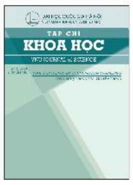
VNU Journal of Science:
Computer Science and Communication Engineering

Journal homepage: http://www.jcsce.vnu.edu.vn/index.php/jcsce



Original Article

Estimate the Memory Bounds Required by Shared Variables in Software Transactional Memory Programs

Nguyen Ngoc Khai[1,2] , Truong Anh Hoang[1,*], Dang Duc Hanh[1]

[1]VNU University of Engineering and Technology, 144 Xuan Thuy, Cau Giay, Hanoi, Vietnam
[2]Hanoi University of Natural Resources and Environment, 41A Phu Dien, Nam Tu Liem, Hanoi, Vietnam

Received 01 December 2020
Revised 07 June 2021; Accepted 31 July 2021

Abstract: Estimating memory required by complex programs is a well-known research topic. In this work, we build a type system to statically estimate the memory bounds required by shared variables in software transactional memory (STM) programs. This work extends our previous works with additional language features such as explicitly declared shared variables, introduction of primitive types, and allowing loop body to contain any statement, not required to be well-typed as in our previous works. Also, the new type system has better compositionality compared to available type systems.

Keywords: Type system, software transactional memory, memory bound.

1. Introduction

Usually, the programmers have to take care of the resource usage of the program. However, this task is more difficult when multi-threaded programs are written. Synchronizing threads in multi-threaded programs is often handled by lock-based mechanism [1]. However, this mechanism is often error-prone with lock errors such as deadlock or livelock.

As an alternative approach for lock-based mechanism, the STM [2, 3] makes it easier for multi-threaded programming since the programmer does not have to pay attention to the lock errors and the management of locks, so they can focus on the business functions of the programs.

However, there is a disadvantage of the STM mechanism that consumes more memory than the traditional mechanisms. The results of the experiment in [4] have shown that the STM program can consume more resources than conventional programs up to 22 times (3 times on average). This is due to when a thread is spawned, it will duplicate shared variables from the parent thread so it can use these variables

[*]Corresponding author.
E-mail address: hoangta@vnu.edu.vn

https://doi.org/10.25073/2588-1086/vnucsce.280

independently. These variables will be released when the thread is synchronized with its parent thread. Therefore, estimating the memory used by shared variables in the STM programs is necessary to optimize the program and reduce the risks of being out of memory runtime exceptions.

Due to the implicit synchronization between threads, and nesting of transactions (detailed in Sections 2.1, 3.2), estimating maximum memory for shared variables is a non-trivial problem. The current work on this problem often uses dynamic methods [4]. That is, they execute the program and use tools to measure the resources consumed by the program. They do this several times, then take the average or maximum value of the measurements. This method is easy to do, but it gives only approximate results. The method that we propose in this work is a static method based on type theory. The results are proven mathematically to ensure correctness, so our results are reliable.

In the previous works, we have developed several type systems to estimate the resource usage of the STM programs [5-8], in which we assume that the resources are the parameters provided by the user. Users still have to calculate manually the resources of each transaction, so this work is still generalized, not completely automated.

This work is extended from our previous work [6], in which we add more features to the language to make it less abstract than our previous works and closer to programming languages available. We offer some basic features of the STM mechanism in practical programming languages. Specifically, we adjusted some of the syntax and semantics as follows: i) Shared variables are explicitly declared. Expressions and values are more strictly expressed; ii) Expressions in the loop body can contain any statement and are not required to be a well-typed expression.In the previous language [6], the loop body expression must be a well-typed one and contain no spawn statement. This improvement makes the language much stronger but leads to many challenges and difficulties for typing; iii) The formula to calculate the resource consumption of the program and the auxiliary functions such as onacid, `new` have also been changed to be more suitable; iv) The semantic rules of language, environment, and status of the program are redefined in more details. This makes their implementation easier in practice.

We prove that the bound is sharp for all statements except the conditional statement.

