# A Type System for Counting Logs of Multi-threaded Nested Transactional Programs[*]

Anh-Hoang Truong[1], Dang Van Hung[1], Duc-Hanh Dang[1], and Xuan-Tung Vu[2]

[1] VNU University of Engineering and Technology - Hanoi
[2] Japan Advanced Institute of Science and Technology

**Abstract.** We present a type system to estimate an upper bound for the resource consumption of nested and multi-threaded transactional programs. The resource is abstracted as transaction logs. In comparison to our previous work on type and effect systems for Transactional Featherweight Java, this work exploits the natural composition of thread creation to give types to sub-terms. As a result, our new type system is simpler and more effective than our previous one. More important, it is more precise than our previous type system. We also show a sketch proof for the correctness of the estimation and a type inference algorithm that we have implemented in a prototype tool.

**Keywords**: *resource bound, software transactional memory, type systems*

## 1 Introduction

Software Transactional Memory [12] has been introduced as an alternative to the locked-based synchronization for the shared memory concurrency. It has become a focus for intensive theoretical researches and practical applications for quite a long time. One of the recent transactional models that support advanced features of programming such as nested and multi-threaded transactions is described in [9]. In this model, a transaction is said to be nested if it is contained in another transaction; the former is called child transaction, and the latter is called parent transaction. The rule is that the child transaction must commit before their parent does. Furthermore, a transaction is multi-threaded when threads are created and run inside the transaction. These threads when created, run in parallel with the thread executing that transaction. For independent manipulation of shared variables, a child thread will make a copy of all variables of its parent thread. When the parent thread commits a transaction, all the child threads created inside that transaction must join the commit of their parent. We call this kind of commits *joint commits*, and the time when these commits

[*] This research is funded by Vietnam National Foundation for Science and Technology Development (NAFOSTED) under grant number 102.03-2014.23. The research is also partly supported by the research project QG.14.06, Vietnam National University, Hanoi

occur *joint commit point*. Joint commits act as the synchronizations of parallel threads.

In the implementations in practice, each transaction has its own local copy of memory called log to store shared variables for independent accesses during its execution. Each thread may execute a number of nested transactions, and thus may contain a corresponding number of logs. In addition, a child thread also stores a copy of its parent's logs so that it can be executed independently with its parents until a joint commit point. At the time when all child threads and their parent are synchronized via joint commits, their own logs and the copies they keep relating to the transaction are consulted to check for conflicts and potentially performing a roll-back. These logs are then discarded – the corresponding memory resources are freed. A major complication for the static analysis is that the number of logs cloned (resource allocation) when a new thread is created is implicit and the commit statements (resource deallocation) that need to synchronize with each other are also implicit, so the resources used by these programs are difficult to estimate.

The number of cloned logs may affect the efficiency of parallel threads and the maximal number of logs that coexist may affect the safety of the program when the memory is limited. Therefore, a precise estimation of the upper bound of number of coexisting logs in a multi-threaded, nested transactional memory program plays a crucial role.

In our previous works [10, 14] we gave type and effect systems for estimating the upper bounds of number of logs that coexist at the same time. We are not satisfied with those estimations, and take a new approach to solve the problem. This work was initiated from our previous work [14] with the advantage that our new system can infer more precise bound, and it is simpler with some natural sacrifice on the compositionality.

The main contributions of this work are as follows:

– A simpler type system with natural composition of terms.
– Our correctness proofs are all briefly shown.
– A type inference algorithm and a tool for inferring types for the core part of the language.

**Related work** Estimating resource usage has been studied in various settings. Hughes and Pareto [8] introduce a strict, first order functional language with a type system such that well-typed programs run within the space specified by the programmer. Tofte and Talpin [13] use inference system to describe a memory management for programs that perform dynamic memory allocation and de-allocation.

Hofmann and Jost [6] compute the linear bounds on heap space for a first-order functional language. In terms of imperative and object-oriented languages, Wei-Ngan Chin et al. [5] verify memory usages for object-oriented programs. Programmers are required to annotate the memory usage and size relations for methods as well as explicit de-allocation. In [7], Hofmann and Jost use a type system to calculate the heap space bound as a function of input for an object oriented language. In [4] the authors statically compute upper bounds of resource consumption of a method using a non-linear function of method's parameters. The bounds are not precise and their work is not type-based. Braberman et al. [2] calculate non-linear symbolic approximation of memory bounds for Java programs involving both data structures and loops. In [3] the authors propose type systems for component languages with parallel composition but the threads run independently. In [1], Albert  et al. compute the heap consumption of a program as a function of its data size. [11] proposes a fast algorithm to statically find the upper bounds of heap memory for a class of JavaCard programs.

Our analysis not only takes care of multi-threading – many of the cited works are restricted to sequential or functional languages – but also of the complex and implicit synchronization (by joint commits) structure entailed by the transactional model.

The rest of the paper is structured as follows. In the next section we will explain informally the problem and the approach by a motivating example. Section 3 introduces the formal syntax and operational semantics of the calculus. Section 4 presents a new type system. The soundness of the analysis is sketched in Section 5 and then, a type inference algorithm is sketched in Section 6. Section 7 is the conclusion of the paper.

## 2 Motivating Example

We use the following example program (see Listing 1.1) which is taken from our previous work [10, 14] as our running example to demonstrate the problem and our new approach.

**Listing 1.1.** A nested  multi-threaded program.

```
onacid;//thread 0
 onacid;
  spawn(e1;commit;commit);//thread 1
  onacid;
   spawn(e2;commit;commit;commit);//thread 2
  commit;
  e3;
 commit;
 e4;
commit
```

In this program, the statements `onacid` and `commit` are to start and to close a transaction, respectively, and must be paired. The expressions `e1, e2, e3` and `e4` represent subprograms. The statement `spawn` is to open a thread with the code represented by the parameter of the statement. The behavior of this program is depicted in Figure 1. The starting transaction command `onacid` and ending transaction command `commit` are denoted by [ and ] in the figure, respectively. The command `spawn` creates a new thread running in parallel with its parent thread. The new thread duplicates the logs of the parent thread for storing a copy of variables of the parent thread. In our example, when spawning `e1` the main thread has opened two transactions; so thread 1 executes `e1` inside these two transactions and must do two commits to close them. This is why after `e1`, thread 1 needs to execute two `commit` commands.

Figure 1 illustrates that the parallel threads of a transaction must commit the transaction at the same time. The right-hand edges of the boxes mark these synchronizations.

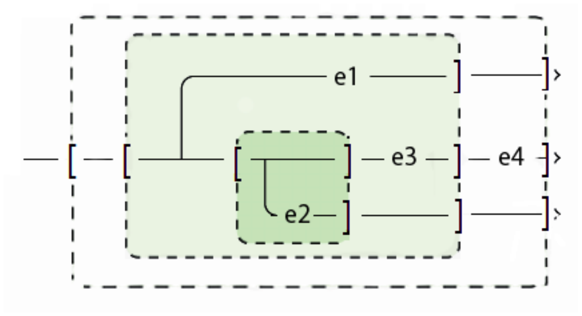

**Fig. 1.** Threads dependencies and join commits.

Suppose that `e1` represents a single transaction with no child transaction, `e2` represents a transaction with two nested children transactions (`e2=onacid; onacid;commit;commit`), `e3` stands for a transaction with three nested children transactions inside, and `e4` does for one with four nested transactions inside. So, the maximum resource consumption occurs right after spawning `e2`. At that time, `e1` contributes 3 open transactions (2 from the main thread, and 1 of itself), `e2` contributes 5 open transactions (3 from the main thread, and 2 of its self), the main thread contributes 3 transactions. And the total will be $3 + 5 + 3 = 11$ transactions. During `e3` and `e4`, the number of open transactions is smaller since many transactions have been committed before that. One of the difficulties for the analysis is that it must capture those implicit synchronization points at compile time.

