Calculate statically maximum log memory used by multi-threaded transactional programs

Anh-Hoang Truong[1], Ngoc-Khai Nguyen[2], Dang Van Hung[1], and Dang Duc Hanh[1]

[1] VNU University of Engineering and Technology
[2] Hanoi University of Natural Resources and Environment

Abstract. During the execution of multi-threaded and transactional programs, when new threads are created or new transactions are started, memory areas called logs are implicitly allocated to store copies of shared variables so that the threads can independently manipulate these variables. It is not easy to manually calculate the peak of memory allocated for logs when programs have arbitrary mixes of nested transactions and new thread creations. We develop a static analysis to compute the amount of memory used by logs in the worst execution scenarios of the programs. We prove the soundness of our analysis and we show a prototype tool to infer the memory bound.

Keywords: memory bound, transactional memory, static analysis

1 Introduction

We address the problem of determining the memory bound of transactional programs at compile time to ensure that they can run smoothly without out of memory errors. To describe the problem more precisely, we use a core language in which transactional and multi-threading statements are based on [12] and other features are generalized so that transactional programs in other imperative languages can be translated to our language for the memory estimation problem. The key features we borrow from [12] allow programmers to mix creating new threads and opening new transactions.

When one transaction is nested in another, we called the former parent transaction, and the latter child one. Child transactions must commit before their parent does. When a transaction is started, a memory area called *log* is allocated for storing a copy of shared variables. A transaction that started but has not committed yet is called an *open* transaction. Inside open transactions, the programmers can also create new threads. A new thread in this case will make a copy of transaction logs of its parent thread. When a parent thread commits a transaction, all the child threads that are created inside the parent transaction must join the commit with their parent. This kind of commits is called *joint commit*, and the time when these commits occur is *joint commit point*. Joint commits act as implicit synchronizations of parallel threads. If a transaction has no child threads, the commit is a normal (local) commit. Both types of commits release the memory allocated for the logs. Now, we can formulate the problem as follows. Given the size of transaction logs in the program, compute the maximal memory requirement for the whole program.

In our previous studies [13, 15, 16], we built type systems to count the maximum number of logs that can coexist at runtime. This number gives us raw information about the memory used by transaction logs. To infer precisely the maximal amount of memory that transaction logs may use, we need information about the size of each log. Therefore, in this work we extend the start transaction statement in our previous work to contain this information. But this does not mean the programmers have to annotate this size information as it can be synthesized by identifying shared variables of transactions. Then, we develop a type system to estimate the maximum memory that transaction logs may require. It turns out that the ideas of type structures in our previous work can be reused, but the type semantics and typing rules are novel and different from the those in our previous works. The type system with its soundness proofs and a prototype tool are our main contributions in this work.

Estimating resource usage in general and estimating memory resource in particular has always been an active research problem. In [17], Wegbereiter gave methods to analyze the complexity of Lisp programs by using recursive function. Hughes and Pareto [11] introduce a strict, first-order functional language with a type system such that well-typed programs run within the space specified by the programmer. Hofmann and Jost [10] compute the linear bounds on heap space for a first-order functional language. Later, they use a type system to calculate the heap space bound as a function of input for an object oriented language. Wei-Ngan Chin *et al.* [8] studied memory usages of object-oriented programs. In [7] 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.* [4, 6] calculate symbolic approximation of memory bounds for Java programs. In [5] the authors propose type systems for component languages with parallel composition but the threads run independently. Albert *et al.* have many works in resource estimation for programs. In [3], they compute the heap consumption of a program as a function of its data size. In [1, 2], they studied the problem in the context of distributed and concurrent programs. In [14], Pham *et al.* proposes a fast algorithm to statically find the upper bounds of heap memory for a class of JavaCard programs. In [9], Jan Hoffmann and Zhong Shao also use type system to estimate resource usage of parallel programs but for a functional language.

The works mentioned above focus only on sequential or functional language. The language that we study here is different as it is multi-threaded and nested transactional language with complex and implicit synchronization. The type system that we develop in this work is significantly different from the ones in our previous work even though it looks similar. Note that compared to our previous work, the semantics of several type elements are completely different.

The rest of the paper is structured as follows. In the next section we informally explain the problem and the approach via 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 represented in Section 5. A prototype tool with its main algorithm to compute memory bound is described in Section 6. Section 7 concludes and outlines our future work.

2 Motivating example

We use the sample program in Listing 1.1, borrowed from [15], to explain the problem and our approach. Note that this example focuses on the core of the language. Real programs will have many other constructs of programming languages such as procedures, method calls, message passing, variables, and other computation primitives. These programs can be converted to equivalent programs, w.r.t. transactional and multi-threading behaviours, in our core language.

In this code snippet, the statements `onacid` and `commit` are for starting and closing a transaction [12]. The statement `spawn` is for creating a thread with the code represented by the parameters of the statement. The `onacid` statement in our previous works has no parameters, but in this work it is associated with a number to denote the size of the memory needed to allocate to the log of the transaction at runtime.

Listing 1.1. A nested multi-threaded program.

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

The behavior of this program is depicted in Figure 1. The starting transaction statement `onacid` and ending transaction statement `commit` are denoted by [and] in the figure, respectively. The statement `spawn` creates a new thread running in parallel with its parent thread and is described by the horizontal lines. The new thread duplicates the logs of the parent thread for storing a copy of the value of variables of the parent thread so that it can manipulate these variables independently.

In this example, when spawning thread 1, thread 0 has opened two transactions, so thread 1 makes two copies of thread 0's logs and hence on line 3 the parameter of `spawn` contains the last two commits to close them. These commits must be synchronized with the commits in lines 8 and 10 of the thread 0, and form a so-called joint commit. Joint commits are described by the rectangular dotted line in Figure 1. The right-hand edges of the boxes mark these synchronizations. The left-hand edges are the corresponding open transactions that the joint commits must jointly close.

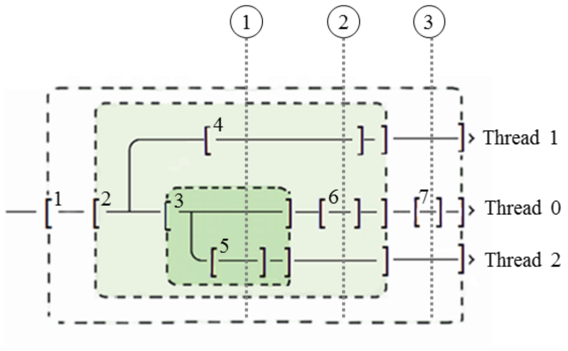

Fig. 1. Threads dependencies and join commits.

We now try to manually calculate the maximum memory used by logs to answer the question: How much more memory is required by the software transactional memory mechanism?

- At the point ①: The total memory used for logs, denoted by m_1, is the sum of:
  - log memory for the first two and the forth transactions of thread 0: 1+2+3=6,
  - log memory for thread 1: (1+2)+4=7, since thread 1 clones two logs of its parent thread,
  - log memory for thread 2: (1+2+3)+5=11 since thread 2 clones three logs of its parent thread.

