A type system for finding upper resource bounds of multi-threaded programs with nested transactions

Tung Vu Xuan
University of Engineering
and Technology, VNU Hanoi
toilatung90@gmail.com

Hoang Truong Anh
University of Engineering
and Technology, VNU Hanoi
truonganhhoang@gmail.com

Thi Mai Thuong Tran
Department of Informatics,
University of Oslo, Norway
tmtran@ifi.uio.no

Martin Steffen
Department of Informatics,
University of Oslo, Norway
msteffen@ifi.uio.no

Abstract

We present a static, compositional analysis based on a type and effect system to estimate an upper bound for the resource consumption for nested and multi-threaded transactional programs. This work extends our previous type system for Transactional Featherweight Java to allow more liberal use of transactions in the semantics. The new types are also more expressive and structurally simpler using a linear representation instead of a tree representation for capturing static approximation of resource consumption. We prove soundness of our analysis.

1 Introduction

Software Transactional Memory [12] has been introduced as an alternative to locked-based synchronization for shared memory concurrency. It has become the focus of intensive theoretical research and for practical applications.

One of the recent transactional models supports advanced features such as nested and multi-threaded transactions is described in [10]. In this model, a transaction is nested if it contains a number of transactions, and the child transactions must commit before their parent. Furthermore, a transaction is multi-threaded when threads are allowed to run inside the transaction and in parallel with the parent thread executing that transaction, as well. The threads spawned inside a transaction will inherit the open transactions and thus the memory usage of its parent thread. When the parent thread commits a transaction, all the child threads must join the commit of their parent. We call the commits of the child threads and their parent *joint commits*. Due to this form of synchronization, the parallel threads inside a transaction do not run independently.

In a typical implementation, each thread has its own local copy of memory called log per transaction to record memory accesses during its execution. In particular a child thread will also store a copy of its parent's log so that the child thread can be executed independently with its parents until commit time. At commit time when all child threads and their parents join via a commit, their own logs and the copies are consulted to check for conflicts and potentially perform a roll-back. A major complication for the static analysis is that the memory locations are implicitly copied into the local logs, the resources used by a transactional program are difficult to estimate.

We develop a type system to statically estimate the memory resource consumption in terms of the maximum number of logs that co-exist at the same time. This work extends our previous work [14] by removing a restriction on the semantics of the language which does not allow new transactions opened inside spawned threads after a

joint commit of their parent. Moreover the type system here is a simplification of the previous one, using a linear numeric representation instead of tree representation as in the previous work. More concretely, the type judgements now are based on what we call sequences of *tagged numbers* reflecting the resource consumption of the transactional behaviour. Below we informally describe the calculation of memory resource a transactional program could consume at runtime. Consider the following pseudo-code of a program containing nested and multi-threaded transactions:

```
1 onacid;                                              //thread 0
2     onacid;
3         spawn(e1; commit; commit);                   // thread 1
4         onacid;
5             spawn (e2; commit; commit; commit); // thread 2
6         commit;
7         e3;
8     commit;
9 commit;
```

The program is illustrated in Figure 1. The starting transaction command $onacid$ and ending transaction command $commit$ are denoted by [and], resp. The $spawn$ command creates a new thread running in parallel with the parent thread. The new thread makes a copy of local variables of the parent thread into its local environment. In our example, when spawning e_1 the main thread has opened two transactions so thread 1 executes e_1 inside these two transactions and must do two commits to close them. That is the reason why after e_1, thread 1 needs to execute two $commit$ commands. Figure 1(b) illustrates that the parallel threads must commit a transaction at the same time. The right-hand edges of the boxes mark these synchronizations.

Suppose e_1 opens and closes one transaction, e_2 two, e_3 three and e_4 four. The maximum resource consumption occurs after spawning e_2. At that time, e_1 contributes 3 transactions (2 from the main thread, and 1 of its self), e_2 contributes 6 transactions (3 from the main thread, and 3 of its self), the main thread contributes 3 transaction. And the total will be $2 + 6 + 3 = 11$ transactions.

As mentioned above parallel threads are not completely independent. They join with their parents via a commit which is an implicit synchronization point. The difficulty for the analysis is that it must capture those implicit synchronization points at compile time. Furthermore the analysis needs to contain enough information in order to analyze the resource consumption *compositionally*.

