tag{3}$$

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 in a *cumulative* manner, i.e., their balance includes n_1, the contribution of e before the thread is spawned, and the contribution of the new thread itself.

For clarity, we do not integrate the effect system with the underlying type system of Section 2.2. Instead, we concentrate on the effects in isolation. In the appendix, we show the combined type and effect system. Variables, the null-expression, and field access have no effects (cf. rules T-VAR, T-NULL, and T-FIELD, so they leave the balance unchanged and since no threads are generated, the multi-set of balances is empty. For conditionals $\mathsf{if}\, e \,\mathsf{then}\, e_1 \,\mathsf{else}\, e_2$, we insist that the (boolean) expression e does not change the balance, and that the two branches e_1 and e_2 agree on a balance n'. An assignment has no effect (cf. rule T-ASSGN), as we require, that the expression being assigned and the expression designating the object, whose field we assgn to, are evaluated already. Likewise, creating a new object has no effect (cf. rule T-NEW).

In a sequential composition (cf. rule T-SEQ), the effects are accumulated. Creating a new thread by executing $\mathsf{spawn}\, e$ 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 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. The dual two commands of onacid and commit simply increase, resp. decrease the balance by 1 (cf. rule T-ONACID and T-COMMIT). The rule T-METH deals with method declarations. Rule T-SUB captures a notion of *subsumption* where by $S_1 \leq S_2$ we mean the subset relation on multi-sets.

We illustrate the effect system with the following example:

Example 3.1. *The follwing derivation shows the effect for 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*.*

$$\dfrac{0 \vdash e_1 :: n_1', \{\} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{n_2 \vdash e_3 :: n_3, \{\}}}{n_1' \vdash e_2 :: n_2, \{\} \qquad n_2 \vdash \mathsf{spawn}\, e_3 :: n_2, \{n_3\}}}{n_1' \vdash (e_2; \mathsf{spawn}\, e_3) :: n_2, \{n_3\}}}{n_1' \vdash \mathsf{spawn}(e_2; \mathsf{spawn}\, e_3) :: n_1', \{n_2, n_3\} \qquad n_1' \vdash e_4 :: n_4, \{\}}}{n_1' \vdash \mathsf{spawn}(e_2; \mathsf{spawn}\, e_3); e_4 :: n_4, \{n_2, n_3\}}}{0 \vdash e_1; \mathsf{spawn}(e_2; \mathsf{spawn}\, e_3); e_4 :: n_4, \{n_2, n_3\}}$$

The derivation demonstrates the two most interesting rules T-SEQ *and* T-SPAWN *with a starting balance is 0 for simplicity. Assume that the expressions* $e_1, \ldots e_4$ *themselves have the following balances* $0 \vdash e_i :: n_i', \{\}$*. Based on Lema 3.2 we have:*

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

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

$$\frac{n \vdash v :: n, \emptyset \qquad mtype(C,m) :: n', S \qquad 0 \vdash \vec{e} :: \vec{n}, \vec{S}}{n \vdash v.m(\vec{e}) : n + n' + \sum_i n_i, S \cup \bigcup_i (S_i + n)}\ \text{T-CALL}$$

$$\frac{n \vdash v_1 : n, \emptyset \qquad n \vdash v_2 : n, \emptyset}{n \vdash v_1.f_i := v_2 : n, \emptyset}\ \text{T-ASSGN} \qquad \frac{}{n \vdash \mathsf{new} C :: n, \emptyset}\ \text{T-NEW}$$

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

$$\frac{}{n \vdash \mathsf{onacid} : n+1, \emptyset}\ \text{T-ONACID} \qquad \frac{n \geq 1}{n \vdash \mathsf{commit} : n-1, \emptyset}\ \text{T-COMMIT}$$

$$\frac{n \vdash e : n', S_1 \qquad S_1 \leq S_2}{n \vdash e : n', S_2}\ \text{T-SUB}$$

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

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

Table 3: Effect system

$$\begin{array}{l} n_1' \vdash e_2 :: n_1' + n_2' = n_2 \\ n_2 \vdash e_3 :: n_2 + n_3' = n_3 \\ n_1' \vdash e_4 :: n_1' + n_4' = n_4 \end{array}$$

In order to check the type of T-SEQ*, we have to apply* T-SEQ *for the whole expression from left to right (note that calculation performing in the other way around is similar and have the same result). First we evaluate the type of the first expression* e_1 *and then the rest as the second expression* $\mathsf{spawn}(e_2; \mathsf{spawn}\, e_3); e_4$.