  So m_1 = 6+7+11 = 24.
- At the point ②: The total memory used for logs, denoted by m_2, is the sum of:
  - log memory for the first three transactions of thread 0: 1+2+6=9,
  - log memory for thread 1: (1+2)+4=7 as above,
  - log memory for thread 2: (1+2)=3, since thread 2 now has only two logs copied from its parent thread.

  So m_2 = 9+7+3 = 19.
- At the point ③: Similarly, we have m_3 = 8+1+1 = 10.
  So in the worst case, the maximum memory allocated to logs is $\max(m_1, m_2, m_3) =$ 24 units.

Note that our language was inspired and abstracted from [12] as we focus on transactional and multi-threaded features to estimate the additional memory, which is implicitly allocated by the implementation the language at runtime. The additional commits that a child thread has, e.g., the last two commits in `spawn(onacid(4);commit;commit;commit);`, make the language harder to use but this gives programmers more power to control when the commits can start. This is because between these commits, there can be other computations and transactions as well as new threads, e.g., `spawn(onacid(4);commit;commit;e';commit;e'');` and `e, e''` may be some other lengthy computations and the programmers want to do the commits before them. The analysis that we will present in Section 4 will signal error for programs that do not have matching `onacid`s and `commit`s.

The language can be simplified by allowing the compiler to automatically insert these commits to the end of the thread, but then programmers will have less control on the behaviour of the transactions. Or even better, a smarter compiler can insert the missing commits for a child threads as soon as the shared variables are no longer being manipulated by the threads. For example, if `e''` in the example in the previous paragraph does not access any shared variables, then the compiler can insert a commit before `e''`. In both situations, the type system that we present here can compute the worst scenario of log memory that the runtime required.

So, there is a trade-off in the language design. Our language design here can cover the other cases where commits are automatically inserted to the child threads by the compiler. Last, here the language does not have loops because we believe that spawning threads and creating transactions are expensive operations and ones usually put these constructs outside loops with unknown number of iterations. For loops with a fixed number of iterations, the sequential composition can encode them.

3 Transactional language

3.1 Syntax

Figure 2 gives the syntax of our language, called TM (transactional memory). In the first line, *program* P can be *empty*, notation 0, or a composition of parallel threads $P \parallel P$. $p(e)$ denotes a thread with *identifier* p executing term e.

For *term* e, we assume the language has a set of atomic statements A, ranged over by α. **onacid**(n) and **commit** are statements for starting and committing a transaction. Parameter n in **onacid**(n) represents the number of memory units allocated when opening the new transaction. Note that in reality n can be synthesized by the compiler based on the sizes of shared variables in the scope of the transaction. That means programmers do not have to annotate this size information. $e_1; e_2$ denotes sequencing of statements and $e_1 + e_2$ denotes branching. The last statement **spawn**(e) is for creating a new thread executing e.

3.2 Dynamic semantics

The (global) run-time environment is structured as a collection of local environments. Each local environment is a sequence of logs with their sizes. We formally define the local and global environments as follows.

$$
\begin{aligned}
P &::= 0 \quad | \quad P \parallel P \quad | \quad p(e) \\
e &::= \alpha \quad | \quad \mathbf{onacid}(n) \quad | \quad \mathbf{commit} \quad | \\
&\qquad e_1; e_2 \quad | \quad e_1 + e_2 \quad | \quad \mathbf{spawn}(e)
\end{aligned}
$$

Fig. 2. TM syntax

Definition 1 (Local environment). *A* local environment E *is a finite sequence of* log id*'s and their size:* $l_1{:}n_1; \ldots; l_k{:}n_k$*. The environment with no element is called the* empty environment*, denoted by* ϵ*.*

For an environment $E = l_1{:}n_1; \ldots; l_k{:}n_k$, we denote $[\![E]\!] = \sum_{i=1}^{k} n_i$ the number of memory units used E, and $|E| = k$ the number of elements in E.

Definition 2 (Global environment). *A* global environment Γ *is a collection of* thread id*'s and their local environments,* $\Gamma = \{p_1{:}E_1, \ldots, p_k{:}E_k\}$*.*

The log memory used by $\Gamma = \{p_1{:}E_1, \ldots, p_k{:}E_k\}$, denoted by $[\![\Gamma]\!]$, is defined by: $[\![\Gamma]\!] = \sum_{i=1}^{k} [\![E_i]\!]$.

For a global environment Γ and a set P of threads, we call the pair Γ, P a *state*. We have a special state *error* for stuck states—the states at which no other transition rules can be applied. The dynamic semantics is defined by *transition rules* between states of the form $\Gamma, P \Rightarrow \Gamma', P'$ or $\Gamma, P \Rightarrow \mathit{error}$ in Table 1.

$$
\frac{p' \mathit{fresh} \quad \mathit{spawn}(p, p', \Gamma) = \Gamma'}{\Gamma, P \parallel p(\mathbf{spawn}(e_1); e_2) \Rightarrow \Gamma', P \parallel p(e_2) \parallel p'(e_1)} \text{ S-SPAWN}
$$

$$
\frac{l\ \mathit{fresh} \quad \mathit{start}(l{:}n, p, \Gamma) = \Gamma'}{\Gamma, P \parallel p(\mathbf{onacid}(n); e) \Rightarrow \Gamma', P \parallel p(e)} \text{ S-TRANS}
$$

$$
\frac{\mathit{intranse}(\Gamma, l : n) = \mathbf{p} = \{p_1, .., p_k\} \quad \mathit{commit}(\mathbf{p}, \Gamma) = \Gamma'}{\Gamma, P \parallel \coprod_1^k p_i(\mathbf{commit}; e_i) \Rightarrow \Gamma', P \parallel \coprod_1^k p_i(e_i)} \text{ S-COMM}
$$

$$
\frac{i = 1, 2}{\Gamma, P \parallel p(e_1 + e_2) \Rightarrow \Gamma, P \parallel p(e_i)} \text{ S-COND}
$$

$$
\frac{}{\Gamma, P \parallel p(\alpha; e) \Rightarrow \Gamma, P \parallel p(e)} \text{ S-SKIP}
$$

$$
\frac{\Gamma = \Gamma' \cup \{p : E\} \quad |E| = 0}{\Gamma, P \parallel p(\mathbf{commit}; e) \Rightarrow \mathit{error}} \text{ S-ERROR-C} \qquad \frac{\Gamma = \Gamma' \cup \{p : E\} \quad |E| > 0}{\Gamma, P \parallel p() \Rightarrow \mathit{error}} \text{ S-ERROR-O}
$$

Table 1. TM dynamic semantics

Table 1 uses some auxiliary functions described as follows. Note that the function names are from [12] and congruence rules are applied for processes: $P \parallel P' \equiv P' \parallel P$, $P \parallel (P' \parallel P'') \equiv (P \parallel P') \parallel P''$ and $P \parallel 0 \equiv P$.