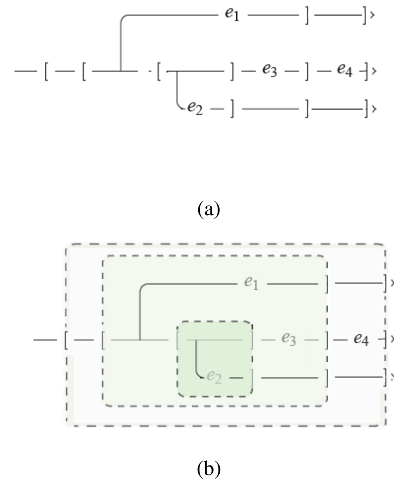

Figure 1: Nested, multi-threaded transactions and join synchronization

The rest of the paper is structured as follows. In the rest of Section 1, we discuss some related work. Section 2 introduces syntax and operational semantics of the calculus. Section 3 presents a type system for estimating the resource consumption using a linear numeric representation. The soundness of the analysis is sketched in Section 4. We conclude in Section 5.

Related work

Estimating resource usage has been studied in various settings. [9] introduces a strict, first order functional language with a type system such that well-typed programs run within the space specified by the programmer. The paper [13] uses inference system to describe a memory management for programs that perform dynamic memory allocation and de-allocation. Hofmann and Jost [7] use a linear type system to compute linear bounds on heap space for a first-order functional language. For imperative and object-oriented languages Wei-Ngan Chin et al. [5] verifies memory usages for object-oriented programs. Programmers have to annotate the memory usage and size relations for methods as well as explicit de-allocation. In [8], Hofmann and Jost use a type system to calcu-

late the heap space bound as a function of input for an object oriented language. They successfully treat inheritance, downcast, update and aliasing. In [4] the authors present an algorithm to statically compute upper bounds of the amount of memory consumption of a method as a non-linear function of method's parameters. The bounds are not precise. Their work is not type-based and the language does not include explicit de-allocation. Braberman et al. [2] calculate non-linear symbolic approximation of memory bounds for Java programs involving both data structures and loops. However the bounds are not easily precise due to various factors. [6] present a similar approach.

The authors in [3] study the use of logical methods to infer precise memory consumption of Java bytecode programs with resource annotation by pre- and post-conditions. 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 languages — but also of the complex and implicit synchronization (by joint commits) structure entailed by the transactional model.

2 Transactional Featherweight Java

Transactional Featherweight Java (TFJ) is an object calculus featuring threads and imperative constructs needed to model transactions, supporting a quite expressive transactional concurrency model.

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

Table 1: TFJ syntax

2.1 Syntax

The syntax of TFJ is shown in Table 1. We use P for process terms, e for expressions. $p(e)$ is a thread with identifier p and expression e being executed; the thread label p is distinct for every thread. Sets of processes can be empty **0** or consists of a number of processes $p(e)$ running in parallel, where parallel composition is written as $\parallel$.

The metavariable L ranges over class definitions $class\ C\{\vec{f}; \vec{M}\}$, where C is the name of the class, $\vec{f}$ presents the list of fields, $\vec{M}$ captures the list of methods. Inheritance is not supported in this language. A method definition $M ::= m(\vec{x})\{e;\}$ consists of the name m of the method, the list of parameters $\vec{x}$, the method body e which is a expression. Moreover, v stands for values which can be object references r, variables x or $null$. Values are expressions that can no longer be evaluated. Finally, an expression can be either a value v, a field access $v.f$, a field update $v.f := v$, a conditional structure $if\, v\, then\, e\, else\, e$, a sequential composition specified by let-construct $let\, x = e\ in\ e$, a method call $v.m(\vec{v})$, an object construction $new\, C()$, a thread creation $spawn\, e$, a transaction start command $onacid$ or a transaction close command $commit$. The expression $spawn\ e$ creates a new thread to evaluate e. The execution of e takes place inside the same nesting of transactions as the thread executing $spawn\ e$, i.e $spawn\ e$ will cause the current environment being copied into the new thread. For readability, we write $e_1; e_2$ to indicate sequencing of expressions e_1 and e_2.

2.2 Semantics

The semantics of TFJ is given by two-levels of operational rules; the local and the global semantics is shown in Table 2 and 3, respectively. The local semantics deals with the evaluation of *one* single *thread* and reduces configurations of the form E, e, where e is an expression and E is a *local environment.*.