Lemma 3.2. *If* $n_1 \vdash e :: n_2$*, then* $n_1 + n \vdash e :: n_2 + n$.

Proof. By straightforward induction over the length of derivation. □

4 Semantics

In this section, we describe the dynamic semantics of TFJ in terms of two different levels: local semantics and global semantics.

The local semantics is given in Table 4 on the following page. These local rules specify the "core" of the semantics, as the paper claims; there are 4 of those and they work as follows. Note that the rule specify exactly one command, not a whole expression with an expression inside. The four rules are pretty straightforward, as far as the expression is concerned. As well as the second component of the local configuration, i.e., the local environment, involved, the situation is a bit more complex, or rather a bit unspecific, as the details of E is given only abstractly. Only later, E and the corresponding manipulations are concretized, yielding in this paper, the *versioning* semantics and the semantics with strict two-phase locking. In the premises of the rules, the E is consulted, looking up object references. Not only that, the E is also changed. The change is not only done when executing a *write*-command, but also for a read command: in the versioning semantic, it records, information about of accesses to the storage, and it might *copy* in values into the log of the local transaction.

Rule R-FIELD deals with field lookup; the command is of the syntactic form $r.f_i$, where r is a location and f_i is the i's field of the object referenced by r. Basically, the read-function consults the environment E to look-up r, and finds the *object* $C(\vec{r})$ referenced by r. Besides that, the read-function changes the E environment to E'. This change, as said, will be described and concretized later. The change, abstractly, will somehow mention/record, *in which transaction* the location is referenced. Note that at this stage (with *read* etc. not concretely given), there is no mentioning of transactions in the reduction rules, so we don't see in which transaction the reduction takes place, at least not on the transition label.

The next rule R-ASSIGN treats assignment. As in the rule for look-up, the environment is consulted to retrieve the object $C(\vec{r})$ (and changing E to E' as a side-effect). The second premise does the update itself, using the *write*-operation. In the label, both the location that contains the object and the location that is stored in the field, are mentioned. Rule R-INVOKE deals with method invocation, and works as expected. The final, local rule R-NEW specifies object instantiation, and uses the *extend*-operation

The five rules of the *global* semantics are given in Table 5. The global operational semantics works on configurations of the following form:

$$\Gamma \vdash P \tag{4}$$

where P is a *program* and Γ is a *program state.* We will call Γ a *global environment.* Basically, a program P consists of a number of threads evaluated in parallel (cf. the abstract syntax from Table 1 on page 3), where each thread corresponds to one expression, whose evaluation is described by the local rules. As seen, each local expression/thread has a sequence E of bindings, called the local environment. 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, called *thread environments*.

Definition 4.1 (Global enviroment)**.** *A* global environment Γ *of type GEnv is a finite mapping from thread names to local environments. A global environment is written as*

$$\frac{}{E \vdash \mathsf{let}\, x : T = \mathsf{null}\, \mathsf{in}\, e \rightarrow E \vdash e}\ \text{R-NULL}$$

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

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

$$\frac{\mathit{read}(r,E) = E', C(\vec{u}) \qquad \mathit{fields}(C) = \vec{f}}{E \vdash \mathsf{let}\, x : T = r.f_i\, \mathsf{in}\, e \xrightarrow{rd\ r} E' \vdash \mathsf{let}\, x : T = u_i\, \mathsf{in}\, e}\ \text{R-FIELD}$$

$$\frac{\mathit{read}(r,E) = E', C(\vec{r}) \qquad \mathit{write}(r \mapsto C(\vec{r}) \downarrow_i^{r'}, E') = E''}{E \vdash \mathsf{let}\, x : T = r.f_i := r'\, \mathsf{in}\, e \xrightarrow{wr\ rr'} E'' \vdash \mathsf{let}\, x : T = r'\, \mathsf{in}\, e}\ \text{R-ASSIGN}$$