- In the rule S-SPAWN, the function $spawn(p, p', \Gamma)$ adds to Γ a new element with thread id p' and a local environment cloned from the local environment of p. Formally, suppose $\Gamma = \{p : E\} \cup \Gamma''$ and $spawn(p, p', \Gamma) = \Gamma'$, then $\Gamma' = \Gamma \cup \{p' : E'\}$ where $E' = E$.
- In the rule S-TRANS, the function $start(l : n, p, \Gamma)$ creates one more log with the label l and with the size n units of memory at the end of the local environment of p_i. If $start(l{:}n, p_i, \Gamma) = \Gamma'$ where $\Gamma = \{p_1 : E_1, \ldots, p_i : E_i, \ldots, p_k : E_k\}$ and l is a fresh label, then $\Gamma' = \{p_1 : E_1, \ldots, p_i : E_i', \ldots, p_k : E_k\}$, where $E_i' = E_i; l : n$.
- In the rule S-COMM, the function $intranse(\Gamma, l : n)$ returns a set of all threads, denoted by $\mathbf{p}$, in Γ whose local environments contain log id l and this log id is the last element of the local environments. That is $intranse(\Gamma, l{:}n) = \mathbf{p} = \{p_1, .., p_k\}$ then:
  - for all $i \in \{1..k\}$, p_i has the form $E_i'; l : n$.
  - for all $p' : E' \in \Gamma$ such that $p' \notin \{p_1, .., p_k\}$ we have E' does not contain log id l.
- Also in the rule S-SPAWN, the function $commit(\mathbf{p}, \Gamma)$ removes the last log id in the local environments of all threads in $\mathbf{p}$. That is, suppose $intranse(\Gamma, l{:}n) = \mathbf{p}$ and $commit(\mathbf{p}, \Gamma) = \Gamma'$, then for all $p' : E' \in \Gamma'$, if $p' \in \mathbf{p}$, then $p' : (E'; l : n) \in \Gamma$. Otherwise, $p' : E' \in \Gamma$.

Note that function *spawn* copies the labels of the parent thread's environment to the local environment of the new thread and the function *intranse* finds these labels to identify threads that need synchronization in a joint commit.

The rules in Table 1 have the following meanings:

- The rule S-SPAWN says that a new thread is created with the statement **spawn**. The statement **spawn**(e_1) creates a new thread p' executing e_1 in parallel with its parent thread p, and changes the environment from Γ to Γ'.
- The rule S-TRANS is for the cases where thread p creates a new transaction with the statement **onacid**. A new transaction with label l is created, and changes the environment from Γ to environment Γ'.
- The rule S-COMM is for committing a transaction. In this rule $\coprod_1^k p_i(E_i)$ stands for $p_1(e_1) \parallel .. \parallel ..p_k(e_k)$. If the current transaction of thread p is l, then all threads in the transaction l have to joint commit when transaction l commits.
- The rule S-COND is to select one of the two branches e_1 or e_2 to continue.
- The rule S-SKIP is for other computation statements of the language, which we assume they do not interfere with our multi-threading and transactional semantics, so we can skip them.
- The rules S-ERROR-C and S-ERROR-O are used in cases there are mismatches in starting and committing transactions. For instance, `onacid;spawn(commit ;commit);commit` has a mismatch in the second commit in `spawn(commit; commit)`. $p()$ in S-ERROR-O means there is missing commit(s) in the program.

4 Type system

The main purpose of our type system is to identify the maximum log memory that a TM program may require. The type of a term in our system is computed from what we call sequences of *tagged numbers*, which is an abstract representation of the term's transactional behavior w.r.t. log memory.

4.1 Types

Inspired from our previous works [15], our types are finite sequences over the set of so called *tagged numbers*. A tagged number is a pair of a *tag* and a non-negative natural number $\mathbb{N}^+$. We use four tags, or signs, $\{+, -, \neg, \sharp\}$ for denoting opening, commit, joint commit and accumulated maximum of memory used by logs, respectively. The set of all tagged number is denoted by ${}^{T}\mathbb{N}$. So ${}^{T}\mathbb{N} = \{\,{}^{+}n\,,\ {}^{-}n\,,\ {}^{\sharp}n\,,\ {}^{\neg}n\ \mid\ n \in \mathbb{N}^+\}$. The meanings of these tag numbers is described below.

- The tag number ${}^{+}n$ says that the open transaction has a log whose size is n units of memory. Note that this semantics is different from ones in our previous works where it denotes the number of consecutive `onacid`s.
- The tag number ${}^{-}n$ means there are n consecutive commits statements,
- The tag number ${}^{\neg}n$ means there are n threads that require synchronization at a joint commit,
- The tag number ${}^{\sharp}n$ says the current maximum of memory units used by the term is n.

To help the readers better understand types, we give the following type examples. `onacid(2)` has type ${}^{+}2$, `commit` has type ${}^{-}1$, `onacid(2);commit` has type ${}^{+}2\ {}^{-}1$. Later, we will explain how this type can be converted to its equivalent form: ${}^{\sharp}2$, by matching and combining $+$ and $-$ elements. For the sequential composition statement `onacid(1); onacid(2); commit; commit`, its type is ${}^{+}1\ {}^{+}2\ {}^{-}1\ {}^{-}1$ or its equivalent form ${}^{+}1\ {}^{\sharp}2\ {}^{-}1$ or ${}^{\sharp}3$. For `spawn(onacid(4);commit ;commit;commit)`, its type is ${}^{+}4\ {}^{-}1\ {}^{-}1\ {}^{-}1$ and can be simplified to ${}^{\sharp}4\ {}^{\neg}1\ {}^{\neg}1$ by matching and combining $+$ and $-$ elements and identifying joint commits elements. Note that we do not combine the two consecutive ${}^{\neg}$. Instead, we will match and combine with a suitable ${}^{\neg}$ of some other term that will be executed in another thread. ${}^{\sharp}2\ {}^{\sharp}4$ and ${}^{\sharp}4\ {}^{\sharp}3$ can be converted to its equivalent type ${}^{\sharp}4$ since they all reflect that the maximum units of memory used is 4.

We will develop rules to associate a sequence of tagged numbers with a term in TM. During computation, a tag with zero (e.g. ${}^{+}0$, ${}^{-}0$, etc.) 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 we also automatically insert ${}^{\sharp}0$ element whenever needed.