The language to be used is a variant of Featherweight Java extended by transactional constructs known as Transactional Featherweight Java [9]. In our previous work, we allowed more freedom in splitting the whole program term into arbitrary sub-terms and then we gave types to all these sub-terms. This approach leads to a bit complex type systems as we have to handle quite lot of possible combinations. In this work we restrict the way to combine terms. We type the inner most thread first and then combine with its sibling threads (parent) and so on. For the running example program, we type the expression in line 5 first, and then combine it with the part from line 6 to 10. Then the line 4 will be typed with the part from line 5 to 10. Now the part from line 4 to 10 will be combine with the child thread in line 3. Then line 2 will be typed with the line 3 to 10, and then we can type the whole program from line 1 to 10. In this way, we have a more precise estimation of the maximum number of logs.

## 3 The Language TFJ

Transactional Featherweight Java (TFJ) is an object calculus featuring threads and imperative constructs that are needed for modeling concurrent and nested transaction systems.

$$
\begin{array}{llll}
P & ::= & \mathbf{0} \mid p(e) \mid P \parallel P & \text{processes} \\
L & ::= & \mathbf{class}\ C\{\bar{f}, \bar{M}\} & \text{class definition} \\
M & ::= & m(\bar{x})\{e\} & \text{methods} \\
v & ::= & r \mid x \mid \mathbf{null} & \text{values} \\
e & ::= & v \mid v.f \mid v.f := v \mid v.m(\bar{v}) \mid \mathbf{new}\ C() & \text{expressions} \\
 & & \mid \mathbf{let}\ \ x = e\ \mathbf{in}\ e \mid \mathbf{if}\ v\ \mathbf{then}\ e\ \mathbf{else}\ e & \\
 & & \mid \mathbf{spawn}(e) \mid \mathbf{onacid} \mid \mathbf{commit} &
\end{array}
$$

**Fig. 2.** TFJ syntax.

### 3.1 Syntax

The syntax of TFJ is given in Figure 2, and it is similar Featherweight Java (FJ) except for the first and the last line. The first line is for defining run time threads/processes and the last line is three commands for creating a thread, starting and committing a transaction. These three commands make the language multi-threaded and transactional. Other commands are of the FJ language: $L$ is class definition, $M$ is method declaration, the values of $v$ can be reference, variable, or **null**, and the term $e$ can be a value, a field access, an assignment, a method call, a object creation in the first line for $e$. The second syntax line for $e$ is for sequencing and choice. $\bar{x}$ ($\bar{v}$) is vector of $x$ ($v$). The detailed explanation of the syntax was given in [10]. In this paper, we restrict ourselves to a sub-language of TFJ in which we use only the simple form of the expression **let** for sequencing. Namely, $e_1; e_2$ can be expressed by **let** $x = e_1$ **in** $e_2$ where $x$ does not occur in $e_2$ with the assumption that no lazy evaluation applied. We allow only this form of **let** from now on.

### 3.2 Dynamic semantics

The semantics of TFJ is given by two sets of operational rules: the rules for the object semantics in Table 1, and the rules for the transactional and multi-threaded semantics in Table 2. The object semantics rules are standard and similar to Featherweight Java. We highlight here only the key points concerning

$$E, \mathbf{let}\ x = v\ \mathbf{in}\ e \rightarrow E', e[v/x] \quad \text{L-RED}$$

$$E, \mathbf{let}\ x_2 = (\mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e)\ \mathbf{in}\ e' \rightarrow E, \mathbf{let}\ x_1 = e_1\ \mathbf{in}\ (\mathbf{let}\ x_2 = e\ \mathbf{in}\ e') \quad \text{L-LET}$$

$$E, \mathbf{let}\ x = (\mathbf{if\ true\ then}\ e_1\ \mathbf{else}\ e_2)\ \mathbf{in}\ e \rightarrow E, \mathbf{let}\ x = e_1\ \mathbf{in}\ e \quad \text{L-COND}_1$$

$$E, \mathbf{let}\ x = (\mathbf{if\ false\ then}\ e_1\ \mathbf{else}\ e_2)\ \mathbf{in}\ e \rightarrow E, \mathbf{let}\ x = e_2\ \mathbf{in}\ e \quad \text{L-COND}_2$$

$$\frac{read(E, r) = E', C(\bar{u}) \quad fields(C) = \bar{f}}{E, \mathbf{let}\ x = r.f_i\ \mathbf{in}\ e \rightarrow E', \mathbf{let}\ x = u_i\ \mathbf{in}\ e} \ \text{L-LOOKUP}$$

$$\frac{read(E, r) = E', C(\bar{u}) \quad write(r \rightarrow C(\bar{u}) \downarrow_i^{u'}, E') = E''}{E, \mathbf{let}\ x = r.f_i := u'\ \mathbf{in}\ e \rightarrow E'', \mathbf{let}\ x = u'\ \mathbf{in}\ e} \ \text{L-UPD}$$

$$\frac{read(E, r) = E', C(\bar{u}) \quad mbody(C, m) = (\bar{x}, e)}{E, \mathbf{let}\ x = r.m(\bar{r})\ \mathbf{in}\ e' \rightarrow E', \mathbf{let}\ x = e[\bar{r}/\bar{x}, r/this]\ \mathbf{in}\ e'} \ \text{L-CALL}$$

$$\frac{r\ fresh \qquad extend(r \mapsto C(\overline{\mathbf{null}}), E) = E'}{E, \mathbf{let}\ x = \mathbf{new}\ C()\ \mathbf{in}\ e \rightarrow E', \mathbf{let}\ x = r\ \mathbf{in}\ e} \ \text{L-NEW}$$

**Table 1.** Object semantics.

creating and removing logs of the transactional and multi-threaded semantics rules. The readers are referred to [10] for a detailed explanation of the semantics.

Generally, the (global) runtime environment is a collection of threads. Each thread contains a local environment which is a sequence of logs. For managing these threads and logs we give an id to each thread and log. The object semantics mainly works on the local environment and does not manipulate logs as well as threads. This part is similar to Featherweight Java. The transactional and multi-threaded semantics creates and destroys threads and logs.

Now we formally define the local and global environments and explain in more details about the transactional and multi-threaded semantics rules.

**Definition 1 (Local environment).** *A local environment $E$ is a finite sequence of labeled logs $l_1{:}log_1; \ldots; l_k{:}log_k$, where the ith element of the sequence consists of a transaction label $l_i$ (a transaction name) and the* log $log_i$ *corresponding to the transaction $l_i$.*

For $E = l_1{:}log_1; \ldots; l_k{:}log_k$, we call $k$ the size of $E$, and denote by $|E|$. The size $|E|$ is the nesting depth of transactions, namely $l_1$ is the outer-most transaction and $l_k$ is the inner-most one. Note that the inner-most one exists for each thread. The empty environment is the empty sequence of labeled logs, and is denoted by $\epsilon$.