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

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

$$\frac{}{E \vdash \mathsf{let}\, x : T (\mathsf{if}\ v = v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)\, \mathsf{in}\, e \rightarrow E \vdash \mathsf{let}\, x : T = e_1\, \mathsf{in}\, e}\ \text{R-COND}_1$$

$$\frac{v_1 \neq v_2}{E \vdash \mathsf{let}\, x : T = (\mathsf{if}\ v_1 = v_2\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)\, \mathsf{in}\, e \rightarrow E \vdash \mathsf{let}\, x : T = e_2\, \mathsf{in}\, e}\ \text{R-COND}_2$$

Table 4: Semantics (local)

$t_1{:}E_1, \ldots t_k{:}E_k$ *(the order of bindings does not play a role, and each thread name can occur at most once).*

So the steps are of the form:

$$\Gamma \vdash P \stackrel{\alpha}{\Longrightarrow}_t \Gamma' \vdash P' \tag{5}$$

The subscript t denotes the identity of the thread that does the step, and α the label, as before. Besides the 3 labels for the local steps, there will be more kinds of labels, however.

The rule G-PLAIN simply *lifts* a local steps to the global level. And thus the reflect-operation also lifts the change of E' to Γ'. The rule SPAWN deals with the spawn-expression, which starts a new thread. As that involves two threads, the original and the new one, it is being dealt with at the global level. The transition is labeled with the original thread t that executes those steps. The identity t' of the new thread must be fresh. Note that the calculus allows thread creation, but does not feature thread classes. Also, the functionality of the *spawn*-operation will be dealt with later. Anyway, a freshly created thread executes in the same transaction(s) as the spawning thread.

The next three important rules treat the two central commands of the calculus, those dealing with directly with the transactions. The first one G-TRANS covers onacid, which starts a transaction. The rule here itself is pretty unspectacular. The step simply creates a new label l, and all the work is done in the implementation of *start*. Rule G-COMMIT deals with committing a transaction, i.e., making its effect globally known in an instantaneous manner. The step from P to $\acute{P}$ seems clear enough. The rest is a bit more complex. What is a bit strange is that this rule is not longer too abstract, i.e., suddenly the inner structure of Γ and E is exposed (and not postponed until we get a special incarnation of the transactional semantics). The Γ contains the binding for the thread t which actually performs the commitment statement. The associated E is consulted to get the name of the transaction. The function *intranse* takes a transaction label l and a global environment Γ, and gives back the sequence (or rather the set) of thread labels, which are running *inside* the mentioned transaction. The argument Γ is a list (or rather a set) associating for each thread a local environment and the function iterates therefore through that set. In the introductory part, they state that a thread can execute *within* a transaction l. However, transactions can be nested. So that might mean, a thread may (indirectly) execute within more than one transaction. Maybe: The configuration contains a number of threads in front of their respective commit-statement. They are all take in one step, so that seems to be some *join*-synchronization. And the third rule says that an *commit error* will be raised if the program commits non-transaction code.

The following function is needed to define the nestedness of threads within transitions in Definition 4.3 below.

Definition 4.2. *Given a local environment E, the function $l : LEnv \to List\ of\ TrName$ is defined inductively as follows: $l(\varepsilon) = \varepsilon$, and $l(l{:}\_, E) = l, l(E)$.*

Overloading the definition, we lift the function straightforwardly to global environments (with type $l : TName \times GEnv \to List\ of\ TrName$), such that $l(t, (t{:}E), \Gamma) = l(E)$.

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

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

$$\frac{P = P' \parallel t\langle r\rangle \qquad \Gamma = t{:}E, \Gamma'}{\Gamma \vdash P \xRightarrow{t}_{ki} \Gamma' \vdash P'} \text{ G-THKILL}$$

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

$$\frac{P = P'' \parallel t\langle \mathsf{let}\, x : T = \mathsf{commit}\, \mathsf{in}\, e\rangle \qquad P' = P'' \parallel t\langle \mathsf{let}\, x : T = \mathsf{null}\, \mathsf{in}\, e\rangle \qquad \Gamma = \Gamma'', t{:}E \qquad E = E', l{:}\rho \qquad \mathit{intranse}(l, \Gamma) = \vec{t} = t_1 \ldots t_k \qquad \mathit{commit}(\vec{t}, \vec{E}, \Gamma) = \Gamma' \quad n_1{:}E_1, n_2{:}E_2, \ldots n_k{:}E_k \in \Gamma \quad \vec{E} = E_1, E_2, \ldots, E_k}{\Gamma \vdash P \xRightarrow{t}_{co} \Gamma' \vdash P'} \text{ G-COMM}$$

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

Table 5: Semantics (global)

The first definition, extracting the list of transaction labels from a local environment E is a straightforward projection, simply extracting the sequence of transaction labels. As for the *order* of the transactions, as said: the most recent, the innermost transaction label is to the right.