In the following, let s range over ${}^{T}\mathbb{N}$, ${}^{T}\bar{\mathbb{N}}$ be the set of all sequences of tagged numbers, S range over ${}^{T}\bar{\mathbb{N}}$ and let $m, n, l, ..$ range over $\mathbb{N}$. The empty sequence is denoted by ϵ as usual. For a sequence S we denote by $|S|$ the length of S, and write $S(i)$ for the ith element of S. For a tagged number s, we denote $\mathrm{tag}(s)$ the tag of s, and $|s|$ the natural number of s (i.e. $s = {}^{\mathrm{tag}(s)}|s|$). For a sequence $S \in {}^{T}\bar{\mathbb{N}}$, we write $\mathrm{tag}(S)$ for the sequence of the tags of the elements of S and $\{S\}$ for the set of tags appearing in S. Note that $\mathrm{tag}(s_1 \ldots s_k) = \mathrm{tag}(s_1) \ldots \mathrm{tag}(s_k)$. We also write $\mathrm{tag}(s) \in S$ instead of $\mathrm{tag}(s) \in \{S\}$ for 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 behavior, 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* $\mathrm{tag}(S)$ *does not contain '$--$', '$\sharp\sharp$', '$+-$', '$+\sharp-$', '$+\neg$' or '$+\sharp\neg$' and $|S(i)| > 0$ for all i.*

The intuition here is that we can always simplify/shorten a sequence S without changing its interpretation. The seq function below reduces a sequence in ${}^{T}\bar{\mathbb{N}}$ to a canonical one. Note the pattern '$+-$' does not appear on the left, but we can insert ${}^{\sharp}0$ to apply the function. The last two patterns, '$+\neg$' and '$+\sharp\neg$', will be handled by the function jc later in Definition 8.

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

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

In this definition, the second and the third lines are for simplifying the representation. The last line is for local commits—the commits that do not synchronize with other threads.

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, `onacid(5)` on line 5 cannot run in parallel with `onacid(6)` on line 7.

With our type given to a term e, segments can be identified by examining the type of e in **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 $\mathit{first}(S, sig)$ 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 to be local commits. Then, when we discover that there is no local transaction starting statement (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 that has no $+$ element to a so-called *joint sequence.*

Definition 5 (Join). *Let $S = s_1 \dots s_k$ be a canonical sequence such that $+ \notin \{S\}$ and assume $i = \mathit{first}(S, -)$. Then, function* join(S) *recursively replaces $-$ in S by $\neg$ as follows:*

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

Note that in Definition 5 the canonical sequence S contains only $\sharp$ elements interleaved with $-$ or $\neg$ elements. After applying the join function, we get joint sequences. These joint sequences contain 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. The joint sequences are merged together in the following definition:

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:*

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

The definition is well-formed, since S_1, S_2 are joint sequences so they have only $\sharp$ and $\neg$ elements. In addition, the number of $\sharp$s 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. Note that for the merge function is used for terms like **spawn**$(e_1); e_2$, in which we compute the type for e_1, then apply the join function to obtain a joint sequence—the type of **spawn**(e_1). Then, we need to compute a matching joint sequence from e_2 to merge with the joint sequence of the type of **spawn**(e_1).

We need one more function, which we use to type terms of the form $e_1 + e_2$. For these terms, we require that the external transactional behaviors 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 behaviors. 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 we remove all $\sharp$ elements from them, then the remaining two sequences are identical. The* alt *function is recursively defined as:*

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

4.2 Typing rules

The language of types T is defined by the following syntax:

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

$$\frac{}{-n \vdash \textbf{onacid}(n) :\ {}^{+}n}\ \text{T-ONACID} \qquad \frac{n \in \mathbb{N}^{+}}{n \vdash \textbf{commit} :\ {}^{\neg}1}\ \text{T-COMMIT}$$

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

$$\frac{n_i \vdash e_i : S_i \quad i = 1,2 \quad S = \mathrm{seq}(S_1 S_2)}{n_1 + n_2 \vdash e_1; e_2 : S}\ \text{T-SEQ}$$

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

$$\frac{n \vdash e_i : S_i^{\rho} \quad i = 1,2 \quad S = \mathrm{merge}(S_1, S_2)}{n \vdash e_1; e_2 : S^{\rho}}\ \text{T-MERGE}$$

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

Table 2. Typing rules

The second kind of type S^{ρ} is used for term **spawn**(e) as it needs to synchronizes with their parent thread if there is any joint commit. The treatments of two cases are different, so we denote $\mathit{kind}(T)$ the kind of T, which can be empty (normal) or ρ depending on which case T is.

The type environment encodes the transaction context for the term being typed. The typing judgment is of the form:

$$n \vdash e : T$$

where $n \in \mathbb{N}$ is the type environment. When n is negative, it means e uses n units of memory for its logs when executing e. When n is positive, it means e can free n units of memory of some log.

The typing rules for our calculus are shown in Table 2. Note that we do not have rules for typing α as we assume they do not interfere with multi-threading and transactional semantics, so we can remove them when typing the programs. We assume that in these rules functions seq, jc, merge, 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 ρ 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 in T-MERGE. The remaining rules are straightforward except for the rule T-JC in which we need the new function jc (in Definition 8). The rule T-JC handles the joint commit between the threads running in parallel. The last $+$ element in S_1, say ${}^{+}n$, will be matched with the first $\neg$ element in S_2, say ${}^{\neg}l$ (Figure 3). But after ${}^{+}n$, there can be a $\sharp$ element, say ${}^{\sharp}n'$, so the local peak of memory units used by the term having type ${}^{+}n\ {}^{\sharp}n'$ is $n + n'$. Before

$^{\neg}l$ there can be a $^{\sharp}l'$, so when we do the joint commit of terms having type $^{\neg}l$ with its starting transaction having type ^{+}n the type of the segment will be $l' + l * n$. After combining ^{+}n from S_1 and $^{\neg}l$ from S_2 we can simplify the new sequences and repeat the join commits of jc. Thus, the function jc is defined as follows:

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

$$\begin{aligned} \mathrm{jc}(S_1' \, ^{+}n \, ^{\sharp}n', \, ^{\sharp}l' \, ^{\neg}l \, S_2') &= \mathrm{jc}(\mathrm{seq}(S_1' \, ^{\sharp}(n+n')), \mathrm{seq}(^{\sharp}(l' + l * n) \, S_2')) \ \textit{if } l > 0 \\ \mathrm{jc}(^{\sharp}n', \, ^{\sharp}l' \, S_2') &= \mathrm{seq}(^{\sharp}\mathrm{max}(n', l') \, S_2') \ \textit{otherwise} \end{aligned}$$

Note that in this definition of jc the pattern matching in the first line has higher priority than one in the second line.

As our type reflects the behavior of a term, so the type of a well-typed program contains only a sequence of single $^{\sharp}n$ element where n is the maximum number of units of memory used when implementing the program.

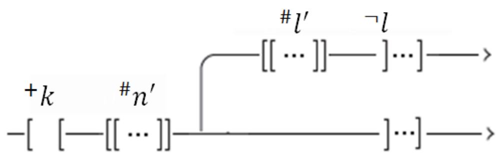

Fig. 3. Joint commit parallel threads

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 crucial property for our correctness proofs. It states that the typing environment combined with the type of its term always produces a 'well-formed' structure.