**Transactional and multi-threaded semantics** We will give the transition rules for the semantics in the form $\Gamma, P \Rightarrow \Gamma', P'$ or $\Gamma, P \Rightarrow error$, where $\Gamma$ is a global environment and $P$ is a set of processes of the form $p(e)$. A global environment contains local environments of threads and is defined as follows.

**Definition 2 (Global environment).** *A global environment $\Gamma$ is a finite mapping from thread id to its local environment, $\Gamma = p_1{:}E_1, \ldots, p_k{:}E_k$, where $p_i$ is a thread id and $E_i$ is the local environment of the thread $p_i$.*

We denote by $|\Gamma|$ the size of $\Gamma$ which is the total number of logs in $\Gamma$, i.e., $|\Gamma| = \sum_{i=1}^{k} |E_i|$. At the staring point, the global environment for the main thread $p_1$ of the program is $p_1 : \epsilon$. During the execution of the program, the global environment changes according to the semantic rules. Our goal is to effectively find an upper bound of $|\Gamma|$ as it represents the number of logs in all (concurrent) threads. Note that testing all possible paths is not feasible because of parallel threads and the choice expression.

The global steps make use of a number of functions accessing and changing the global environment: $reflect(p_i, E'_i, \Gamma)$, $spawn(p, p', \Gamma)$, $start(l, p_i, \Gamma)$, $intranse(\Gamma, l)$, and $commit(\bar{p}, \bar{E}, \Gamma)$. These functions were defined in [10]. Their brief explanations are as follows.

- In rule G-PLAIN, the function *reflect* only updates the local changes to the global environment.
- In rule G-SPAWN, the function $spawn(p, p', \Gamma)$ creates a new thread $p'$ with a cloned copy of transactions in $p$, so the number of logs in $p$ is duplicated.
- In rule G-TRANS, the function $start(l, p_i, \Gamma)$ creates one more log with the fresh label $l$ in thread $p_i$.
- In rule G-COMM, we denote $\coprod_1^k p_i(e_i)$ for $p_1(e_1) \parallel \ldots \parallel p_k(e_k)$. The rule requires $k$ threads to do joint commit, and each thread releases one log, so $k$ logs are to be removed as expressed in the function *commit*. $k$ threads are all threads that are synchronized by the joint commits. They contain one parent thread and $k-1$ child threads that were directly spawned by the parent. The function *intranse* identifies the $k$ threads with the same label $l$ for synchronization.

$$\frac{E, e \rightarrow E', e' \quad p : E \in \Gamma \quad reflect(p, E', \Gamma) = \Gamma'}{\Gamma, P \parallel p(e) \Rightarrow \Gamma', P \parallel p(e')} \text{ G-PLAIN}$$

$$\frac{p' \ fresh \quad spawn(p, p', \Gamma) = \Gamma'}{\Gamma, P \parallel p(\mathbf{let}\ x = \mathbf{spawn}(e_1)\ \mathbf{in}\ e_2) \Rightarrow \Gamma', P \parallel p(\mathbf{let}\ x = \mathbf{null}\ \mathbf{in}\ e_2) \parallel p'(e_1)} \text{ G-SPAWN}$$

$$\frac{l \ fresh \quad start(l, p, \Gamma) = \Gamma'}{\Gamma, P \parallel p(\mathbf{let}\ x = \mathbf{onacid}\ \mathbf{in}\ e) \Rightarrow \Gamma', P \parallel p(\mathbf{let}\ x = \mathbf{null}\ \mathbf{in}\ e)} \text{ G-TRANS}$$

$$\frac{p : E \in \Gamma \quad E = ..; l : log; \quad intranse(\Gamma, l) \ = \bar{p} \ = p_1..p_k \quad commit(\bar{p}, \bar{E}, \Gamma) = \Gamma'}{\Gamma, P \parallel \coprod_1^k p_i(\mathbf{let}\ x = \mathbf{commit}\ \mathbf{in}\ e_i) \Rightarrow \Gamma', P \parallel (\coprod_1^k p_i(\mathbf{let}\ x = \mathbf{null}\ \mathbf{in}\ e_i))} \text{ G-COMM}$$

$$\frac{\Gamma = \Gamma''; p : E \quad |E| = 0}{\Gamma, P \parallel p(\mathbf{let}\ x = \mathbf{commit}\ \mathbf{in}\ e) \Rightarrow error} \text{ G-ERROR}$$

**Table 2.** Transactional and threading semantics.

Note that a global environment may contain threads with their own local environments, and each local environment in turn contains transactions with their own logs. Therefore, a global environment may contain transactions with the same labels because some transactions are copied by a **spawn** operation.

## 4 Type System

The purpose of our type system is to determine the maximal number of logs that can be created for a TFJ program. The type of a term in our system is computed from what we call sequences of *tagged* numbers, which are abstract representation of the term's transactional behaviour.

### 4.1 Types

To represent the transactional behaviour of a term, we use a set of four symbols (called *tags* or *signs*) $\{+, -, \neg, \sharp\}$. The tags $+$ and $-$ abstractly represent the starting of a transaction and the committing of a transaction, respectively. The tag $\neg$ is used for the joint commit of transactions in parallel threads and the last one, $\sharp$, is used for accumulating the maximum number of logs created. To make it more convenient for computing on these sequences later, we associate a tag with a non-negative natural number $n \in \mathbb{N}^+ = \{0, 1, 2, ..\}$ to form *tagged numbers*. So our types use finite sequences over the set of tagged numbers ${}^T\mathbb{N} = \{ {}^+n\, , {}^-n\, , {}^\sharp n\, , {}^\neg n \mid n \in \mathbb{N}^+\}$. We will try to give rules to associate a sequence of tagged numbers with a term (expression) of TFJ.

During computation, a tag with zero may be produced but it has no effect to the semantics of the sequence so we will automatically discard it when it appears. To simplify the presentation [3], we also automatically insert ${}^\sharp 0$ element whenever needed. In our type inference implementation we do not need to insert these elements. Intuitively, for a term to type, ${}^+n$ $({}^-n)$ means there are $n$ consecutive **onacid** (**commit**) in the term, and ${}^\neg n$ means that there are $n$ threads needed to be synchronized with some **onacid** in a joint commit to complete the transaction in the term, and ${}^\sharp n$ says $n$ is the maximal number of logs created by the term.

In the sequel, let $s$ range over ${}^T\mathbb{N}$, ${}^T\bar{\mathbb{N}}$ be the set of all sequences of tagged numbers, and $S$ range over ${}^T\bar{\mathbb{N}}$ and $m, n, l, ..$ range over $\mathbb{N}$. The empty sequence is denoted by $\epsilon$ as usual. For a sequence $S$ we denote by $|S|$ the length of $S$, and write $S(i)$ for the $i$th element of $S$. For a tagged number $s$, we denote $\text{tag}(s)$ the tag of $s$, and $|s|$ the natural number of $s$ (i.e., $s = {}^{\text{tag}(s)}|s|$). For a sequence $S \in {}^T\bar{\mathbb{N}}$, we write $\text{tag}(S)$ for the sequence of the tags of the elements of $S$, i.e., $\text{tag}(s_1 \dots s_k) = \text{tag}(s_1) \dots \text{tag}(s_k)$ and $\{S\}$ for the set of tags appeared in $S$. We also write $\text{tag}(s) \in S$ instead of $\text{tag}(s) \in \{S\}$ for the simplicity.