Given a transaction, the following function determines the threads for which the given transaction is (properly) "nested" in a global environment, i.e., those threads which execute *inside* the given transaction but where the transaction is *not the current, directly enclosing* transaction. So, "nested" may be a slight misnomer: It does not mean that a transaction occurs *not outermost.*

Definition 4.3 (Nesting)**.** *Given a global environment, the function nested* : $TrName \times GEnv \to List\ of\ TName$ *returns the list/set of all threads nested inside a given transaction.*

Next we prove that the type and effect system does what it is designed to do. The safety property the type system guarantees is the absence of commit errors. The main part of the proof is preservation of well-typedness under reduction, also called *subject reduction.*

Lemma 4.4 (Subject reduction (local))**.** *Let* $n = |E|$. *If* $n \vdash e :: n', S'$ *and* $E \vdash e \xrightarrow{\alpha} E' \vdash e'$, *then* $|E'| = n$ *and* $n \vdash e' :: n', S'$. *(we have to define lenght of E)*

Proof. First observe that by the properties of *read*, *write*, and *extend*, $|E| = |E'|$. Proceed by case analysis over the operational rules of Table 4. The cases R-FIELD, R-ASSIGN, R-INVOKE, and NEW are immediate. For R-COND$_1$, we need subsumption (R-COND$_2$ works analogously):

Case: R-COND$_1$
In this case the expression e before the step is of the form $\mathsf{if}\, v = v\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2$ and the step is given as $E \vdash e \to E \vdash e_1$. Note that $E' = E$, and hence $|E'| = n$. Concerning the typing, we are given $n \vdash \mathsf{if}\, v = v\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 :: n', S'$, which implies by the premises of rule T-COND that $n \vdash e_1 :: n', S_1$ and $n \vdash e_2 :: n', S_2$ with $S' = S'_1 \cup S'_2$. The result follows by subsumption (rule T-SUB). □

The global semantics accesses and changes the global enviroments 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 4.5. *The following functions on global environments are specified as follows.*

1. *The function reflect satisfies the following condition: if* $reflect(t, E, \Gamma) = \Gamma'$ *and* $\Gamma = t_1{:}E_1, \ldots, t_n{:}E_n$, *then* $\Gamma' = t_1{:}E'_1, \ldots, t_n{:}E'_n$ *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* $\Gamma = t_1{:}E_k,\ldots,t_k{:}E_k$ *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* $commit(E,t,\Gamma) = \Gamma'$ *for a* $\Gamma = \Gamma'',t{:}E$ *and for a* $\vec{t} = intranse(l,T)$*, and for a* $P = P'' \parallel t_i\langle \mathsf{commit}; e_i\rangle$ *then* $\Gamma' = t'_i{:}E''_i,t_i{:}E'_i$ *where* $t_i \in \vec{t},t'_i \notin \vec{t}$*,* $t'_i{:}E'_i \in \Gamma$ *and* $P' = t'_i\langle e'_i\rangle \parallel t_i\langle \mathsf{null}; e_i\rangle$*, with* $|E''_i| = |E'_i|$ *and* $|E'_i| = |E_i|-1$.

Lemma 4.6 (Subject reduction)**.** *If* $\Gamma \vdash P : ok$ *and* $\Gamma \vdash P \rightarrow \Gamma' \vdash P'$*, then* $\Gamma' \vdash P' : ok$.

Proof. Proceed by case analysis on the rules of the operational semantics from Table 5 (except rule G-COMMERROR for commit errors).

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 $E \vdash e \xrightarrow{\alpha} E' \vdash e'$ 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 4.4) $n \vdash e' :: n',S'$. 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-PARALLEL and the properties of *reflect* from Definition 4.5.1).