Theorem 1 (Type judgment property). *If $n \vdash e : T$ and $n \geqslant 0$, then* $\mathrm{sim}(^{+}n, T) = {}^{\sharp}m$ *for some m (i.e.* $\mathrm{sim}(^{+}n, T)$ *has the form of single element with tag $\sharp$) and $m \geqslant n$ where* $\mathrm{sim}(T_1, T_2) = \mathrm{seq}(\mathrm{jc}(S_1, S_2))$ *with S_1, S_2 is T_1, T_2 without ρ.*

Proof. By induction on the typing rules in Table 2.

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

$\mathrm{sim}(\,^{+}(n_1+n_2)\,,S) = \,^{\sharp}m$ and $m \geqslant m_1 + m_2$. We have

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

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

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

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

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

- 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:

$$6 \vdash \mathbf{onacid}(5);\mathbf{commit};\mathbf{commit};\mathbf{commit};\mathbf{commit} : \,^{\sharp}5\,\,^{\neg}1\,\,^{\neg}1\,\,^{\neg}1 \qquad (1)$$

Then, by applying the rule T-SPAWN, we have:

$$6 \vdash e^5_5 : (\,^{\sharp}5\,\,^{\neg}1\,\,^{\neg}1\,\,^{\neg}1\,)^{\rho} \qquad (2)$$

Now, we want to use T-MERGE and we need a term such that its type matches the type of e^5_5. We find that e^6_{10} satisfies this condition since its type can be derived using T-SEQ, T-ONACID, T-COMMIT as follows:

$$6 \vdash e^6_{10} : \,^{\neg}1\,\,^{\sharp}6\,\,^{\neg}1\,\,^{\sharp}7\,\,^{\neg}1$$

By applying T-PREP, we have a matching type with (2). So we can apply T-MERGE to get the type for e^5_{10} as follows:

$$6 \vdash e^5_{10} : (\,^{\sharp}5\,\,^{\neg}2\,\,^{\sharp}6\,\,^{\neg}2\,\,^{\sharp}7\,\,^{\neg}2\,)^{\rho}$$

With $-3 \vdash e_4^4 : {}^{+}3$, we can apply T-JC to get the type of e_{10}^4 as follows:

$$3 \vdash e_{10}^4 : {}^{\sharp}11\ {}^{\neg}2\ {}^{\sharp}7\ {}^{\neg}2 \tag{3}$$

as

$$\begin{aligned}&\mathrm{jc}({}^{+}3\,,\ {}^{\sharp}5\ {}^{\neg}2\ {}^{\sharp}6\ {}^{\neg}2\ {}^{\sharp}7\ {}^{\neg}2\,) = \mathrm{jc}(\mathrm{seq}({}^{\sharp}3\,), \mathrm{seq}({}^{\sharp}(5+3*2)\ {}^{\sharp}6\ {}^{\neg}2\ {}^{\sharp}7\ {}^{\neg}2\,))\\ &= \mathrm{jc}({}^{\sharp}3\,,\ {}^{\sharp}11\ {}^{\neg}2\ {}^{\sharp}7\ {}^{\neg}2\,) = \mathrm{seq}({}^{\sharp}11\ {}^{\neg}2\ {}^{\sharp}7\ {}^{\neg}2\,) = {}^{\sharp}11\ {}^{\neg}2\ {}^{\sharp}7\ {}^{\neg}2\end{aligned}$$

Similarly, we can calculate the type for the term on line 3: $3 \vdash e_3^3 : (\,{}^{\sharp}4\ {}^{\neg}1\ {}^{\neg}1\,)^{\rho}$. This type matches (3) so we can apply T-MERGE and get the type for e_{10}^3 as follows:

$$3 \vdash e_{10}^3 : {}^{\sharp}15\ {}^{\neg}3\ {}^{\sharp}7\ {}^{\neg}3$$

Type for line 1 to line 2 is: $-3 \vdash e_2^1 : {}^{+}1\ {}^{+}2$. Apply T-JC for e_2^1 and e_{10}^3 we get:

$$0 \vdash e_{10}^1 : {}^{\sharp}24$$

since

$$\begin{aligned}&\mathrm{jc}({}^{+}1\ {}^{+}2\,,\ {}^{\sharp}15\ {}^{\neg}3\ {}^{\sharp}7\ {}^{\neg}3\,) = \mathrm{jc}(\mathrm{seq}({}^{+}1\ {}^{\sharp}2\,), \mathrm{seq}({}^{\sharp}21\ {}^{\sharp}7\ {}^{\neg}3\,))\\ &= \mathrm{jc}({}^{+}1\ {}^{\sharp}2\,,\ {}^{\sharp}21\ {}^{\neg}3\,) = \mathrm{jc}(\mathrm{seq}({}^{\sharp}3\,), \mathrm{seq}({}^{\sharp}24\,)) = \mathrm{jc}({}^{\sharp}3\,,\ {}^{\sharp}24\,) = {}^{\sharp}24\end{aligned}$$

The program is well-typed and the maximum memory that it needs in this case is 24 units. In the next section, we will show the soundness of the type system.

5 Correctness

To prove the correctness of our type system, we need to show that a well-typed program does not use more memory 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 total number of units of memory used for logs by the program in the global environment is always smaller than or equal to n.

A state is a pair Γ, P where $\Gamma = \{p_1 : E_1, \ldots, p_k : E_k\}$ and $P = \coprod_1^k p_i(e_i)$. We say Γ 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 number of units of memory that have been created or copied in thread p_i, and S_i represents the number of units of memory that will be created when executing e_i. Therefore, total memory used by thread p_i is expressed by $\mathrm{sim}({}^{+}[\![E_i]\!]\,, S_i)$, where the sim function is defined in Theorem 1. We will show that $\mathrm{sim}({}^{+}[\![E_i]\!]\,, S) = {}^{\sharp}n$ for some n. We denote this value n as $[\![E_i, S_i]\!]$. Then, the total memory of logs of a program state, included in Γ and the potential logs that will be created when executing the remaining program, denoted by $[\![\Gamma, P]\!]$, and is defined by:

$$[\![\Gamma, P]\!] = \sum_{i=1}^{k} [\![E_i, S_i]\!]$$

Since $\llbracket \Gamma, P \rrbracket$ represents the maximum number of units *from* the current state and $\llbracket \Gamma \rrbracket$ is the number of units *in* the current state, we have the following lemma.