The set ${}^T\bar{\mathbb{N}}$ can be partitioned into equivalence classes such that all elements in the same class represent the same transactional behaviour, and for each class we use the most compact sequence as the representative for the class and we call it canonical element.

**Definition 3 (Canonical sequence).** *A sequence $S$ is canonical if* $\text{tag}(S)$ *does not contain '$++$', '$--$', '$\sharp\sharp$', '$+-$', '$+\sharp-$', '$+\neg$' or '$+\sharp\neg$' as subsequences, and furthermore, $|s| > 0$ for all element $s$ of $S$.*

[3] We can avoid the insertion as shown in our implementation for the type inference algorithm in Section 6.

The intuition here is that we can always simplify/shorten a sequence $S$ without changing the interpretation of the sequence w.r.t. the resource consumption to make it canonical: simply all the tagged zero can be removed without any effect to the behavior of the term, and double tags can be converted to single tags by the following seq function. The seq function below is to reduce a sequence in ${}^{T}\bar{\mathbb{N}}$ to a canonical one. Note the pattern $+-$ does not appear, but we can insert ${}^{\sharp}0$ for ${}^{\sharp}l$ in the last definition of seq to handle this case. The last two patterns, '$+\neg$' and '$+\sharp\neg$', will be handled by the function jc later (Definition 8).

**Definition 4 (Simplify).** *Function* seq *is defined recursively as follows:*

$$\begin{aligned}
\mathrm{seq}(S) &= S \textit{ when } S \textit{ is canonical}\\
\mathrm{seq}(S\,{}^{\sharp}m\ {}^{\sharp}n\,S') &= \mathrm{seq}(S\,{}^{\sharp}\max(m,n)\,S')\\
\mathrm{seq}(S\,{}^{+}m\ {}^{+}n\,S') &= \mathrm{seq}(S\,{}^{+}(m+n)\,S')\\
\mathrm{seq}(S\,{}^{-}m\ {}^{-}n\,S') &= \mathrm{seq}(S\,{}^{-}(m+n)\,S')\\
\mathrm{seq}(S\,{}^{+}m\ {}^{\sharp}l\ {}^{-}n\,S') &= \mathrm{seq}(S\,{}^{+}(m-1)\ {}^{\sharp}(l+1)\ {}^{-}(n-1)\,S')
\end{aligned}$$

As illustrated by Figure 1, threads are synchronized by joint commits (dotted rectangles). So these joint commits split a thread into so-called *segment*s and only some segments can run in parallel. For instance, in the running example of the paper `e1` can run in parallel with `e2` and `e3`, but not with `e4`. With the type given to an expression $e$, segments can be identified by examining the type of the expression $e$ inside **spawn**$(e)$ for extra $-$ or $\neg$. For example, in **spawn**$(e_1); e_2$, if the canonical sequence of $e_1$ has $-$ or $\neg$, then the thread of $e_1$ must be synchronized with its parent which is the thread of $e_2$. Function merge in Definition 6 is used in these situations, but to define it we need some auxiliary functions.

For $S \in {}^{T}\bar{\mathbb{N}}$ and for a tag $sig \in \{+,-,\neg,\sharp\}$, we introduce the function $first(S, sig)$ that returns the smallest index $i$ such that $\mathrm{tag}(S(i)) = sig$. If no such element exists, the function returns 0. A commit can be a local commit or, implicitly, a joint commit. At first, we presume all commits be a local commit. Then when we discover that there is no local transaction starting command (i.e., **onacid**) to match with a local commit, that commit should be a joint commit. The following function performs that job and converts a canonical sequence (with no $+$ element) to a so-called *joint sequence.*

**Definition 5 (Join).** *Let $S = s_1 \ldots s_k$ be a canonical sequence and assume $i = first(S, -)$. Then function* join$(S)$ *recursively replaces $-$ in $S$ by $\neg$ as follows:*

$$\begin{aligned}
\mathrm{join}(S) &= S && \textit{if } i = 0\\
\mathrm{join}(S) &= s_1..s_{i-1}\ {}^{\neg}1\ \ \mathrm{join}(\,{}^{-}(|s_i|-1)\ \ s_{i+1}..s_k) && \textit{otherwise}
\end{aligned}$$

Since the function join is idempotent, joint sequences are well-defined and do not contain elements with $+$ or $-$ tags. We also simplify it to its canonical form so we can assume that joint sequence contains only $\sharp$ elements interleaved with $\neg$ elements. A joint sequence is used to type a term inside a **spawn** or a term in the main thread. Now we can define the merge function.

**Definition 6 (Merge).** *Let $S_1$ and $S_2$ be joint sequences such that the number of $\neg$ elements in $S_1$ and $S_2$ are the same (can be zero). The* merge *function is defined recursively as:*

$$\begin{aligned} \mathrm{merge}(\,^{\sharp}m_1\,,\,^{\sharp}m_2\,) &= \,^{\sharp}(m_1+m_2) \\ \mathrm{merge}(\,^{\sharp}m_1\ ^{\neg}n_1\ \ S_1',\,^{\sharp}m_2\ ^{\neg}n_2\ \ S_2') &= \,^{\sharp}(m_1+m_2)\ ^{\neg}(n_1+n_2)\ \mathrm{merge}(S_1',S_2') \end{aligned}$$

The definition is well-formed, because joint sequences $S_1$ and $S_2$ have only $\sharp$ and $\neg$ elements. In addition, the number of $\neg$ are the same in the assumption of the definition. So we can insert $^{\sharp}0$ to make the two sequences match over the defined patterns.

For the conditionals **if** $v$ **then** $e_1$ **else** $e_2$, we require that the external transactional behaviours of $e_1$ and $e_2$ are the same, i.e., when removing all the elements with the tag $\sharp$ from them, the remaining sequences are identical. Let $S_1$ and $S_2$ be such two sequences. Then, they can always be written as $S_i = \,^{\sharp}m_i\ ^{*}n\ S_i'$, $i = 1,2$, $* = \{+,-,\neg\}$, where $S_1'$ and $S_2'$ in turn have the same transactional behaviours. On this condition for $S_1$ and $S_2$, we define the choice operator as follows.