Formally, we adapt the type rules defined in our previous works [5, 6] to the new STM language in order to make the typing more accurate. Several rules such as T-NEW and T-BOOL are combined to make them more general. We adjusted the rules T-COND and T-WHILE to type better conditional and loop expressions. The main contributions of this paper are summarized as follows:

- An improved STM language whose features are closer to current practical language;
- A type system to estimate the memory bound required by shared variables in the STM programs and discusses the sharpness of these boundaries.

The rest of the article is organized as follows: In the next section, we present an overview of the STM mechanism with an illustrative example of a research context, and discuss related works. In Section 3, we describe our STM language. Section 4 is about the type system. In Section 6, we discuss the sharpness of the type system and typing for the example program. Section 7 provides the conclusion and future works.

2. Preliminary

To describe the problem more clearly, in this section, we explain some of the typical features of STM programs and illustrate our context research with an example. We survey related works on the STM mechanism and the estimation of resources used by the programs.

2.1. Characteristics of STM-based Programs

The STM mechanism is an alternative to lock-base mechanism to control concurrency programs. It has several typical features as follows:

- *Complex nested transactions:* A transaction can be spawned within another open transaction. When a transaction is spawned within another transaction, we call the first transaction a parent transaction, the latter a child transaction. Child transactions must be committed before the parent transaction. Within a thread, if a transaction commits without referring to other threads we call *local commit*. When a transaction commits, the threads spawned within it also have to be synchronized. We call this *joint commit*.

- *Duplicate shared variables:* When a new thread is spawned, it copies the shared variables of the parent thread into their own variables for use independently. After the transaction is completed, these threads compare the values of these shared variables to synchronize. If there is no conflict, they are synchronized, otherwise, they are canceled or roll-back. This depends on the design of the programming language. When using this mechanism, the program does not need to use locks so it avoids lock errors. However, copying shared variables of threads causes the program to consume more memory.

- *Implicit synchronization:* For some actual programming languages, the programmer may not need to write statements to synchronize between threads, their compiler automatically adds those statements during the execution. This is called implicit synchronization. It is convenient for the programmer because they do not care about synchronization between threads. However, this also creates other constraints, we will analyze more in detail in section 3.2.

Because of this implicit synchronization, we need to analyze the program at the semantic level of the language rather than at the source code level. In addition, these programs have some nested transactions, threads run in parallel but are not independent. This makes estimating the memory consumed by these programs really complex. Our solution to this problem is to build a type system. This is a static program analysis method that has been used in many works [9].

The main purpose of this type system is to estimate the maximum memory required by shared variables in STM programs. However, users can develop them to suit their purpose, for example, the calculation of the cost of memory, CPU, network bandwidth resources, or the number of gases required for smart contracts in Ethereum, etc.

2.2. An Illustrative Example

To describe the problem, we consider an STM program segment as shown in Listing 1.

In this code, the statement `onacid` is used to open a transaction; The statement `commit` is used to commit a transaction or joint commit transactions between the parent thread and its child threads. The statement `spawn` is used to spawn a new thread; The variables declared with the keyword `shared` in front are shared variables between the parent and child threads.

Under the STM mechanism, these variables are cloned by child threads from their parent threads to become their own variables; the remaining statements such as declaring and initializing variables, conditional statement, loop statement are similar to other common languages.

The behavior of this program is described in the Figure 1. The symbol [describes the operations that open a transaction; the symbol] as the closing of a transaction or joint commit between the parent thread and child threads;

The dashed square i) represents the joint commits of the parent and child threads; The point marked by the symbol ♦ represents the branch of the conditional statement: they can turn to branch e_1 or branch e_2; the dashed square ii) represents the closing of the transaction outside of the conditional statement. Visually we realize that the maximum memory the program needs to consume by the STM mechanism can be at points ①, ②, or ③.