Lemma 1. *If $\Gamma \models P$, then $\llbracket \Gamma, P \rrbracket \geqslant \llbracket \Gamma \rrbracket$.*

Proof. By the definition of $\llbracket \Gamma, P \rrbracket$ and $\llbracket \Gamma \rrbracket$, we only need to show $\llbracket E_i, S_i \rrbracket \geqslant \llbracket E_i \rrbracket$ for all i. This follows from Theorem 1. □

Lemma 2 (Subject reduction). *If $\Gamma \models P$ and $\Gamma, P \Rightarrow \Gamma', P'$, then $\Gamma' \models P'$ and $\llbracket \Gamma, P \rrbracket \geqslant \llbracket \Gamma', P' \rrbracket$.*

Proof. The proof is done by checking one by one all the semantics rules in Table 1. For each rule, we need to prove two parts: (i) $\Gamma' \models P'$ and (ii) $\llbracket \Gamma, P \rrbracket \geqslant \llbracket \Gamma', P' \rrbracket$.

- For S-SPAWN, by the assumption we have $\Gamma \models P$ and $\{p : E\} \in \Gamma$ and $P = P_1 \parallel p(\mathbf{spawn}(e_1); e_2)$ and $\llbracket E \rrbracket \vdash \mathbf{spawn}(e_1); e_2 : S$ for some P_1, S. By the definition of function *spawn* we have $\Gamma' = \Gamma \cup \{p' : E\}$ and by the rule S-SPAWN, we have $P' = P_1 \parallel p'(e_1) \parallel p(e_2)$.
  - For (i), by definition of $\models$ we only need to prove that $\llbracket E \rrbracket \vdash e_1 : S_1$ and $\llbracket E \rrbracket \vdash e_2 : S_2$ for some S_1, S_2 because P' differs from P only in the terms e_1 and e_2 of p' and p.
    Since $\llbracket E \rrbracket \vdash \mathbf{spawn}(e_1); e_2 : S$ and $\mathbf{spawn}(e_1); e_2$ can only be typed by T-MERGE, we have $\llbracket E \rrbracket \vdash \mathbf{spawn}(e_1) : S_1'$ and $\llbracket E \rrbracket \vdash e_2 : S_2$ for some S_1' and S_2. By T-SPAWN we have $\llbracket E \rrbracket \vdash e_1 : S_1$ where $S_1' = \mathrm{join}(S_1)^{\rho}$. So (i) holds and we also have $S = \mathrm{merge}(S_1, S_2)$.
  - For (ii), first we denote $n_i = \mathrm{seq}(\mathrm{jc}(\,^{+}\llbracket E \rrbracket\,, S_i))$ with $i = 1, 2$. We have:

    $$\begin{aligned}
    &\llbracket \Gamma, P \rrbracket - \llbracket \Gamma', P' \rrbracket \\
    &= \llbracket E, S \rrbracket - (\llbracket E, S_1 \rrbracket + \llbracket E, S_2 \rrbracket) && \text{def. of } \llbracket . \rrbracket \\
    &= \mathrm{seq}(\mathrm{jc}(\,^{+}\llbracket E \rrbracket\,, S)) - (n_1 + n_2))) && \text{def. of } \llbracket . \rrbracket \\
    &= \mathrm{seq}(\mathrm{jc}(\,^{+}\llbracket E \rrbracket\,, \mathrm{merge}(S_1, S_2))) - (n_1 + n_2) && S = \mathrm{merge}(S_1, S_2) \\
    &= (n_1 + n_2) - (n_1 + n_2) && \text{properties of } S_1, S_2 \\
    &= 0
    \end{aligned}$$

    So (ii) holds.
- For S-TRANS, similar to the previous case, by the assumption we have $\Gamma \models P$, $\{p : E\} \in \Gamma$ and $P = P_1 \parallel p(\mathbf{onacid}(n); e)$ and $\llbracket E \rrbracket \vdash \mathbf{onacid}(n); e : S$ for some P_1, S. By the definition of function *start* we have $\Gamma' = \Gamma \backslash \{p : E\} \cup \{p : E'\}$ where $\llbracket E' \rrbracket = \llbracket E \rrbracket + n$. By the rule S-TRANS, we have $P' = P_1 \parallel p(e)$.
  For (i), by definition of $\models$ we need to prove that $\llbracket E \rrbracket + n \vdash e : S'$ for some S'. Since $\mathbf{onacid}(n); e$ can be typed by T-SEQ or T-JC, we have two cases to consider:
  - By the rule T-SEQ, $\llbracket E \rrbracket + n \vdash e : S'$ follows immediately by the typing rule when the first component is $-n \vdash \mathbf{onacid}(n) : \,^{+}n\,$. So (i) holds.

For (ii), we have:

$$\begin{aligned}
&\llbracket \mathit{\Gamma}, P\rrbracket - \llbracket \mathit{\Gamma}', P'\rrbracket && \\
&= \llbracket E, S\rrbracket - \llbracket \llbracket E\rrbracket + n, S'\rrbracket && \text{def. of } \llbracket . \rrbracket \\
&= \mathrm{seq}(\mathrm{jc}(\,{}^{+}\llbracket E\rrbracket\,,\ {}^{+}n\, S')) - \mathrm{seq}(\mathrm{jc}(\,{}^{+}(\llbracket E\rrbracket + n)\,, S')) && \text{def. of } \llbracket . \rrbracket \\
&= \mathrm{seq}(\,{}^{+}\llbracket E\rrbracket\,,\ {}^{+}n\, S') - \mathrm{seq}(\,{}^{+}(\llbracket E\rrbracket + n)\,, S') && \neg \notin S' \\
&= \mathrm{seq}(\,{}^{+}\llbracket E\rrbracket\,,\ {}^{+}n\, S') - \mathrm{seq}(\,{}^{+}\llbracket E\rrbracket\ {}^{+}n\,, S') && \text{Def. 4} \\
&= 0 && \text{Def. 4}
\end{aligned}$$

So (ii) holds.

- By the rule T-JC, similarly, $\llbracket E\rrbracket + n \vdash e : S'$ follows immediately by the typing rule when the first component is $-n \vdash \mathbf{onacid}(n) : {}^{+}n$. So (i) holds.

  For (ii), we have:
  $$\begin{aligned}
  &\llbracket \mathit{\Gamma}, P\rrbracket - \llbracket \mathit{\Gamma}', P'\rrbracket && \\
  &= \llbracket E, S\rrbracket - \llbracket \llbracket E\rrbracket + n, S'\rrbracket && \text{def. of } \llbracket . \rrbracket \\
  &= \mathrm{sim}(\,{}^{+}\llbracket E\rrbracket\,, \mathrm{jc}(\,{}^{+}n\,, S')) - \mathrm{sim}(\,{}^{+}(\llbracket E\rrbracket + n)\,, S') && S = \mathrm{jc}(\,{}^{+}n\,, S') \\
  &= \mathrm{seq}(\mathrm{jc}(\,{}^{+}\llbracket E\rrbracket\,, \mathrm{jc}(\,{}^{+}n\,, S'))) - \mathrm{seq}(\mathrm{jc}(\,{}^{+}(\llbracket E\rrbracket + n)\,, S')) && \text{def. of jc} \\
  &= \mathrm{seq}(\mathrm{jc}(\,{}^{+}\llbracket E\rrbracket\,, \mathrm{jc}(\,{}^{+}n\,, S'))) - \mathrm{seq}(\mathrm{jc}(\,{}^{+}\llbracket E\rrbracket\,, \mathrm{jc}(\,{}^{+}n\,, S'))) && \text{def. of jc} \\
  &= 0 &&
  \end{aligned}$$

  So (ii) holds.