**Definition 7 (Choice).** *Let $S_1$ and $S_2$ be two sequences such that if removing all $\sharp$ elements from them the remaining two sequence are identical. The* alt *function is recursively defined as:*

$$\begin{aligned} \mathrm{alt}(\,^{\sharp}m_1\,,\,^{\sharp}m_2\,) &= \,^{\sharp}\mathrm{max}(m_1,m_2) \\ \mathrm{alt}(\,^{\sharp}m_1\ ^{*}n\ S_1',\,^{\sharp}m_2\ ^{*}n\ S_2') &= \,^{\sharp}\mathrm{max}(m_1,m_2)\,^{*}n\ \ \mathrm{alt}(S_1',S_2') \end{aligned}$$

## 4.2 Typing Rules

Now we are ready to introduce our formal typing rules. The language of types $T$ is defined by the following syntax:

$$T = S \mid S^{\rho}$$

The second kind of types, $S^{\rho}$, is used for terms inside a `spawn` expression which need to be synchronized with their parent thread. The treatment of two cases is different, so we denote $kind(T)$ the kind of $T$, which can be empty (normal) or $\rho$ depending on which case $T$ is. The type environment encodes the transaction context for the expression being typed. The typing judgement is of the form:

$$n \vdash e : T$$

where $n \in \mathbb{N}$ is the type environment. When $n$ is positive, it represents the number of opening transactions that $e$ will close, by commits or joint commits in $e$.

$$\frac{}{-1 \vdash \textbf{onacid} : {}^{+}1} \text{ T-ONACID}$$

$$\frac{}{1 \vdash \textbf{commit} : {}^{-}1} \text{ T-COMMIT}$$

$$\frac{n \vdash e : S}{n \vdash \textbf{spawn}(e) : (\mathrm{join}(S))^{\rho}} \text{ T-SPAWN}$$

$$\frac{n \vdash e : S}{n \vdash e : \mathrm{join}(S)^{\rho}} \text{ T-PREP}$$

$$\frac{n_1 \vdash e_1 : S_1 \quad n_2 \vdash e_2 : S_2 \quad S = \mathrm{seq}(S_1 S_2)}{n_1 + n_2 \vdash \textbf{let } x = e_1 \textbf{ in } e_2 : S} \text{ T-SEQ}$$

$$\frac{n_1 \vdash e_1 : S_1 \quad n_2 \vdash e_2 : S_2^{\rho} \quad S = \mathrm{jc}(S_1, S_2)}{n_1 + n_2 \vdash \textbf{let } x = e_1 \textbf{ in } e_2 : S} \text{ T-JC}$$

$$\frac{n \vdash e_1 : S_1^{\rho} \quad n \vdash e_2 : S_2^{\rho} \quad S = \mathrm{merge}(S_1, S_2)}{n \vdash \textbf{let } x = e_1 \textbf{ in } e_2 : S^{\rho}} \text{ T-MERGE}$$

$$\frac{n \vdash e_i : T_i \quad i = 1, 2 \quad kind(T_1) = kind(T_2) \quad T_i = S_i^{kind(T_i)}}{n \vdash \textbf{if } v \textbf{ then } e_1 \textbf{ else } e_2 : \mathrm{alt}(S_1, S_2)^{kind(S_1)}} \text{ T-COND}$$

$$\frac{mbody(m) = e \quad n \vdash e : T}{n \vdash v.m(\bar{v}) : T} \text{ T-CALL}$$

$$\frac{e \in \{v, v.f, v.f = v', newC()\} \quad n \in \mathbb{N}}{n \vdash e : \emptyset} \text{ T-SKIP}$$

**Table 3.** Typing rules.

The typing rules for our calculus are shown in Table 3. We assume that in these rules the functions $\mathrm{seq}, \mathrm{jc}, \mathrm{merge}, \mathrm{alt}$ are applicable, i.e., their arguments satisfy the conditions of the functions. The rule T-SPAWN converts $S$ to the joint sequence and marks the new type by $\rho$ so that we can merge with its parent in T-MERGE. The rule T-PREP allows us to make a matching type for the $e_2$ in T-MERGE. In T-CALL we assume that the auxiliary function *mbody* returns the body of method $m$. For sequencing (`let`), we have three rules: T-SEQ, T-MERGE and T-JC, where our previous work [14] has only two. Here we simplify the typing by allowing only a certain combinations of sequencing. This increases the preciseness of the type system as shown by the example at the end of this section. The remaining rules are straightforward except for the rule T-JC in which we need the new function jc. This function is explained below.

In rule T-JC, $e_2$ may have several segments, and let $l$ be the number of join commit threads. The last $+$ element in $S_1$, say ${}^{+}n$, will be matched with the first $\neg$ element in $S_2$, say ${}^{\neg}l$. But after ${}^{+}n$, there can be a $\sharp$ element, say ${}^{\sharp}n'$, and the local maximal number of logs for ${}^{+}n\ {}^{\sharp}n'$ is $n+n'$ (but we will define step-by-step so in the following definition of jc we only add 1 to $n'$ at a time). Similarly, before ${}^{\neg}l$ there can be a ${}^{\sharp}l'$, so the maximum of log at this point is at least $l+l'$. After removing one $+$ from $S_1$ and one $\neg$ from $S_2$ we can simplify the new sequences so that the patterns can appear in the next recursive call of jc. Thus, the function jc is defined as follows. Note that we do not define the function for all patterns and this is a reason for the loss of compositionality.

**Definition 8 (Joint commit).** *Function* jc *is defined recursively as follows:*

$$\mathrm{jc}(S_1'\ {}^{+}n\ {}^{\sharp}n',\ {}^{\sharp}l'\ {}^{\neg}l\ S_2') = \mathrm{jc}(\mathrm{seq}(S_1'\ {}^{+}(n-1)\ {}^{\sharp}(n'+1)), \mathrm{seq}({}^{\sharp}(l'+l)\ S_2'))\ \textit{if}\ l,n>0$$
$$\mathrm{jc}({}^{\sharp}n',\ {}^{\sharp}l'\ {}^{\neg}l\ S_2') = \mathrm{seq}({}^{\sharp}\mathrm{max}(n',l')\ {}^{\neg}l\ S_2')$$

In the definition of jc, we implicitly assume that the first definition will be applied if there exists ${}^{\sharp}n$ with $n>0$. As we can see in Section 6, the type inference algorithm naturally satisfies this condition.

As our type reflects the behaviour of a term, so the type of a well-typed program contains only a sequence of single $\sharp$ element expressing the upper bound of logs that can be created when we execute the program and the typing environment is 0.

**Definition 9 (Well-typed).** *A term $e$ is* well-typed *if there exists a type derivation for $e$ such that $0 \vdash e : {}^{\sharp}n$ for some $n$.*

A typing judgment has a property that its environment has just enough opening transactions for the (join) commits in $e$ as expressed by $T$. Due to the lack of space, the proof of the theorem is skipped here. You can find it in a complete version of this paper.

**Theorem 1 (Type judgment property).** *If $n \vdash e : T$ and $n \geq 0$ then* $\mathrm{sim}({}^{+}n, T) = {}^{\sharp}m$ *and $m \geq n$ where* $\mathrm{sim}(T_1,T_2) = \mathrm{seq}(\mathrm{jc}(S_1,S_2))$ *with $S_i$ is $T_i$ without $\rho$.*

*Proof (sketch).* By induction on the typing rules in Table 3.

- The case T-ONACID does not apply as $n<0$.
- The case T-COMMIT is trivial since $\mathrm{sim}({}^{+}1\ {}^{\neg}1) = {}^{\sharp}1$ so $m=1=n$.
- For T-SEQ, by induction hypotheses (IH) we have $\mathrm{seq}(\mathrm{jc}({}^{+}n_i, S_i)) = \mathrm{seq}({}^{+}n_i\ S_i) = {}^{\sharp}m_i$ for $i=1,2$ since $S_i$ have no $\neg$ elements. We need to prove that $\mathrm{sim}({}^{+}(n_1+n_2), S) = {}^{\sharp}m$ and $m \geq m_1+m_2$. We have:

$$\begin{aligned}
\mathrm{sim}({}^{+}(n_1+n_2), S) &= \mathrm{seq}(\mathrm{jc}({}^{+}(n_1+n_2), \mathrm{seq}(S_1S_2))) && \\
&= \mathrm{seq}({}^{+}n_2\ {}^{+}n_1\ S_1S_2) && \neg \notin S_1S_2 \\
&= \mathrm{seq}({}^{+}n_2\ ({}^{+}n_1\ S_1)S_2) && \text{Def. 4} \\
&= \mathrm{seq}({}^{+}n_2\ {}^{\sharp}m_1\ S_2)) && \text{IH} \\
&= {}^{\sharp}(m_1+m_2) && \text{IH, Def. 4}
\end{aligned}$$

so the lemma holds.