Listing 1. Example of a multi-threaded

```
1  int a=0; b=1;
   //Local variables
2  onacid;
   //Open first transaction
3   shared int x1=0;
4   spawn{
5    onacid;
6     shared int y1=0;
7     shared int y2=0;
8     shared int y3=0;
      //Do something
9    commit;
10   if (a>b) then{
11    onacid;
12     shared int y4=0;
13     shared bool y5=0;
       //Do something
14     onacid;
15      shared bool y6=0;
        //Do something
24     commit;
25   }
26   onacid;
27    shared bool y10=0;
      //Do something
28   commit;
29   commit; //Commit for onacid at
                line 11 or line 19
30   commit; //Joint commit with
                onacid at line 2
31  }
32  onacid;
33   share int x2=0;
     //Do something
34  commit;
35 commit;
   //Commit with onacid at line 2
```

Assuming that a bool variable needs 1 memory unit, an integer variable needs 2 memory units, the maximum memory required for shared variables in this program at each time is:

- At point ①, the maximum memory required is 12 units as follows:
  - Thread 0 requires 4 units in which 2 units are for the variable `x1` in the first transaction, another 2 units for the variable `x2` in the 2nd transaction.
  - Thread 1 requests 8 units in which 2 units are required because thread 1 clones variable `x1` from thread 0 by STM mechanism, 6 units for variables `y1`, `y2`, and `y3`.
- At point ②, the maximum memory required is 10 units as follows:
  - Thread 0 requires 4 units for the variables `x1` and `x2`.
  - Thread 1 requires 6 or 5 units as follows:
    + 2 units for the variable `x1`, which is cloned from thread 0 by the STM mechanism.
    + If the statement `if` turns to branch p_1: 4 units for the variables `y4`, `y5`, and `y6`.

    In this case, at point ② the program consumes 10 units.

    + If the statement `if` turns to branch p_2: 3 units for variables `y7`, `y8`, and `y9`.

    In this case, at point ② the program consumes 9 units.
- At point ③, the maximum memory required is 8 units as follows:
  - Thread 0 requires 4 units for the variables `x1` and `x2`.
  - Thread 1 requires 6 or 4 units:

    +2 units for the variable `x1` (it is cloned from thread 0).

    +If the statement `if` turns to branch p_1: 4 units for the variables `y4`, `y5`, and `y10`.

  In this case, at point ③, the program consumes 10 units.

    +If the statement `if` turns to branch p_2: 2 units for the variables `y7` and `y10`.

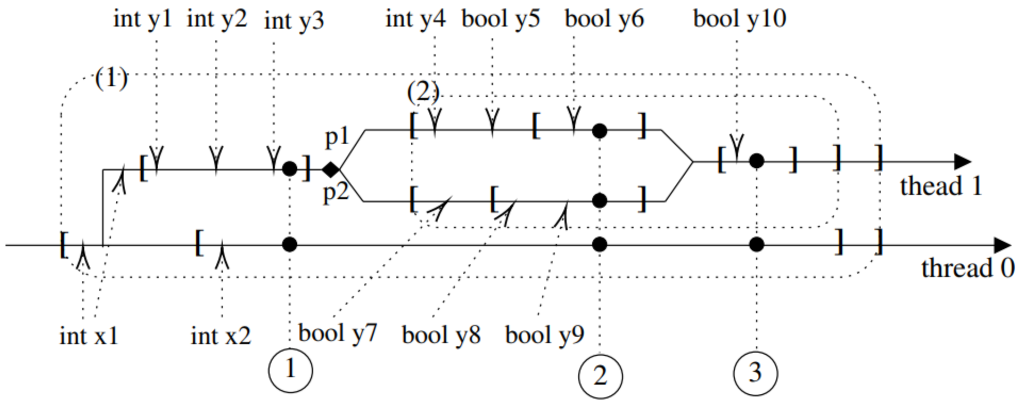


Figure 1. Describe the behavior of the program in Listing.

In this case, at point ③ the program consumes 8 units. According to the above analysis, we realize that the maximum memory required by the program is 12 units at point ①.

2.3. Related Works

STM mechanism: The STM mechanism has been studied for a long time, in which [10] can be considered as the first proposal of this approach. In this work, there are limits that transactions are static, the data set is already known. In work [13], Herlihy *et al.* have overcome the limits on the above static transactions by dynamic transactions in libraries of the Java language. Then there are many studies that continue to be developed and implemented in practice, such as in Java language [11-14], Haskell language [15], C++ language [16]. The above works showed that the STM mechanism is very potential and practical.