Definition 1 (Local environment). *A local environment E is a finite sequence of the form $l_1{:}log_1, \ldots, l_k{:}log_k$, i.e., of pairs of transaction labels l_i and the corresponding* log *log_i. We write $|E|$ for denoting the number of pairs $l{:}log$, which is called the size of E.*

The sequences of transaction names and logs are used

$$E, let\, x = v\, in\, e \rightarrow E, e|_{x=v} \quad \text{R-RED}$$

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

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

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

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

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

$$\frac{read(E,r)=E',C(\vec{r}) \quad mbody(C,m)=(\vec{x},e)}{E, let\, x=r.m(\vec{r})\, in\, e' \rightarrow E', let\, x=e|_{\vec{x}=\vec{r},this=r}\, in\, e'} \quad \text{R-CALL}$$

Table 2: Local semantics

to represent the nesting structure. The transactions later in the sequence are executed inside the former transactions. This means that the left-most transaction refers to the inner-most transaction, the most recently started one. Consequently, committing terminates transactions from right to left and also removes the corresponding bindings in the local environment. The number $|E|$ thus specifies the nesting depth of the thread, i.e., the number of transactions which have been started but not yet committed. $|E|$ in our analysis is the current amount of allocated memory for the thread. A log_i just keeps track of changes to the local memory of a tread wrt. transaction l_i. The local level only concerns the current thread and consists of rules for the commands for reading, writing, method calls and for creating new objects.

At the global level, the reduction is of the form: $\Gamma, P \Rightarrow \Gamma', P'$ or $\Gamma, P \Rightarrow error$, where Γ is the global environment and P is a process.

Definition 2 (Global environment)**.** *A global environment Γ is a finite map, written as $p_1{:}E_1, \dots p_k{:}E_k$, from threads names p_i to local environments E_i.*

In general, each process contains a number of threads running in parallel. Each thread executes an expression whose the evaluation is described by the local rules. For each thread p, we need the corresponding local environment E. Thus, Γ is a set of bindings of the form $p{:}E$ where the order of the bindings does not play a role because the threads run in parallel. The global steps make use of a number of functions accessing and changing the global environment:

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

1. *The function reflect propagates the changes from a local environment to a global environment: if $reflect(p_i, E_i', \Gamma) = \Gamma'$ and $\Gamma = p_1{:}E_1, \dots, p_i{:}E_i, \dots, p_k{:}E_k$, then $\Gamma' = p_1{:}E_1, \dots, p_i{:}E_i', \dots, p_k{:}E_k$ with $|E_i| = |E_i'|$.*

2. *The function $spawn(p, p', \Gamma)$ creates a new thread p' from a parent thread p: Assume $\Gamma = p{:}E, \Gamma''$ and $p'{:}E' \notin \Gamma$ and $spawn(p, p', \Gamma) = \Gamma'$, then $\Gamma' = p{:}E, p'{:}E', \Gamma''$ and $|E| = |E'|$.*

3. *The function $start(l, p_i, \Gamma)$ opens a new transaction l in thread p_i: if $start(l, p_i, \Gamma) = \Gamma'$ for $\Gamma = p_1{:}E_1, \dots, p_i{:}E_i, \dots, p_k{:}E_k$ and for a fresh l, then $\Gamma' = p{:}E_1, \dots, p_i{:}E_i', \dots, p_k{:}E_k$, with $|E_i'| = |E_i| + 1$.*

4. *The function $intranse(\Gamma, l)$ returns a set of threads currently have the same transaction l: Assume $\Gamma =$*

$\Gamma'', p{:}E$ *s.t.* $E = E', l{:}log$ *and* $intranse(\Gamma, l) = \vec{p}$*, then*

(a) $p \in \vec{p}$ *and*

(b) for all $p_i \in \vec{p}$ *we have* $\Gamma = \ldots, p_i{:}(E'_i, l{:}log_i), \ldots$*.*

(c) for all threads p' *with* $p' \notin \vec{p}$ *and where* $\Gamma = \ldots, p'{:}(E', l'{:}log'), \ldots$*, we have* $l' \neq l$*.*

5. *The function* $commit$ *closes a transaction. Note that the effect of one transaction may be copied into many threads due to spawn function, so when this transaction closes, all the threads containing it must synchronize via a joint commit: if* $commit(\vec{p}, \vec{E}, \Gamma) = \Gamma'$ *for* $\Gamma = \ldots, p_i : E_i, \ldots, p_j : E_j, \ldots$ *and for* $\vec{p} = intranse(\Gamma, l)$*,* $p_i \notin \vec{p}$*,* $p_j \in \vec{p}$ *then* $\Gamma' = \ldots, p_i : E'_i, \ldots, p_j : E'_j, \ldots$ *with* $|E'_i| = |E_i|$ *and* $|E'_j| = |E_j| - 1$*.*

The five reduction rules of the *global* semantics are given in Table 3.

3 Type system

The purpose of our type system is to determine an upper bound on the resource consumption of a TFJ program in terms of the maximum number of transactions running at any given moment. The type of a term is a sequence of *tagged* numbers, which is an abstract representation of the term's transactional behaviour, i.e., capturing the effects of starting new transactions, of committing local transactions, and of committing transactions jointly with other threads. An important property of the type system is compositionality, so that the analysis scales for larger programs.