– For T-JC, by induction hypotheses we have $\text{seq}({}^{+}n_1\, S_1) = {}^{\sharp}m_1$ and $\text{seq}(\text{jc}({}^{+}n_2\,, S_2)) = {}^{\sharp}m_2$ . Similarly to the previous case, we have:

$$\begin{aligned}
\text{sim}({}^{+}(n_1+n_2))\,, S) &= \text{seq}(\text{jc}({}^{+}(n_1+n_2))\,, \text{jc}(S_1, S_2))) && S = \text{jc}(S_1, S_2)\\
&= \text{seq}(\text{jc}({}^{+}(n_1+n_2)\, S_1, S_2))) && \text{jc definition}\\
&= \text{seq}(\text{jc}({}^{+}n_2\; {}^{\sharp}m_1\,, S_2))) && \text{IH}\\
&= {}^{\sharp}\text{max}(n_2+m_1, m_2) && \text{jc definition, IH}\\
&\geq {}^{\sharp}\text{max}(n_2+n_1) && m_1 \geq n_1 \text{ by IH}
\end{aligned}$$

– For T-MERGE, by induction hypotheses we have $\text{seq}(\text{jc}({}^{+}n_1\,, S_1) = {}^{\sharp}m_1$ and $\text{seq}(\text{jc}({}^{+}n_2\, S_2) = {}^{\sharp}m_2$ . Similarly to the previous case, we have:

$$\begin{aligned}
\text{seq}(\text{jc}({}^{+}n\,, S)) &= \text{seq}(\text{jc}({}^{+}n\,, \text{merge}(S_1, S_2))) && S = \text{merge}(S_1, S_2)\\
&= \text{seq}(\text{jc}({}^{+}n\,, S_1)) + \text{seq}(\text{jc}({}^{+}n\,, S_2)) && \text{properties of } S_1, S_2\\
&= {}^{\sharp}(m_1+m_2) && \text{IH}
\end{aligned}$$

– For the remaining rules, the lemma holds by the induction hypotheses.

**Typing the running example program** Let us try to make a type derivation for the program in Listing 1.1. We denote $e^l_m$ for the part of the program from line $l$ to line $m$. First, using T-SEQ, T-ONACID, T-COMMIT we have that the expression inside **spawn** in line 5 have the type ${}^{\sharp}2\ {}^{\neg}1\ {}^{\neg}1\ {}^{\neg}1$ . Then, by applying the rule T-SPAWN, we have:

$$3 \vdash e^5_5 : (\,{}^{\sharp}2\ {}^{\neg}1\ {}^{\neg}1\ {}^{\neg}1\,)^{\rho} \tag{1}$$

Now, we can use T-MERGE and we need an expression such that its type matches ${}^{\sharp}2\ {}^{\neg}1\ {}^{\neg}1\ {}^{\neg}1$ . We find that $e^6_{10}$ satisfies this condition since its type is derived using the rules T-SEQ, T-ONACID, T-COMMIT as:

$$3 \vdash e^6_{10} :\ {}^{\neg}1\ {}^{\sharp}3\ {}^{\neg}1\ {}^{\sharp}4\ {}^{\neg}1$$

By applying T-PREP, we have a matching type with (1). So we can apply T-MERGE and get:

$$3 \vdash e^5_{10} : (\,{}^{\sharp}2\ {}^{\neg}2\ {}^{\sharp}3\ {}^{\neg}2\ {}^{\sharp}4\ {}^{\neg}2\,)^{\rho}$$

Now we apply T-JC to get the type for $e^4_{10}$:

$$2 \vdash e^4_{10} :\ {}^{\sharp}4\ {}^{\neg}2\ {}^{\sharp}4\ {}^{\neg}2 \tag{2}$$

since $\text{jc}({}^{+}1\,, {}^{\sharp}2\ {}^{\neg}2\ {}^{\sharp}3\ {}^{\neg}2\ {}^{\sharp}4\ {}^{\neg}2\,) = \text{jc}({}^{\sharp}1\,, \text{seq}({}^{\sharp}4\ {}^{\sharp}3\ {}^{\neg}2\ {}^{\sharp}4\ {}^{\neg}2\,)) = \text{jc}({}^{\sharp}1\,, {}^{\sharp}4\ {}^{\neg}2\ {}^{\sharp}4\ {}^{\neg}2\,) = \text{seq}({}^{\sharp}1\ {}^{\sharp}4\ {}^{\neg}2\ {}^{\sharp}4\ {}^{\neg}2\,) = {}^{\sharp}4\ {}^{\neg}2\ {}^{\sharp}4\ {}^{\neg}2$ .

In the same way, we can calculate the type for the expression in line 3 which is $(\,{}^{\sharp}1\ {}^{\neg}1\ {}^{\neg}1\,)^{\rho}$. This type matches (2) so we can apply T-MERGE and get the type for $e^3_{10}$:

$$2 \vdash e^3_{10} :\ {}^{\sharp}5\ {}^{\neg}3\ {}^{\sharp}4\ {}^{\neg}3 \tag{3}$$

Applying T-JC for $-2 \vdash e_2^1 : {}^{+}2$ and (3) we get:

$$0 \vdash e_{10}^1 : {}^{\sharp}11$$

since $\mathrm{jc}(\,{}^{+}2\,,\ {}^{\sharp}5\ {}^{\neg}3\ {}^{\sharp}4\ {}^{\neg}3\,) = \mathrm{jc}(\,{}^{+}1\ {}^{\sharp}1\,,\ {}^{\sharp}8\ {}^{\neg}3\,) = \mathrm{jc}(\,{}^{\sharp}2\,,\ {}^{\sharp}11\,) = \mathrm{seq}(\,{}^{\sharp}2\ {}^{\sharp}11\,) = \ {}^{\sharp}11\,.$

The program is well-typed and the maximum number of logs that coexist during the execution of the program is 11.

**Example to show sharper bound** Our type system is sharper than the type system in our previous work [14] as demonstrated by the program in Listing 1.2. The expression in lines 4-5 has the type $(\,{}^{\neg}2\,)^{\rho}$. The one in lines 1-3 has the type ${}^{+}1\ {}^{\sharp}1$. Applying the rule T-JC we get $\mathrm{jc}(\,{}^{+}1\ {}^{\sharp}1\,,\ {}^{\neg}2\,) = \ {}^{\sharp}\mathrm{max}(1+1, 0+2) \ = \ {}^{\sharp}2\,.$ Our previous type system will return the bound of ${}^{\sharp}3$, which is less precise than this bound.