The estimation of resources used by the program: Estimating resources used by the program is an important issue, and it has been of interest to many scientists. In [17], the authors present an overview of resource estimation issues in software engineering. In [18], the authors present a method to find the upper bounds for the size of permutation codes via linear programming. In [19] Jan Hoffmann *et al.* propose a method of automatic resource bound analysis for the OCaml program. He used techniques to parameterize resources to deduce the time, memory, and energy needed by the program. In [4], Klein *et al.* found that the energy consumption of the software transactional memory mechanism was higher than the lock-based mechanism of up to 22 times (average more than 3 times).

In [20], the authors introduce the first automatic analysis for deriving bounds on the worst-case evaluation costs of parallel first-order functional programs. The analysis is performed by a novel type system for amortized resource analysis. The main innovation is a technique that separates the reasoning about sizes of data structures and evaluation cost within the same framework. In recent years, for the Ethereum blockchain platform, in [21], the authors design and implement a tool to automatically infer sound gas upper bounds for smart contracts. In Ethereum, gas (in Ether, a cryptographic currency like Bitcoin) is the execution fee compensating the computing resources of miners for running smart contracts [22]. In [22], the authors analyzed the Solidity language and its compilation. Then they propose a tool for automatically locating gas-costly patterns by analyzing smart contracts' bytecodes.

Our work focuses on multi-threaded languages, using STM mechanisms. We focus on solving the problems of complex nested transactions, implicit synchronization between threads. Our method is static program analysis, based on the type system.

In works [7, 23], we have built a type system for core STM language, which only contains the most basic statements. This type system can calculate the maximum number of transactions that an STM program requires in the worst case. These works help us make basic rules and prove the attributes of the type system conveniently. The type system in our work [5] was developed from the type system in our previous works [7, 23], in which we have improved the type system to calculate the maximum resource that the program needs to use rather than just count the maximum number of transactions as in the previous works.

In [5], each transaction is provided a parameter. These parameters represent the resource that the transaction requires. Our type system calculates the maximum log memory that the program needs to use based on those parameters.

The results of those works are generalized so that users can develop them to suit their purposes, such as calculating the cost of memory resources, CPU, network bandwidth, time, or the number of gas required for smart contracts in Ethereum. However, the calculation of the consumed resource of each transaction (the parameter of the transaction) is manual, so this is still semi-automatic. Then, in [24], we have improved the language and type system to calculate the log memory from the program's shared variables in a completely automated way. In [6], our type system is developed from our works [5, 8], in which we expanded the language, so that it was closer to the actual language, and we have also improved the type system for more convenient calculations.

The main feature of the type system in this work is the ability to composition. This means that it is possible to type any term of the program, then we combine them to get the type of the program.

3. Transactional Language

In this section, we present our STM language. This language is developed from [6], where we improve some syntax and semantics rules so that it is closer to the actual language as introduced in Section 1. Specifically, we add the S-ASSIGN rule, improve the S-WHILE, S-NEW, S-TRANS rules, and add some auxiliary functions to make the presentation of the rules more detailed. We explain these improvements in each rule in more detail.

3.1. Syntax

The syntax of the language is presented in Table 1.. At line 1., P represents threads or processes, which can be an empty thread, denoted by $\emptyset$, or multiple parallel threads $P||P$, or a thread p executing expression e. At line 2., T represents primitive types, which can be the integer or boolean type. D at line 3. is a list of variable declarations, which can be the shared variables or local variables. O at line 4. represents the operators, which can be arithmetic operations such as $+$, $-$, $\times$, $\div$ (denoted by $\bullet$); relational operations such as $=$, $>$, $\geq$, $<$, $\leq$ (denoted by $\blacksquare$); logical operations such as AND, OR (denoted by $\blacklozenge$), NOT (denoted $\blacktriangle$).

At lines 5., 6., v_i, v_b are integer and logic values respectively. At line 7., v is values which can be integer values or logical values. At line 8., e_i is an integer expression or an integer value; At line 9., e_b is a boolean expression or a logic value.