- For S-COMM, similarly, by the assumption we have $\mathit{\Gamma} \models P$ and $\coprod_1^k (p_i : E_i) \in \mathit{\Gamma}$ and $P = P_1 \cup \coprod_1^k p_i(\mathbf{commit}; e_i)$ and $\llbracket E_i\rrbracket \vdash \mathbf{commit}; e_i : S_i$ for some P_1, E_i, S_i, $i = 1..k$. By the definition of function **commit** we have $\coprod_1^k (p_i : E_i') \in \mathit{\Gamma}'$. By the rule S-COMM we have $P' = P_1 \cup \coprod_1^k p_i(e_i)$ where $\llbracket E_i'\rrbracket = \llbracket E_i\rrbracket - n$.

  - For (i), by definition of $\models$ we need to prove that $\llbracket E_i'\rrbracket \vdash e_i : S_i'$ for some S_i'. As $\mathbf{commit}; e_i$ is typed by T-SEQ and the **commit** is a joint commit, this follows by the typing rule when the first expression is $+n \vdash \mathbf{commit} : {}^{\neg}1$ and we have $S_i = {}^{\neg}1\, S_i'$ where S_i' is the type of e_i. So (i) holds.

- For (ii) we have:

$$\begin{aligned}
&[\![\Gamma, P]\!] - [\![\Gamma', P']\!] \\
&= \sum_1^k([\![E_i, S_i]\!]) - \sum_1^k([\![E'_i, S'_i]\!]) && \text{def. of } [\![.]\!] \\
&= \sum_1^k([\![E_i,\ {}^{\neg}1\, S'_i]\!] - [\![E'_i, S'_i]\!]) && S_i = {}^{\neg}1\, S'_i \\
&= \sum_1^k(\mathrm{seq}(\mathrm{jc}(\ {}^{+}[\![E_i]\!]\,,\ {}^{\neg}1\, S'_i)) - \mathrm{seq}(\mathrm{jc}(\ {}^{+}[\![E'_i]\!]\,, S'_i))) && \text{def. of } [\![.]\!] \\
&= \sum_1^k(\mathrm{seq}(\mathrm{jc}(\ {}^{+}[\![E_i]\!]\,,\ {}^{\neg}1\, S'_i)) - \mathrm{seq}(\mathrm{jc}(\ {}^{+}([\![E_i]\!] - n)\,, S'_i))) && [\![E'_i]\!] = [\![E_i]\!] - n \\
&= \sum_1^k(\mathrm{seq}(\mathrm{jc}(\ {}^{+}([\![E_i]\!] - n)\,,\ {}^{\sharp}n\, S'_i)) - \mathrm{seq}(\mathrm{jc}(\ {}^{+}([\![E_i]\!] - n)\,, S'_i))) && \text{def. of jc} \\
&\geqslant 0
\end{aligned}$$

So (ii) holds.

- For S-COND, by the assumption we have $\Gamma \models P$, $\{p : E\} \in \Gamma$. $P = P_1 \parallel p((e_1 + e_2); e)$ and $[\![E]\!] \vdash (e_1 + e_2); e : S$ for some E, S. By the rule S-COND we have $\Gamma' = \Gamma$ and $P' = P_1 \parallel p(e_i; e), i = 1, 2$.
For (i), by definition of $\models$ we need to prove that $[\![E]\!] \vdash e_i; e : S'$ with $i = 1, 2$. As the external transactional behaviors of e_1 and e_2 are the same, we have $[\![E]\!] \vdash e_i; e : S'$. So (i) holds.
For (ii), first we denote $\mathrm{alt}(S_1, S_2) = S_1 \odot S_2$ and $\mathrm{merge}(S_1, S_2) = S_1 \otimes S_2$. We have two cases to consider:
  - In the case $[\![E]\!] \vdash e_i : S_i$, then: $[\![E]\!] \vdash (e_1 + e_2); e : S_1 \odot S_2 T$ in which, T is the type of e and the function *alt* is defined in Definition 7. The function $\mathrm{alt}(S_1, S_2)$ returns the maximum sequence from S_1 and S_2 (choosing larger one from *corresponding* $\sharp$ *components* of S_1 and S_2) which has the *same structure* with S_1 and S_2. In addition, $S_1 \odot S_2 T$, $S_i T$ must be sequences of tagged numbers in order to make their corresponding expressions well-typed. , we have:

$$\begin{aligned}
&[\![\Gamma, P]\!] - [\![\Gamma', P']\!] \\
&= [\![E, S]\!] - [\![E, S']\!] && \text{def. of } [\![.]\!] \\
&= \mathrm{seq}(\mathrm{jc}(\ {}^{+}[\![E]\!]\,, (S_1 \odot S_2)T)) - \mathrm{seq}(\mathrm{jc}(\ {}^{+}[\![E]\!]\,, S_i T)) && \text{def. of } [\![.]\!] \\
&\geqslant 0 && \text{def. of alt}
\end{aligned}$$

    So (ii) holds.
  - In the case $[\![E]\!] \vdash e_i : S_i^{\rho}$, then: $[\![E]\!] \vdash (e_1 + e_2); e : (S_1 \odot S_2)^{\rho} T$.
    As in the former case, $\mathrm{alt}(S_1, S_2)$ chooses larger one from corresponding $\sharp$ components of S_1 and S_2. Furthermore, the merge function is defined

in Definition 6, which sums the corresponding $\sharp$ components of two sequences. Therefore, we have:

$$\begin{aligned}
&[\![\Gamma, P]\!] - [\![\Gamma', P']\!] \\
&= [\![E, S]\!] - [\![E, S']\!] && \text{def. of } [\![.]\!] \\
&= \text{seq}(\text{jc}(\ {}^{+}[\![E]\!]\ , S)) - \text{seq}(\text{jc}(\ {}^{+}[\![E]\!]\ , S')) && \text{def. of } [\![.]\!] \\
&= \text{seq}(\text{jc}(\ {}^{+}[\![E]\!]\ , (S_1 \odot S_2) \otimes T)) - \text{seq}(\text{jc}(\ {}^{+}[\![E]\!]\ , S_i \otimes T)) && \text{def. of } \odot, \otimes \\
&\geqslant 0 && \text{def. of alt}
\end{aligned}$$

So (ii) holds. □

Now we come to the correctness property of our type system. A well-typed program will not use more units of memory than the one stated in its type.

Theorem 2 (Correctness). *Suppose* $0 \vdash e : {}^{\sharp}n$ *and* $p_1 : \epsilon, p_1(e) \Rightarrow^* \Gamma, P$, *then* $[\![\Gamma]\!] \leqslant n$.