**Listing 1.2.** An example program to show sharper bound.

```
1  onacid;
2  onacid;
3  commit;
4  spawn(commit);
5  commit
```

The type derivation for the program is as follows, where the names of typing rules is simplified to the 3th and 4th characters, [ and ] are for **onacid** and **commit**, respectively.

$$\dfrac{\dfrac{-1 \vdash [:\ {}^{+}1 \qquad \dfrac{-1 \vdash [:\ {}^{+}1 \quad +1 \vdash ]:\ {}^{\neg}1}{0 \vdash [;]:\ {}^{\sharp}1}\,\text{SE}}{-1 \vdash [;[;]:\ {}^{+}1\ {}^{\sharp}1}\,\text{SE} \qquad \dfrac{\dfrac{+1 \vdash ]:\ {}^{\neg}1}{+1 \vdash \textbf{spawn}(]):\ {}^{\neg}1^{\,\rho}}\,\text{SP} \quad \dfrac{+1 \vdash ]:\ {}^{\neg}1}{+1 \vdash ]:\ {}^{\neg}1^{\,\rho}}\,\text{PR}}{+1 \vdash \textbf{spawn}(]);]:\ {}^{\neg}2^{\,\rho}}\,\text{ME}}{0 \vdash [;[;];\textbf{spawn}(]);]:\ {}^{\sharp}2}\,\text{JC}$$

## 5 Correctness

To show that our type system meets our purpose mentioned in the introduction of this paper, we need to show that a well-typed program does not create more logs than the amount expressed in its type. Let our well-typed program be $e$ and its type is ${}^{\sharp}n$. We need to show that when executing $e$ according to the semantics in Section 3, the number of logs in the global environment is always smaller than $n$.

Recall, a state of a program is a pair $\Gamma, P$ where $\Gamma = p_1 : E_1; \ldots ; p_k : E_k$ and $P = \coprod_1^k p_i(e_i)$. We say $\Gamma$ satisfies $P$, notation $\Gamma \models P$, if there exist $S_1, \ldots , S_k$ such that $|E_i| \vdash e_i : S_i$ for all $i = 1, \ldots , k$. For a component $i$, $E_i$ represents the logs that have been created or copied in thread $p_i$, and $S_i$ represents the number of logs that will be created when executing $e_i$. Therefore, the behavior of thread $p_i$ in term of logs is expressed by $\text{sim}(\,^{+}|E_i|\,, S_i)$, where the sim function is defined in Theorem 1. We will show that $\text{sim}(\,^{+}|E_i|\,, S) = \,^{\sharp}n$ for some $n$. We denote this value $n$ as $|E_i, S_i|$. Then *the total number of logs* of a program state – ones in $\Gamma$ and the potential ones that will be created when the remaining program is executed – denoted by $|\Gamma, P|$, is defined by:

$$|\Gamma, P| = \sum_{i=1}^{k} |E_i, S_i|$$

Since $|\Gamma, P|$ represents the maximum number of logs *from* the current state and $|\Gamma|$ is the number of logs *in* the current state, we have the following properties.

**Lemma 1.** *If $\Gamma \models P$ then $|\Gamma, P| \geq |\Gamma|$.*

*Proof (sketch).* By the definition of $|\Gamma, P|$ and $|\Gamma|$, we only need to show $|E_i, S_i| \geq |E_i|$ for all $i$. This is followed by Theorem 1.

**Lemma 2 (Subject reduction 1).** *If $E, e \rightarrow E', e'$, and $|E| \vdash e : S$ then there exists $S'$ such that $|E'| \vdash e' : S'$ and $|E, S| \geq |E', S'|$.*

*Proof (sketch).* The proof is done by checking directly all the semantics rules in Figure 1 and is omitted here.

**Lemma 3 (Subject reduction 2).** *If $\Gamma \models P$ and $\Gamma, P \Rightarrow \Gamma', P'$ then $\Gamma' \models P'$ and $|\Gamma, P| \geq |\Gamma', P'|$.*

*Proof (sketch).* The proof is done by checking one by one all the semantics rules in Figure 2. For each rule, we need to prove two parts: (i) $\Gamma' \models P'$ and (ii) $|\Gamma, P| \geq |\Gamma', P'|$.

- For the rule G-SPAWN, by the assumption $\Gamma \models P$ and the definition of $\models$ we have $|E| \vdash \textbf{spawn}(e_1); e_2 : S$ and $|E, S| = n$ for some $E, S, n$. By the rule G-SPAWN, we need to prove the two new elements in $\Gamma'$ that $|E| \vdash \textbf{spawn}(e_1) : S_1$ and $|E| \vdash e_2 : S_2$ for some $S_1, S_2$, but this follows directly from T-MERGE because $\textbf{spawn}(e_1)$ can only be combined with $e_2$ by this typing rule. In addition, we have $S = \text{merge}(S_1, S_2)$.
  By the definition of merge, we have the number of $\neg$'s in $S$, $S_1$ and $S_2$ are the same, and by definition of jc, it equals $|E|$. So $|E, S_i| = n_i$ for some $n_i$ with $i = 1, 2$. So (i) holds.
  For (ii), first, by the definitions of merge and jc we get $n = n_1 + n_2$. We need to show $|\Gamma, P| - |\Gamma', P'| \geq 0$. Since $\Gamma$ and $\Gamma'$ only differ in threads of $E, E_1, E2$, we have $|\Gamma, P| - |\Gamma', P'| = |E, S| - |E, S_1| - |E, S_2| = n - n_1 - n_2 = 0$. So (ii) holds.
- For G-TRANS, similarly to the previous case, by the assumption we have $|E| \vdash \textbf{onacid}; e : S$ and $|e, S| = n$ for some $E, S, n$. The rule G-TRANS creates a new log in $E$ so we need to prove that $|E| + 1 \vdash e : S'$ for some $S'$. Since $\textbf{onacid}; e$ can be typed by T-SEQ or T-JC, we have two cases to consider.

  - By the rule T-SEQ, (i) follows immediately by the typing rule when the first component is $-1 \vdash \textbf{onacid} : +1$. For (ii), similarly to the the previous case, we have $|\Gamma, P| - |\Gamma', P'| = |\,{}^{+}|E|\,,\,{}^{+}1\,S'| - |\,{}^{+}(|E|+1)\,,S'| = 0$. So (ii) holds.
  - By the rule T-JC, (i) follows immediately by the typing rule when the first component is $-1 \vdash \textbf{onacid} : {}^{+}1$. For (ii) we have

$$\begin{aligned}
|\Gamma, P| - |\Gamma', P'| &= |\,{}^{+}|E|\,, \mathrm{jc}({}^{+}1\,, S')| - |\,{}^{+}(|E|+1)\,, S'| \\
&= \mathrm{sim}(|E|, \mathrm{jc}({}^{+}1\,, S') - \mathrm{sim}({}^{+}(|E|+1)\,, S') \\
&= \mathrm{seq}(\mathrm{jc}(|E|, \mathrm{jc}({}^{+}1\,, S')) - \mathrm{seq}(\mathrm{jc}({}^{+}(|E|+1)\,, S')) \\
&= \mathrm{seq}(\mathrm{jc}(|E|, \mathrm{jc}({}^{+}1\,, S')) - \mathrm{seq}(\mathrm{jc}({}^{+}|E|\,, \mathrm{jc}({}^{+}1\,, S'))) \\
&= 0
\end{aligned}$$

  So (ii) holds.