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

$$\frac{\dfrac{\dfrac{n \vdash e_1 :\ 0,S_1}{n \vdash \mathsf{spawn}\, e_1 :\ n,S_1 \cup \{0\}} \qquad n \vdash e_2 :\ 0,S_2}{n \vdash \mathsf{let\ spawn}\, e_1\, \mathsf{in}\ e_2 :\ 0,\{0,\ldots\}}}{t{:}E \vdash t\langle \mathsf{let\ spawn}\, e_1\, \mathsf{in}\ e_2\rangle : ok} \tag{6}$$

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 4.5.2). So we can derive

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

The left sub-goal follows from T-THREAD,T-SEQ, T-NULL, and the right subgoal of the previous derivation (6). The right open subgoal directly corresponds to the left subgoal of derivation (6).

Case: G-THKILL
Straightforward.

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

$$\frac{\dfrac{n \vdash \mathsf{onacid} :: n+1, \emptyset \quad n+1 \vdash e :: 0, \{0,\ldots\}}{n \vdash \mathsf{let\ onacid\ in}\ e :: 0, \{0,\ldots\}}}{t{:}E \vdash t\langle \mathsf{let\ onacid\ in}\ e\rangle : ok} \tag{7}$$

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 4.5.3, we have $\Gamma' = \Gamma'', t{:}E'$ with $|E'| = n+1$. So with the help of right subgoal of the previos derivation (7), 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 4.5.3), the required $\Gamma' \vdash P' : ok$ can be derived, using T-PAR.

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

$$\frac{\dfrac{n \vdash \mathsf{commit} :: n-1, \emptyset \quad n-1 \vdash e_i :\ 0, \{0,\ldots\}}{n \vdash \mathsf{let\ commit\ in}\ e_i :\ 0, \{0,\ldots\}}}{t_i{:}E_i \vdash t_i\langle \mathsf{let\ commit\ in}\ e_i\rangle : ok} \tag{8}$$

For the global environment Γ' after the step, we are given $\Gamma' = \mathit{commit}(\vec{t}, \vec{E}, \Gamma)$ from the premise of rule G-TRANS, where $\vec{t} = \mathit{intranse}(l, \Gamma)$ and $\vec{E}$ are the corresponding local environments. By the properties of *commit* from Definition 4.5.5, 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 (8):

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

For the threads $t_i'\langle e_i'\rangle$ different from t, according to the Definition 4.5.5, we have $|E_i''| = |E_i'|$ so $t_i'{:}E_i'' \vdash t_i'\langle e_i'\rangle : ok$ straightforwardly. As a result, we have $\Gamma' \vdash P' : ok$. □

Lemma 4.7. *If* $\Gamma \vdash P : ok$ *then it is not the case that* $\Gamma \vdash P \rightarrow error$.

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 $P = t\langle \mathsf{let\ commit\ in}\ e\rangle \parallel P'$ 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:

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

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

Corollary 4.8 (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 4.6 and Lemma 4.7. □

5 Conclusion

We presented a type and effect system to avoid improper use of transaction operations, such as not committing a transaction, or committing a transaction when not executing inisde one. In order to keep track of using correctly onacid and commit commands, we have introduced a notation in the type-effect system to accumulate the balance for potentially *all* threads concerned. The soundness of our type checker was proven in the paper to make sure that type errors will not happen under our type system.

5.1 Related work

This paper took the language design of [15] TFJ. That paper is not concerned with the type system and static analysis, but develops and investigates operational semantics for TFJ that assures transactional guarantees. FJ and its variant TFJ are core languages to capture essential features of real languages, in this case of Java resp. of transactional extensions for Java. The proposal of TFJ concerning transaction is rather advanced (allowing nested and concurrent transactions), compared to the traditional way Java deals with concurrency control: each object is equipped with a lock, and the `synchronized`-statement (resp. `sychnronized` methods) can be used to achieve mutual exclusion. The monitors in Java work lock-based (and are not based on transactions); more importantly in the context of our static analysis is: the use of `synchronized` is rather restricted compared to the situation in TFJ, as locking has to adhere to *lexical scoping.* Recent library extensions to Java ("Java 5") give more flexibility in allowing

non-lexically scoped locks via the interface `Lock`. TFJ studied here is, likewise, non-lexically scoped, but allows still more freedom in supporting multi-threading inside one transaction. Therefore, the type-system here can be applied to the simpler setting of Java 5 locks, as well.