Proof. For the starting environment we have: $[\![p_1 : \epsilon, p_1(e)]\!] = \text{sim}(0, {}^{\sharp}n\ ) = {}^{\sharp}n$. So from Lemmas 2 and Theorem 1, the theorem holds by induction on the length of transitions. □

6 Type inference

We have implemented a prototype tool[3] in F# language that can check and infer types for well-typed programs. Listing 1.3 is the main `infer` function, which is similar to the algorithm in [15], that takes a `term` and an 'environment' `headseq` in line 3. The differences are in the implementation of other functions such as seq, jc and merge. Listing 1.2 shows the simplified implementation of seq in Definition 4 where in line 1 `T.P`, `T.M`, `T.X`, `T.J` denote tags $+, -, \sharp, \neg$, respectively. The function finds the patterns defined in Definition 4 and simplifies them.

A program or term is encoded as a list of branches and leaves. The algorithm travels the program from the first statement to the last statement and makes recursive calls when hitting **spawn** statements. When we reach the end of `term` in line 5 we need to compact the result type by calling the seq function (Definition 4). Otherwise, we check if the next statement `x` is a branch or a leaf. If it is a leaf we just update the `headseq` with the new leaf (line 10) and then repeat the inference process. In case `x` is a branch (line 12), we infer 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 tested our tool on several examples. The code contains automated tests and all test cases are passed, i.e., actual results are equal to our expected ones.

Listing 1.2. `seq` function used in `infer`

```
1  type T = P = 0 | M = 1 | X = 2 | J = 3
```

[3] Available at https://github.com/truonganhhoang/tm-infer

```
let rec seq (lst : TagSeq) : TagSeq =
  match lst with
  | [] -> []
  | (_,0)::xs -> seq xs
  | (T.X,l)::(T.X,m)::xs -> seq ((T.X,max l m)::xs)
  | (T.M,l)::(T.M,m)::xs -> seq ((T.M,l+m)::xs)
  | (T.P,l)::(T.M,m)::xs -> seq ((T.X,l)::(T.M,m-1)::xs)
  | (T.P,l)::(T.X,n)::(T.M,m)::xs -> seq ((T.X,l+n)::(T.M,m-1)
     ::xs)
  | x::xs -> x::(seq xs)
```

Listing 1.3. Main type inference algorithm

```
type TagNum = T * int
type Tree =  | Branch of Tree list | Leaf of TagNum
let rec infer (term: Tree list) (headseq: TagNum list) =
  match term with
  | [] -> seq headseq (* simplifies the result *)
  | x::xs ->
    match x with
    | Leaf tagnum ->
      (* expand the head part *)
      let new_head = seq (List.append headseq [tagnum]) in
      infer xs new_head
    | Branch br -> (* a new thread *)
      (* infer the child and parent tail *)
      let child = join (infer br []) in
      let parent = join (infer xs []) in
      (* merge the child and the parent tail *)
      let tailseq = seq (merge child parent) in
      (* join commit with the head *)
      jc headseq tailseq
```

7 Conclusion

We have presented a generalized language whose main features are a mixing of multi-threading and nested transactions. A key new feature in the language is that it contains size information of transaction logs, which in practice can be automatically synthesized by identifying shared variables in the transactions. Then, based on the size information, we can infer statically the maximum memory units needed for transaction logs. Although the language is not easy to use directly as it is designed to give control power to programmers on the behavior of transactions, the analysis we presented can be applied to popular transactional languages as they are special cases of our presented language. The type system of our analysis looks similar to the ones in our previous works, but the semantics of type elements and typing rules are novel and the maximum memory obtained from well-typed programs is of practical value.

We are extending our tool to take real world transactional programs as input and produce the worst execution scenarios of the program where maximum log memory are used.

References

1. Elvira Albert, Puri Arenas, Jesús Correas Fernández, Samir Genaim, Miguel Gómez-Zamalloa, Germán Puebla, and Guillermo Román-Díez. Object-sensitive cost analysis for concurrent objects. *Softw. Test., Verif. Reliab.*, 25(3):218–271, 2015.
2. Elvira Albert, Jesús Correas Fernández, and Guillermo Román-Díez. Peak cost analysis of distributed systems. In Markus Müller-Olm and Helmut Seidl, editors, *Static Analysis - 21st International Symposium, SAS 2014*, volume 8723 of *LNCS*, pages 18–33. Springer, 2014.
3. Elvira Albert, Samir Genaim, and Miguel Gomez-Zamalloa. Heap space analysis for Java bytecode. In *ISMM '07*, New York, NY, USA, 2007. ACM.
4. 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.
5. Marc Bezem, Dag Hovland, and Hoang Truong. A type system for counting instances of software components. *Theor. Comput. Sci.*, 458:29–48, 2012.
6. Vctor Braberman, Diego Garbervetsky, Samuel Hym, and Sergio Yovine. Summary-based inference of quantitative bounds of live heap objects. *Science of Computer Programming*, 92, Part A:56 – 84, 2014. Special issue on Bytecode 2012.
7. 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.
8. 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.
9. Jan Hoffmann and Zhong Shao. Automatic static cost analysis for parallel programs. In Jan Vitek, editor, *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015*, volume 9032 of *LNCS*, pages 132–157. Springer, 2015.
10. Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of POPL '03*. ACM, January 2003.
11. John Hughes and Lars Pareto. Recursion and dynamic data-structures in bounded space: Towards embedded ML programming. *SIGPLAN Notices*, 34(9), 1999.
12. Suresh Jagannathan, Jan Vitek, Adam Welc, and Antony Hosking. A transactional object calculus. *Sci. Comput. Program.*, 57(2):164–186, August 2005.
13. Thi Mai Thuong Tran, Martin Steffen, and Hoang Truong. Compositional static analysis for implicit join synchronization in a transactional setting. In *SEFM 2013*, volume 8137 of *LNCS*, pages 212–228. Springer Berlin Heidelberg, 2013.
14. 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 Antonio Cerone and Stefan Gruner, editors, *Sixth IEEE International Conference on Software Engineering and Formal Methods, SEFM 2008, Cape Town, South Africa, 10-14 November 2008*, pages 259–267. IEEE Computer Society, 2008.

15. Anh-Hoang Truong, Dang Van Hung, Duc-Hanh Dang, and Xuan-Tung Vu. A type system for counting logs of multi-threaded nested transactional programs. In Nikolaj Bjørner, Sanjiva Prasad, and Laxmi Parida, editors, *Distributed Computing and Internet Technology - 12th International Conference, ICDCIT 2016, Proceedings*, volume 9581 of *LNCS*, pages 157–168. Springer, 2016.
16. 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 2012, SoICT '12, Halong City, Quang Ninh, Viet Nam, August 23-24, 2012*, pages 21–30, 2012.
17. B. Wegbreit. Mechanical program analysis. *Communications of the ACM*, 18(9), 1975.