– For the rule G-COMM, by the assumption $\Gamma \models P$ we have $E_i \vdash \textbf{commit}; e_i : S_i$ for some $E_i, S_i$, $i = 1..k'$.
  As $\textbf{commit}; e_i$ can be typed by T-SEQ and the **commit** is a joint commit, (i) follows by the typing rule when the first expression is $+1 \vdash \textbf{commit} : {}^{\neg}1$ and we have $S_i = {}^{\neg}1\,S'_i$ where $S'_i$ is the type of $e_i$.
  For (ii) we have:

$$\begin{aligned}
&|\Gamma, P| - |\Gamma', P'| \\
&= \sum_{1}^{k'} (|E_i, {}^{\neg}1\,S'_i| - |E'_i, S'_i|) \\
&= \sum_{1}^{k'} (\mathrm{seq}(\mathrm{jc}({}^{+}|E_i|\,, {}^{\neg}1\,S'_i)) - \mathrm{seq}(\mathrm{jc}({}^{+}|E'_i|\,, S'_i)) && \text{def. of } |E, S| \\
&= \sum_{1}^{k'} (\mathrm{seq}(\mathrm{jc}({}^{+}|E_i|\,, {}^{\neg}1\,S'_i)) - \mathrm{seq}(\mathrm{jc}({}^{+}(|E_i|-1)\,, S'_i)) && |E'_i| = |E_i| - 1 \\
&= \sum_{1}^{k'} (\mathrm{seq}(\mathrm{jc}({}^{+}(|E_i|-1)\,, {}^{\sharp}1\,S'_i)) - \mathrm{seq}(\mathrm{jc}({}^{+}(|E_i|-1)\,, S'_i)) && \text{def. of jc} \\
&\geq 0
\end{aligned}$$

  So (ii) holds.

– For the remaining rules the proof is similar to the previous cases.

Now we come to the correctness property of our type system.

**Theorem 2 (Correctness).** *Suppose* $0 \vdash e : {}^{\sharp}n$ *and* $p_1 : \epsilon, p_1(e) \Rightarrow^* \Gamma, P$. *Then* $|\Gamma| < n$.

*Proof.* For the starting environment we have $|p_1 : \epsilon, p_1(e)| = \mathrm{sim}(0, {}^{\sharp}n\,) = {}^{\sharp}n$. So from Lemmas 2, 3 and Theorem 1, the theorem holds by trivial induction on the length of derivation.

## 6 Type Inference

In this section we give an algorithm to compute types for the core of our TFJ calculus. Our main type inference algorithm is presented in Listing 1.3 in the functional programming style, which uses the functions defined in the previous sections to compute types (sequence of tagged numbers). The main function `infer` takes an expression `term` and a head 'environment' `headseq` in line 6. The expression is encoded as a list of branches and leaves. A branch corresponds to a new thread. A leaf is a tagged number.

When we reach the end of `term` in line 8 we need to compact the result type by calling *seq* function. Otherwise, we check if the next command `x` is a new branch or not. If not we just update the `headseq` with the new leaf (line 12) and then repeat the inference process. In case `x` is a new branch, we type its term `br` and merge it with the remaining part `xs`. The merged type is combined with the head 'environment' to produce the final result. We have implemented the algorithm in F Sharp and tested on several examples[4]. The code contains automated tests and all test cases are passed, i.e., actual results is equal to our expected ones.

**Listing 1.3.** A type inference algorithm.

```
1  type TagNum = Tag * int
2  type Tree =  | Branch of Tree list | Leaf of TagNum
3  let rec infer (term: Tree list) (headseq: TagNum list) =
4    match term with
5    | [] -> seq headseq (* simplifies the result *)
6    | x::xs ->
7      match x with
8      | Leaf tagnum ->
9        (* expand the head part *)
10       let new_head = seq (List.append headseq [tagnum])
             in
11       infer xs new_head
12     | Branch br -> (* a new thread *)
13       (* infer the child and parent tail *)
14       let child = join (infer br []) in
15       let parent = join (infer xs []) in
16       (* merge the child and the parent tail *)
17       let tailseq = seq (merge child parent) in
18       (* join commit with the head *)
19       jc headseq tailseq
```

## 7 Conclusion

We have presented a new type system that have some advantages over our previous type systems [10,14] for a language that mixes nested transactional memory

[4] https://github.com/truonganhhoang/tfj-infer

and multi-threading. Our type system is much simpler and gives more precise estimation for the maximum number of logs (in the worst case) that can coexist during the execution of a program being typed. Though the new type system is a bit less compositional than the previous ones, we believe that the inference algorithm developed based on this work is more efficient. Like the type system in [14], the one presented in this paper does not restrict opening new transactions after a joint commit as the one presented in [10]. Our next step is to generalize our type system for the larger class of TFJ with more language features.

## References

1. Elvira Albert, Samir Genaim, and Miguel Gomez-Zamalloa. Heap space analysis for Java bytecode. In *ISMM '07*, New York, NY, USA, 2007. ACM.
2. David Aspinall, Robert Atkey, Kenneth MacKenzie, and Donald Sannella. Symbolic and analytic techniques for resource analysis of Java bytecode. In *TGC'10*, number 6084 in LNCS. Springer-Verlag, 2010.
3. Marc Bezem, Dag Hovland, and Hoang Truong. A type system for counting instances of software components. *Theor. Comput. Sci.*, 458:29–48, 2012.
4. Víctor Braberman, Diego Garbervetsky, and Sergio Yovine. A static analysis for synthesizing parametric specifications of dynamic memory consumption. *Journal of Object Technology*, 5(5), 2006.
5. Wei-Ngan Chin, Huu Hai Nguyen, Shengchao Qin, and Martin C. Rinard. Memory usage verification for OO programs. In *Proceedings of SAS '05*, volume 3672 of *LNCS*. Springer-Verlag, 2005.
6. Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of POPL '03*. ACM, January 2003.
7. Martin Hofmann and Steffen Jost. Type-based amortised heap-space analysis (for an object-oriented language). In Peter Sestoft, editor, *Proceedings of ESOP 2006*, volume 3924 of *LNCS*. Springer-Verlag, 2006.
8. John Hughes and Lars Pareto. Recursion and dynamic data-structures in bounded space: Towards embedded ML programming. *SIGPLAN Notices*, 34(9), 1999.
9. Suresh Jagannathan, Jan Vitek, Adam Welc, and Antony Hosking. A transactional object calculus. *Sci. Comput. Program.*, 57(2):164–186, August 2005.
10. Thi Mai Thuong Tran, Martin Steffen, and Hoang Truong. Compositional static analysis for implicit join synchronization in a transactional setting. In *Software Engineering and Formal Methods*, volume 8137 of *LNCS*, pages 212–228. Springer Berlin Heidelberg, 2013.
11. Tuan-Hung Pham, Anh-Hoang Truong, Ninh-Thuan Truong, and Wei-Ngan Chin. A fast algorithm to compute heap memory bounds of Java Card applets. In *Software Engineering and Formal Methods*, 2008.
12. Nir Shavit and Dan Touitou. Software transactional memory. In *Symposium on Principles of Distributed Computing*, pages 204–213, 1995.
13. Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2), 1997.
14. Xuan-Tung Vu, Thi Mai Thuong Tran, Anh-Hoang Truong, and Martin Steffen. A type system for finding upper resource bounds of multi-threaded programs with nested transactions. In *Symposium on Information and Communication Technology SoICT'12*, pages 21–30, 2012.