3.1 Sequence of tagged numbers

To represent local transactional behaviour of a term, we use the set of four tags or signs $\{+, -, \#, \neg\}$ to abstractly represent, respectively, the opening, closing, (local) maximum and joint commit behaviour of the term. These tags are paired with a natural number. So our sequences of tagged numbers are sequences over $\mathbf{N} = \{^+n, ^-n, ^\#n, ^\neg n\}$ where n is a natural number.

Definition 4. $S = s_1 s_2 s_3 \ldots s_k$ *is a* sequence of tagged numbers *iff* $s_i \in \mathbf{N}$ *for all* $i \in \{1, \ldots, k\}$.

The empty sequence is denoted by $\emptyset$. We use $|S|$ to denote the length of S, i.e., k. $sign(s_i)$ gives the sign of s_i, i.e., $sign(s_i) \in \{+, -, \#, \neg\}$ and $|s_i|$ denotes the natural value of s_i without its sign. We write $sign(S)$ for the sequence of signs of S: $sign(s_1 \ldots s_k) = sign(s_1) \ldots sign(s_k)$. We also write $sign(s) \in sign(S)$ when the sign of s appears in $sign(S)$ and $sign(S_1) = sign(S_2)$ if the two sequences of signs are identical. We let $s, t, ..$ range over elements of sequences and m, n, l range over natural numbers.

The set of all tagged sequences can be classified into groups that represent the same behaviour in terms of local transactions. We use the most compact sequence in each group as the canonical element for that group:

Definition 5. *A sequence* S *is* canonical *iff* $sign(S)$ *does not contain '++', '−−', '##', '+−', '+¬', '+#¬' or '+#−' as subsequences, and furthermore* $|s| > 0$ *for all* $s \in S$*.*

For example $^+5^+9$ is not canonical but $^+5^\#9$ and $^\neg4^\neg6$ are. Similarly, $^+m^-n$ or $^+m^\neg n$ and $^+m^\#l^-n$ or $^+m^\#l^\neg n$ are not allowed in a canonical sequence.

The intuition here is that if a sequence contains the above sign patterns, we can simplified/shorten it without changing the interpretation of the sequence wrt. the resource consumption. The last two patterns can be combined to reflect the maximum number of local transactions. A sequence can be reduced to a canonical one by the following rewriting rules:

1. $s \Downarrow \emptyset$ if $|s| = 0$.
   The zero-valued components do not affect the behaviour of a term.

2. $ss' \Downarrow sign(s)(|s| + |s'|)$ if $sign(ss') = ++$ or $--$.
   $^+1$ represents the opening of 1 transaction. When many $+$ components are consecutive, we can shorten them to get the total number of transactions will be opened consecutively. Analogously for $^-1$ which represents the closing of 1 transaction.
   For example $^\#5^-3^-4^\#6 \Downarrow^\# 5^-7^\#6$, or $^\#5^+3^+4^\#6 \Downarrow^\# 5^+7^\#6$.

$$\frac{E,e \to E',e' \qquad reflect(p,E',\Gamma)=\Gamma'}{\Gamma,P \| p(e) \Rightarrow \Gamma',P \| p(e')} \text{ G-PLAIN}$$

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

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

$$\frac{\Gamma{=}\ldots,p{:}E \quad E{=}\ldots,l{:}log \quad intranse(\Gamma,l)=\vec{p}=p_1\ldots p_k \quad commit(\vec{p},\vec{E},\Gamma)=\Gamma' \quad \forall i \in \{1,\ldots,k\}}{\Gamma,\ldots \| p_i(let\ x{=}commit\ in\ e_i) \| \ldots \Rightarrow \Gamma',\ldots \| p_i(let\ x{=}null\ in\ e_i) \| \ldots} \text{ G-COMM}$$

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

Table 3: Global semantics

3. ${}^{\#}m{}^{\#}n \Downarrow^{\#} max(m,n)$.

   As said, the # components represent the local maximum number of transactions. That is the reason why when shortening a sequence, we choose the larger one from two consecutive # components. In other words, we can simplify the meaning of ${}^{\#}m$ to express the number of nested transactions (which is in fact only true partially because of concurrent threads with joint commits). That means these m transactions can be opened concurrently at a moment when the local resource consumption will be maximized. The ${}^{\#}m{}^{\#}n$ pattern shows that the m nested transactions and the n ones are sequential. They affect independently the local maximum resource allocation. So, when shortening them, we choose the one with the larger value.

4. ${}^{+}m{}^{\#}x{}^{-}n \Downarrow^{+} (m-\mu)^{\#}(x+\mu)^{-}(n-\mu)$ where $\mu = min(m,n)$

   This rule takes care of increasing the number of nested transactions when we have more opening-closing pairs surrounding the current nested transactions.