From line 10. to line 14., an expression e can be a statement declaring and initializing new variables, assignment, sequencing (denoted by $e; e$), branch statements, loop statements, the statement `spawn{e}` to create a new thread that executes e, the statement `onacid` is for opening a new transaction, the `commit` statement is for closing a transaction.

We assume that, in each transaction, variables are declared at the beginning of each transaction, and when the program spawns a thread in that transaction, they copy all these variables into the new thread. These variables usually copied in batch and store in a memory area called transaction log, or simply log. The size of the logs will be the total memory of the variables that are cloned or initialized within it.

3.2. Dynamic Semantics

In this section, we present the semantics of the language, define the environments, and auxiliary functions that support the representation of rules. A thread consists of many transactions, and each transaction contains many variables. We call the environment of a thread the local environment [6]. The environment of the program is called the global environment [6], and which is a collection of local environments. Transaction size is the total memory for the variables inside it. We have the definition of the local environment as follows.

Definition 1 (Local environment). A Local environment *E is a finite sequence identifier of the transactions and their sizes, E = {l_1:n_1; l_2:n_2; ...; l_k:n_k}. An environment without any elements is called an empty environment, denoted by* ϵ [6].

Definition 2 (Global environment). A global environment Γ *is a set of identifiers of threads and their local environments, Γ = {p_1:E_1; p_2:E_2; ...; p_k:E_k} [6].*

The total memory consumed by the program at a time is the total memory of the program's open transactions at that time.

Table 1. The syntax of the transactional language

1.	P	::=	$\phi \mid P \parallel P \mid p(e)$	processes/threads
2.	T	::=	**int** $\mid$ **bool**	type
3.	D	::=	**shared** $T\ \vec{x} := \vec{v} \mid T\ \vec{x} := \vec{v}$	initialize variables
4.	O	::=	$\bullet \mid \blacksquare \mid \blacklozenge \mid \blacktriangle$	operators
5.	v_i	::=	n	integer values
6.	v_b	::=	**true** $\mid$ **false**	boolean values
7.	v	::=	$v_i \mid v_b$	values
8.	e_i	::=	$e_i \bullet e_i \mid v_i$	integer expressions
9.	e_b	::=	$e_i \blacksquare e_i \mid e_b \blacklozenge e_b \mid \blacktriangle e_b \mid v_b$	boolean operators
10.	e	::=	$D \mid e;e \mid x := e_i \mid x := e_b$	expressions
11.			$\mid$ **if**(e_b) **then**$\{\ e\ \}$ **else**$\{\ e\ \}$	conditional
12.			$\mid$ **while**$(e_b)\{\ e\ \}$	loop
13.			$\mid$ **spawn**$\{\ e\ \}$	spawn thread
14.			$\mid$ **onacid** $\mid$ **commit**	open/close transactions

Definition 3 (Total memory). The total memory *consumed by the program at a time is* Γ*, and* $[\![\Gamma]\!] = \sum_{i=1}^{n} [\![E_i]\!]$*, where n is the number of threads of the program.*

A pair of an environment Γ and a collection of threads P are called a *state* Γ, P of the program. A special state called *error* describes the fault state - the state at which none of the transaction rules can be applied. The dynamic semantics rules are specified by transition rules of the form $\Gamma, P \Longrightarrow \Gamma', P'$ *or* $\Gamma, P \Longrightarrow error$ as in Table 2.

In Table 2, we assume some equivalence rules: $P \| P' \equiv P' \| P$, $P \| (P' \| P'') \equiv (P \| P') \| P''$ and $P \| 0 \equiv P$, and some auxiliary functions as described below.

- S-TRANS: This rule to start a new transaction (execute statement `onacid`). In this rule, the function *onacid(l, p, Γ)* creates a transaction l with memory size 0 at the end of the local environment of p. If *onacid(l,*p_i*, Γ)= Γ'*, where *Γ = {*p_1*:*E_1*, ...,* p_i*:*E_i*, ...,* p_k*:*E_k*}* and with statement `fresh l` then *Γ'={*p_1*:*E_1*, ...,* p_i*:*E'_i*, ...,* p_k*:*E_k*}* where $E_i'=E_i;l:0$.