There have been a number of further proposals for integrating transactional features into programming languages. The paper [1] presents the AME calculus, a calculus for *automatic mutual conclusion*, a concept proposed in [14]. The sequential core is some λ-calculus with references and imperative update, extended by the possibility to create asynchronouse 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 calculcus and a proof method (implemented in the tool QED) for atomic actions is presented in [8]. Also the following languages or calculi are concerned with transactions, too. AtomCaml [19], X10 [5], Fortress [3], Chapel [6]

Over the years, many static analyses for different purposes and language features have been devised, to assure desired properties ranging from resource consumption (e.g., concerning memory, time ...), absence of deadlocks and race conditions. Most authors (e.g., [4][2][17]...) focus on avoiding data races and deadlocks in multi-threaded Java programs relating to shared-memory issues and synchronization. Static type systems have also been used to impose restrictions assuring transactional semantics, for instance in [12][1][14] A type system for *atomicity* [10][9].

5.2 Future work

The work presented here can be extended to deal with more complex type systems, for instance when dealing with higher-order functions. In that setting, the effect part and the connection to the type system becomes for challenging. Furthermore, we plan to adopt the results for a different language design, more precisely to the language Creol [7, 16], which is based on asynchronously communicating, active objects, in contrast to Java, whose concurrency is based on multi-threading.

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, Jan. 2008.

[2] M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for Java. *ACM Trans. Program. Lang. Syst.*, 28(2):207–255, 2006.

[3] 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.

[4] 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*, pages 211–230, 2002.

[5] 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, pages 519–538. ACM, 2005. In *SIGPLAN Notices*.

[6] Cray. Chapel specification, Feb. 2005.

[7] The Creol language. `http://heim.ifi.uio.no/creol`, 2007.

[8] T. Elmas, S. Qadeer, and S. Tasiran. A calculus of atomic actions. In *Proceedings of POPL '09*. ACM, Jan. 2009.

[9] C. Flanagan and S. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Proceedings of POPL '04*, pages 256–267. ACM, Jan. 2004.

[10] 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, June 2003.

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

[12] T. Harris, S. M. S. Peyton Jones, and M. Herlihy. Composable memory transactions. In *PPoPP'05: 10t ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 48–60, June 2005.

[13] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Object Oriented Programming: Systems, Languages, and Applications (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, August 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, Nov. 2006.

[17] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, pages 308–319. ACM, 2006.

[18] F. Nielson, H.-R. Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[19] M. F. Ringenburg and D. Grossman. AtomCaml: First-class atomicity via rollback. In *ACM International Conference on Functional Programming*, pages 92–104. ACM, 2005. In *SIGPLAN Notices*.

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

A Notes

The general questions about transactions is kept in the `paper` directory, here there are only questions concerning the specific things.

A.1 Development

29. Sept. 2009 we switched to let-semantics

Sept. 2009: Abstract to NWPT

End of July 2009: start

A.2 Special questions

Question A.1 (Spawning)**.** *What is the problem with spawning new threads when it comes to counting the balance?*

Answer: Maybe it's a weakness of the syntax. The problem is that we cannot have a in a directly compositional manner the type of $\texttt{e1;e2}$ calculated from e_1 and e_2, if the balance is just a number. One can see it by the fact that the type system cannot distinguish between $\texttt{e1; e2}$ (when we assume that e_1 and e_2 is single-threaded, and $\texttt{(spawn e1); e2}$. So it means we need to take the "threads" intro account. Let's ignore the standard types. Let's further assume that $\Theta = \vec{t} : \vec{n}$. Let's assume futher that t_0 is the standard thread.

$$\frac{\Gamma \vdash e : \Theta}{\Gamma \vdash \mathsf{spawn}\, e_1 : \Theta, t_0{:}0}\ \text{T-SPAWN}$$

Furthermore, the rule for sequential composition can be as follows

---

$$\frac{\Gamma \vdash e_1 : \Theta_1, t_0{:}n_1 \qquad \Gamma \vdash e_1 : \Theta_2, t_0{:}n_2}{\Gamma \vdash e_1; e_2 : \Theta_1, \Theta_2, n_1 + n_2}\ \text{T-SEQ}$$

□

Index