   For example:

   - ${}^{+}5{}^{\#}3{}^{-}2 \Downarrow {}^{+}(5-2)^{\#}(3+2)^{-}(2-2)$ $\Downarrow^{+} 3{}^{\#}5{}^{-}0$

5. ${}^{+}m{}^{\#}x{}^{\neg}n \Downarrow^{+} (m-1)^{\#}(x+n)$

   The $\neg$ components capture the number of threads inside the latest opened transaction. Each spawned thread makes a copy of this transaction into its own local environment. The transaction thus when closing will contribute as much as its number of threads to the local maximal behaviour.

   For example:

   - ${}^{+}2{}^{\#}4{}^{\neg}2 \Downarrow^{+} 1{}^{\#}6$

Two sequences are equivalent if they are their canonical sequences coincide:

Definition 6. *Two sequences of tagged numbers S_1 and S_2 are* equivalent, *written $S_1 \cong S_2$, iff they both can be reduced to the same canonical sequence.*

To represent the transactional behaviour, we need a few structures and corresponding 'reduction' operators. The first operator $\oplus$ is used calculate the sum of two sequences representing resource behaviour of two threads having joint commits *in the global semantics*, such as $spawn(e_1; commit); e_2; commit; e_3;$ where the e_i's are closed wrt. transactions.

Definition 7. *Given two sequences $S_1 = s_1 \ldots s_k$ and $S_2 = t_1 \ldots t_k$ such that $sign(S_1) = sign(S_2)$, the $\oplus$ operation is defined as follows: $S_1 \oplus S_2 = u_1 \ldots u_k$ where $u_i = sign(s_i)(|s_i| + |t_i|)$.*

As illustrated by Figure 1(b) we need to identify the joint commit points and take the *total of the peak resources* for each joint commit segment. Note that in $spawn(e_1; commit); e_2; commit; e_3;$ we need to type $e = e_1; commit$ first and if a joint commit is needed then the type S_1 of e will contain at least one s with sign $-$ or $\neg$. But e can be sequence with e_2, then with $commit; e_3$; so S_2 may not contain any s with sign $-$ or $\neg$, because the joint commit may appear later.

Definition 8. *Let $S_1 = s_1 \dots s_{k_1}$ and $S_2 = t_1 \dots t_{k_2}$ be two canonical sequences. Suppose that h_1, h_2 are the smallest indices such that $sign(s_{h_1}), sign(t_{h_2}) \in \{-, \neg\}$ and they are 0 if such elements do not exist. Let $\sigma = sign(s_{h_1} t_{h_2})$, $n_1 = |s_{h_1}|$ and $n_2 = |t_{h_2}|$. The merging operation, notation $S_1 \otimes S_2$, is defined recursively as follows:*

$$S_1 \otimes S_2 = \begin{cases} S_1 \oplus S_2 \textit{ if } S_i = \#n \textit{ for } i = \{1,2\} \\ S^{\neg}(1+n_2)(^{-}(n_1-1)S_1' \otimes S_2') \textit{ if } \sigma = -\neg \\ S^{\neg}(n_1+1)(S_1' \otimes ^{-}(n_2-1)S_2') \textit{ if } \sigma = \neg - \\ S^{\neg}2(^{-}(n_1-1)S_1' \otimes ^{-}(n_2-1)S_2') \textit{ if } \sigma = -- \\ S^{\neg}(|s_i|+|t_j|)(S_1' \otimes S_2') \textit{ otherwise} \end{cases}$$

where $s' = s_{i-1}$ if $i > 1$ and $s' = \#0$ otherwise, and similarly, $t' = t_{j-1}$ if $j > 1$ and $t' = \#0$ otherwise, $S = s' \oplus t'$ and $S_1' = s_{i+1} \dots s_n$ and $S_2' = t_{j+1} \dots t_m$.

With this definition, we can see that $\neg$ components express the number of joint commits for the latest opened transaction, i.e the number of threads in this transaction; and each thread in one transaction will increase the resource consumption by one. $s_{h_1}$and t_{h_2} determine the first joint commits of two sequences.

The merging operation is compositional on the right.

Definition 9. $(S_1 \otimes S_2)\ S_3 = S_1 \otimes (S_2\ S_3)$.

For conditionals *if v then e_1 else e_2* we need a simpler merge operation but require that the external transactional behaviour of e_1 and e_2 is the same – their local ones will be their maximum.