- S-COMM: This rule is used to commits a transaction. In this rule, $\coprod_1^k p_i(e_i)$ stands for $p_1(e_1)\|\ldots\|p_k(e_k)$. If the current transaction identifier of p is l, then all threads with transaction identifier l must joint commit when transaction l commit.

In this rule, function *intrans(Γ, l:n)* returns a set of all threads inside this transaction l, denoted by **p**. In the environment *Γ* contains transaction l and this transaction is the last element of this environment. This means that if *intrans(Γ, l:n)* = **p** = $p_1, \ldots, p_k$ then:

– For all $i \in \{1 \ldots k\}, p_i$ has the form $E_i';\ l{:}n$,

– For all $p'{:}E' \in \Gamma$ such that $p' \notin \{p_1, \ldots, p_k\}$ then we have E' does not contain transaction l.

Function *commit*(**p**, Γ*)* removes the last transaction in the local environments of all threads in **p**. Suppose *intrans*(Γ, *l:n*) = **p** and *commit*(**p**, Γ)= Γ′, for all *p'*:*E'* ∈ Γ′, if *p'* ∈ **p**, then *p'*:*(E'; l:n)* ∈ Γ. For other cases *p'*:*E'* ∈ Γ.

- S-NEW: This rule is used to initialize a new shared variable, where function *new(x,l,p, Γ)* initialize a shared variable `x` at the end of transaction l (the last transaction at that time). If $new(x,l,pi,\Gamma)=\Gamma'$, *Γ ={*$p_1$*:*$E_1$*, ...,* p_i*:*E_i*, ...,* p_k*:*E_k*}*, and $E_i=\{l_1{:}n_1;\ \ldots;\ l_j{:}n_j\}$ then

Table 2. The semantics of transactional language

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

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

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

$$\frac{x:T \quad v:T \quad \mathit{new}(x,l,p,\Gamma)=\Gamma' \quad [\![E]\!]>0}{\Gamma, P \parallel p(\mathbf{shared}\ T\ x := v; e) \Longrightarrow \Gamma', P \parallel p(e)}\ \text{S-NEW}$$

$$\frac{i = e_b \downarrow \mathbf{true}\ ?\ 1:2}{\Gamma, P \parallel p(\mathbf{if}(e_b)\ \mathbf{then}\ \{\ e_1\ \}\ \mathbf{else}\ \{\ e_2\ \}; e) \Longrightarrow \Gamma, P \parallel p(e_i; e)}\ \text{S-COND}$$

$$\frac{e' = e_b \downarrow \mathbf{true}\ ?\ e_1;\ \mathbf{while}(e_b)\{\ e_1\ \};\ e_2 : e_2}{\Gamma, P \parallel p(\mathbf{while}(e_b)\{\ e_1\ \};\ e_2) \Longrightarrow \Gamma, P \parallel p(e')}\ \text{S-WHILE}$$

$$\frac{x_{11}:T,\ e_1:T,\ \mathit{write}(x_{11},e_1,l,p,\Gamma) = \ \mathit{isclone}(x_{11})?\ x_{11}=e_0; x_{12}=e_1, \Gamma' : x_{11}=e_1, \Gamma}{\Gamma, P \parallel p(x_{11} := e_1; e_2) \Longrightarrow \Gamma', P \parallel p(e_2)}\ \text{S-ASSIGN}$$

$$\frac{}{\Gamma, P \parallel p(\alpha;\ e) \Longrightarrow \Gamma, P \parallel p(\alpha';\ e)}\ \text{S-SKIP} \qquad \frac{\Gamma = \Gamma' \cup \{p:E\} \quad [\![E]\!]=0}{\Gamma, P \parallel p(\mathbf{commit};\ e) \Longrightarrow \mathit{error}}\ \text{S-ERROR}$$

$\Gamma' = \{p_1{:}E_1, \ldots, p_i{:}E'_i, \ldots, p_k{:}E_k\}$, and $E'_i=\{l_1{:}n_1;\ \ldots;\ l_j{:}n'_j\}$, where $n'_j=n_j+m$, m is the memory size of the initialized variable.

For variables that are declared outside the transactions and local variables, they do not affect the memory resources caused by the STM mechanism, so in this work, we do not care about these variables.

- S-SPAWN: This rule is applied to create a new thread. The statement `spawn{ e1 }` creates thread *p'* for executing *e1* in parallel with thread *p* (it's parent thread) and the environment Γ changes to Γ'. The function $spawn(p, p', \Gamma)$ adds to Γ a new thread *p'*. Its local environment is copied from the local environment of the thread. Suppose $\Gamma = \{p{:}E\} \cup \Gamma''$ and $spawn(p, p', \Gamma) = \Gamma'$ then $\Gamma' = \Gamma \cup \{p'{:}E'\}$ where *E'*=*E*.

- S-ASSIGN: This rule is used to assign a value to a variable as usual standard semantics of programming languages. In this rule, variable x_{11} and value of expression e_1 must be of the same type *T*. e_0 is the value of x_{11} before it is assigned the value e_1. The function *isclone(x_{11})* returns *true* if the variable x_{11} is cloned from another thread (the parent thread of the current thread), and returns *false* if the variable x_{11} is initialized in the current thread.

If the variable x_{11} is initialized in the current thread, then when the program performs the assignment, it is assigned a new value (e_1) instead of the old value (e_0), and the environment Γ is not changed.

If the variable x_{11} is the cloned variable from the current thread's parent thread, then when performing the assignment, the function *write* adds a new variable adjacent to it to store the new value (e_1). Thus, its old value (e_0) is not overwritten. This is to serve the joint commit between the threads. In this case, the environment is changed from Γ to Γ' as follows:

$\Gamma = \Gamma'' \cup \{p_i{:}E_i\}, \Gamma' = \Gamma'' \cup \{p_i{:}E_i'\},$

$E_i = E_i'' \cup \{l_j{:}n_j\}, E_i' = E_i'' \cup \{l_i{:}n_j'\},$

$n_j' = n_j + size(x)$

The function *size(x)* returns the memory size of the variable `x`.

- S-COND: This rule to execute the conditional statement. The expression $i = e_b \downarrow true\ ?\ 1 : 2$ means that if e_b is *true* then *i* get value 1 else *i* get value 2.

- S-WHILE: This rule is used to implement the loop in the program. The expression e'$= e_b \downarrow true?\ e; while(e_b)\{\ e\ \};\ e_2 :\ e_2$ means that if e_b is *true* then *e'=e; while(e_b){ e }; e_2 else e'=e_2.*

In our previous work, expression e in the body of the loop has to close w.r.t transaction and does not contain statement `spawn`. In this work, the body of the loop can be any expression. However, in order to determine the resources consumed by these loop expressions, we are assuming that the maximum number of loops is known. This is also a drawback in this work, and we plan to address it in future work.

- S-SKIP: This rule is used for other operations, such as •, ■, ♦, ▲ which do not affect the transaction and multi-threaded semantics, so we simplify here by a skip operation.

- S-ERROR: This rule is used to handle error cases, e.g., commit in an empty environment.

4. Type System

Our type system aims to estimate the upper bound of the maximum memory required by shared variables in multi-threaded transaction programs. Each program segment (called a term) is typed through a special string, thereby abstracting the transaction behavior of that program.

The type rules presented here are inherited from our previous work [5, 6], however they are improved to match the current language and are described in more detail.