Definition 10. *Let $S_1 = s_1 \dots s_n$ and $S_2 = t_1 \dots t_m$ be two canonical sequences and suppose that $s_i \in S_1$, $t_j \in S_2$ are the first elements (from the left) such that $sign(s_i) = sign(t_j) \in \{+, -, \neg\}$. The max operation, notation $S_1 \odot S_2$, is defined recursively as follows:*

$$S_1 \odot S_2 = \begin{cases} ^{\#}x \textit{ if } (S_1 =^{\#} x \wedge S_2 = \emptyset) \\ \qquad \vee (S_1 = \emptyset \wedge S_2 =^{\#} x) \\ ^{\#}max(m,n)s_i(S_1' \odot S_2') \textit{ otherwise} \end{cases}$$

where $m = |s_{i-1}|$ if $i > 1$ and $m = 0$ otherwise, and similarly, $n = |t_{j-1}|$ if $j > 1$ and $n = 0$ otherwise, and $S_1' = s_{i+1} \dots s_n$ and $S_2' = t_{j+1} \dots t_m$.

For example, $S_1 = {}^{-}2^{\neg}3^{\#}4$ and $S_2 = {}^{\#}2^{-}2^{\#}4^{\neg}3^{\#}5$ we have $S_1 \odot S_2 = {}^{\#}2^{-}2^{\#}4^{\neg}3^{\#}5$.

If a program has some threads running in parallel independently, i.e., without joining commits, we need a to be able to express the type of such program: parallel notation, denoted $\|$.

Definition 11. *If $S_1 = {}^{\#}m$ and $S_2 = {}^{\#}n$ then $S_1 \parallel S_2 = {}^{\#}(m+n)$*

In our approach, each term in the *local-level semantics* has as type a sequence of tagged numbers. The size of the heap (represented as the $+$ component) concatenated with this type tells us about the maximum resource consumption during the execution of that term. We define a function, which calculates the maximum resource consumption given a tagged sequence as input.

Definition 12. *Given a sequence of tagged numbers S, the function $total(S)$ is defined by the following steps:*

1. *Change S to a new sequence $s_1 \dots s_n$ such that $\forall i \in \{2, \dots, n\}$ if $sign(s_i) = \#$ then $sign(s_{i-1}) = +$ and $\forall j \in \{1; \dots; n-1\}$ if $sign(s_j) = +$ then $sign(s_{j+1}) = \#$.*

2. *$total(S) = max(|s_i| + |s_{i+1}|)\ \forall i \in \{1, \dots, n-1\}$ and $sign(s_i) = +$.*

For example, let $S =^{+} 2^{\#}4^{+}1^{\#}4^{+}3^{\#}0$, then $total(S) = max(2+4, 1+4, 3+0) = 6$. Also the equivalent sequence $S' =^{+} 2^{\#}4^{+}1^{\#}4^{+}3$ (i.e., $S \cong S'$) has the same $total(S') = 6$.

3.2 Local level

On the local level, the typing judgements are of the form:

$$E \vdash e : T \tag{1}$$

where T expresses the type of expression e; E is the local environment. T here captures the information about resource consumption of e in terms of maximum number of concurrent transactions at any given moment.

Definition 13. $T = S \mid T \otimes T \mid T \odot T \mid T^{\rho} \mid T \parallel T$

S here is a sequence of tagged numbers. TT concatenates two types. The $\otimes$, $\odot$ operations, $\parallel$ notation have the meanings the same as ones in sequence of tagged numbers. When T satisfy the conditions of these operations, notation; we can apply the corresponding *definition* to calculate the final type, otherwise, we keep the expressions un-calculated. The term with type T^{ρ} is executed in a thread which is parallel with the thread spawning it.

The local derivation rules for expressions by our type system in shown in Table 4. The rule T-ONACID and T-ONACID are used to type expressions $onacid$ and $commit$, respectively. T-LET$_1$, T-LET$_2$ take care of sequential composition of e_1 and e_2. The difference between these two rules is that e_1 in the latter case is of type T_1^{ρ}, i.e., t e_1 is of the form $spawn(e')$ (T-SPAWN). Conditionals $if\ v\ then\ e_1\ else\ e_2$ are given the type $T_1 \odot T_2$, where T_1 is type of e_1, T_2 of e_2. T-CALL rule tells us that if the a method is executed in environment E, and the returned type is T, then the function call will have type T. The returned type of a method is the type of the expression in its body. The final three rules express that a expression of value, field access or field update will have $^{\#}0$ as type. Such expressions do not affect the resource consumption.

Definition 14 (Idempotence)**.** *Let T be a type of any expression, then $(T^{\rho})^{\rho} = T^{\rho}$.*

This definition means that $^{\rho}$ only makes sense when $spawn(e)$ is followed by an other expression (T-LET$_2$ rule). When $spawn(e)$ stands alone, e.g $spawn(spawn(e))$, the first $^{\rho}$ can be ignored.

3.3 Global level

The global level of TFJ's semantics captures the parallel of composition of threads. We give the derivation rules for a process in table 5. The T-PAR-rule takes care of type of a program composed from two program parts P_1 and P_2 running in parallel independently. They are independent because they initially run in their own global environment

$$\frac{E \vdash e{:}T}{p{:}E \vdash p(e){:}T}\ \text{T-THREAD} \qquad \frac{\Gamma_1 \vdash P_1{:}T_1 \qquad \Gamma_2 \vdash P_2{:}T_2}{\Gamma_1,\Gamma_2 \vdash P_1 \parallel P_2{:}T_1 \parallel T_2}\ \text{T-PAR}$$

Table 5: Type system (global level)

Γ_1 and Γ_2 and do not synchronize with each other via joint commits. In the case of $spawn(e)$, a new thread is created and which may synchronize with the parent thread via joint commits (cf. rule T-SPAWN).
Now, well-typed programs are defined as follows:

Definition 15. *A program is* well-typed *if its type contains only one # component.*

As mentioned, the type of an expression or thread is a sequence of tagged numbers, and the # component tells us about the upper bound in resource consumption of that process. Applying the rules to our example from Section 1, we can compute the for that program as follows:

$\Gamma \vdash P : T$
$T =^{+} 1\,^{+}1[^{+}1^{-}1^{-}1^{-}1 \otimes^{+} 1(^{+}1^{+}1^{-}1^{-}1^{-}1^{-}1^{-}1$
$\otimes^{-} 1^{+}1^{+}1^{+}1^{-}1^{-}1^{-}1^{-}1^{+}1^{+}1^{+}1^{+}1^{-}1^{-}1^{-}1^{-}1^{-}1)]$
$=^{+} 2[^{\#}1^{-}2 \otimes^{+} 1(^{\#}2^{-}3 \otimes^{-} 1^{\#}3^{-}1^{\#}4^{-}1)]$
$=^{+} 2(^{\#}1^{-}2 \otimes^{+} 1^{\#}2^{\neg}2^{\#}3^{\neg}2^{\#}4^{\neg}2)$
$1 =^{+} 2(^{\#}1^{-}2 \otimes^{\#} 4^{\#}3^{\neg}2^{\#}4^{\neg}2)$
$=^{+} 2(^{\#}1^{-}2 \otimes^{\#} 4^{\neg}2^{\#}4^{\neg}2)$
$=^{+} 2^{\#}5^{\neg}3^{\#}4^{\neg}3$
$=^{+} 1^{\#}8^{\#}4^{\neg}3$
$=^{+} 1^{\#}8^{\neg}3$
$=^{\#} 11$
The program is well-typed and maximum resource consumption is 11 in terms of number of concurrent transactions.

4 Correctness

This section establishes the soundness of the effect of our type system, i.e, that the static estimation over-approximates the actual potential resource consumption of a program. We start by defining the actual resource consumption of a program:

Definition 16 (Resource consumption)**.** *The weight of a local environment E, written $|E|$ is defined as its length, i.e., the number of its $l{:}log$ bindings. The weight of a*

$$\frac{}{E \vdash onacid{:}{}^{+}1} \text{ T-ONACID} \qquad \frac{|E|>0}{E \vdash commit{:}{}^{-}1} \text{ T-COMMIT}$$

$$\frac{E \vdash e_1{:}T_1 \quad E \vdash e_2{:}T_2}{E \vdash let\ x{=}e_1 in\ e_2{:}T_1 T_2} \text{ T-LET}_1 \qquad \frac{E \vdash e_1{:}T_1^{\rho} \quad E \vdash e_2{:}T_2}{E \vdash let\ x{=}e_1 in\ e_2{:}T_1 \otimes T_2} \text{ T-LET}_2$$

$$\frac{E \vdash e_1{:}T_1 \quad E \vdash e_2{:}T_2}{E \vdash if\ v\ then\ e_1\ else\ e_2{:}T_1 \odot T_2} \text{ T-COND} \qquad \frac{E \vdash e{:}T}{E \vdash spawn\ e{:}T^{\rho}} \text{ T-SPAWN}$$

$$\frac{mtype(C,m){:}E \rightarrow T}{E \vdash v.m(\vec{v}){:}T} \text{T-CALL} \qquad \frac{}{E \vdash v{:}{}^{\#}0} \text{ T-Val}$$

$$\frac{}{E \vdash v.f{:}{}^{\#}0} \text{ T-ACCESS} \qquad \frac{}{E \vdash v.f{:=}v'{:}{}^{\#}0} \text{ T-UPDATE}$$

Table 4: Type system (local level)

global environment Γ, written $|\Gamma|$ is defined as the sum of weights of its local environments.

Given a term e with type S executed in local environment E (in the local semantics, all terms will have types as canonical sequences of tagged numbers without operations or notations). $|E|$ means the current resource allocation, i.e the number of current local opening transactions. We define the function $total(E, e)$ to estimate the maximum resource consumption of the program during the execution of e:

Definition 17. *If $E \vdash e : S$ then the maximum resource consumption during executing e is estimated by the following equations:*

$$total(E, e) = total({}^{+}|E|S) \tag{2}$$

${}^{+}|E|S$ is a sequence of tagged numbers concatenated from ${}^{+}|E|$ and S. $|E|$ is the number of opening transactions, and e might contain either further $commit$s to close these transactions or other commands which affect the local resource allocation. So, ${}^{+}|E|S$ represents the further resource consumption of e.

Because a new thread will copy the current local variables into its own local environment, two local environments may contain some common transactions.

Definition 18. *The function $common(E_1, E_2)$ returns the number of common transactions in E_1 and E_2.*

We also define the function for estimating maximum resource consumption during executing a program:

Definition 19. *Given a program $P = p_1(e_1) \parallel \ldots \parallel p_n(e_n)$. The global environment is $\Gamma = p_1{:}E_1, \ldots, p_n{:}E_n$. The maximum resource consumption during executing this program $total(\Gamma, P)$ is computed as follows:*

1. *The result is computed from the set of sequences: $T = \{{}^{+}|E_i|S_i$ where $i = \{1, \ldots, n\}\}$ in which $E_i \vdash e_i : S_i$.*

2. *$\forall i, j \in \{1, \ldots, n\}, i \neq j$, if $common(E_i, E_j) > 0$, then remove ${}^{+}|E_i|S_i$ and ${}^{+}|E_j|S_j$ from T, also add $({}^{+}|E_i|S_i) \oplus ({}^{+}|E_j|S_j)$ into T.*
   After this step, suppose $T = \{T_1, \ldots, T_m\}$.

3. *$total(\Gamma, P) = \sum_{i=1}^{m} total(T_i)$.*

The purpose of step 2 is to take care of parallel threads with joining commits.

The following lemmas are needed for proving the correctness of our analysis.

Lemma 1 (Subject reduction (local))**.** *If $E \vdash e : S$ and $E, e \rightarrow E', e'$ then $E' \vdash e' : S'$ and $total(E, e) \geq total(E', e')$.*

Lemma 2 (Subject reduction (global))**.** *If $\Gamma \vdash P : T$ and $\Gamma, P \rightarrow \Gamma', P'$ then $\Gamma' \vdash P' : T'$ and $total(\Gamma, P) \geq total(\Gamma', P')$.*

5 Conclusion

We develop and investigate a type system for statically estimating the resource upper bound for a transactional model supporting nested and multi-threaded transactions with join synchronization. The system statically approximates the maximum number of concurrent transactions opening in a program. This work extends our previous work by removing a restriction on child threads where in the previous work we did not allow opening a new transaction inside spawned threads after their parent thread has committed. The representation of type judgements is also simplified by using a linear numeric representation instead of a tree representation; the calculation is therefore simpler while we still guarantee that our analysis is compositional.

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 Martin Wirsing, Martin Hofmann, and Axel Rauschmayer, editors, *TGC'10*, number 6084 in Lecture Notes in Computer Science. Springer-Verlag, 2010.

[3] Gilles Barthe, Mariela Pavlova, and Gerardo Schneider. Precise analysis of memory consumption using program logics. In *SEFM '05*, Washington, DC, USA, 2005. IEEE.

[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 *Lecture Notes in Computer Science*. Springer-Verlag, 2005.

[6] Philippe Clauss, Federico Javier Fernandez, Diego Garbervetsky, and Sven Verdoolaege. Symbolic polynomial maximization over convex sets and its application to memory requirement estimation. *IEEE Transactions on Very Large Scale Integration Systems*, 17, 2009.

[7] Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of POPL '03*. ACM, January 2003.

[8] 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 *Lecture Notes in Computer Science*. Springer-Verlag, 2006.

[9] John Hughes and Lars Pareto. Recursion and dynamic data-structures in bounded space: Towards embedded ML programming. *SIGPLAN Notices*, 34(9), 1999.

[10] Suresh Jagannathan, Jan Vitek, Adam Welc, and Antony Hosking. A transactional object calculus. *Sci. Comput. Program.*, 57(2):164–186, August 2005.

[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 *SEFM'08*, 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] Thi Mai Thuong Tran, Martin Steffen, and Hoang Truong. Estimating Resource Bounds for Software Transactions. Technical report 414, University of Oslo, Dept. of Informatics, December 2011.