4.1. Type

We use a set of symbols with non-negative numbers to represent the type of a term. The type of a term is a finite sequence of numbers with symbols, which are marked by a pair of a symbol and a non-negative natural number in the set $\mathbb{N}^+$. We use the set $\{\star, +, -, \neg, \#\}$ to denote the initialization of a variable, open, commit, joint commit a transaction, and memory maximum allocation for the logs. The set of numbers with symbols is denoted by ${}^T\mathbb{N}$, i.e. ${}^T\mathbb{N} = \{{}^\star n, {}^+ n, {}^- n, {}^\neg n, {}^\# n \mid n \in {}^\star\mathbb{N}\}$. The numbers assigned to these symbols have the following meaning:

- ${}^{\star}n$: Initializes a variable with size n, meaning that it needs to be allocated n units of memory to that variable.
- ${}^{+}n$: The opening of a transaction, the memory allocated for that transaction is n. The case $n=0$ means that a transaction has been opened but no variable is initialized in that transaction.
- ${}^{-}n$: There are n commit statements in succession to finish the previous transactions.
- ${}^{\neg}n$: There are n threads need to synchronize at a time.
- ${}^{\#}n$: The maximum memory required for a term is n units of memory.

Definition 4 (Type of a term). *The type T of a term in our system is defined as follows [6]:*

$$T = S \mid TT \parallel \mathrm{T} \otimes \mathrm{T} \mid T^{\rho} \mid T \oslash T \mid T \|^{k} T$$

The type of a term can be a sequence of tagged numbers S as described in Section 4.1, or synthesized from other types of terms. In this definition, TT means that the type of term is derived from the type of two sequential terms. T^{ρ} means that a term has type T that will be

Table 3. Typing rules

$$\frac{}{0 \vdash \textbf{onacid} : {}^{+}0}\ \text{T-ONACID} \qquad \frac{n \in \mathbb{N}^{*}}{n \vdash \textbf{commit} : {}^{-}1}\ \text{T-COMMIT} \qquad \frac{n \vdash e : T}{n \vdash \textbf{spawn}\{\ e\ \} : T^{\rho}}\ \text{T-SPAWN}$$

$$\frac{x : T \quad v : T \quad n = size(x)}{-n \vdash \textbf{shared}\ T\ x := v :\ {}^{\star}n}\ \text{T-NEW} \qquad \frac{n = isclone(x)\ ?\ 0\ :\ size(x)}{-n \vdash x := e_1 :\ {}^{\star}n}\ \text{T-ASSIGN} \qquad \frac{n_k \vdash e_k : T_k \ \ k = 1,2}{n_1 + n_2 \vdash e_1; e_2 : T_1 T_2}\ \text{T-SEQ}$$

$$\frac{e_b : \textbf{bool} \quad n_k \vdash e_k : T_k \quad k = 1,2}{n \vdash \textbf{if}(e_b)\ \textbf{then}\{\ e_1\ \}\ \textbf{else}\{\ e_2\ \} : T_1 \oslash T_2}\ \text{T-COND} \qquad \frac{e_b : \textbf{bool} \quad n \vdash e : T \quad m = maxloop()}{n' \vdash \textbf{while}(e_b)\{\ e\ \} : T^{m}}\ \text{T-WHILE}$$

$$\frac{n \vdash e : T}{n \vdash p(e) : T}\ \text{T-THREAD} \qquad \frac{n \vdash e_1 : T_1^{\rho} \quad n \vdash e_2 : T_2}{n \vdash e_1; e_2 : T_1^{\rho} T_2}\ \text{T-MERGE}$$

$$\frac{\Gamma_k, P_k \quad n_k \vdash P_k : T_k \quad \Gamma{=}\Gamma_1,\Gamma_2 \quad common(\Gamma_1,\Gamma_2){=}n \quad k{=}1,2}{n_1 + n_2 \vdash P_1 \parallel P_2 : T_1 \parallel^{n} T_2}\ \text{T-PAR}$$

executed in a thread parallel to its parent thread. The $\mathrm{T} \otimes \mathrm{T}$, $T \oslash T$, and $T\|^{k}T$ operations are merged, choice, and parallel operations, respectively, are to create new terms from existing terms, and they will be described in detail in the next section 4.2.

4.2. Typing Rules

The typing rules are described in Table 3., where the type of a term is of the form$n \vdash e$: T, and we read that e has type T. n is the environment of the expression, and it represents the amount of memory consumed or released when the program executes e.

- **T-ONACID, T-COMMIT:** These two rules are used to type expressions **onacid** and **commit** corresponding. The statement onacid to open a transaction, its type is ${}^{+}0$. The statement commit to close the last opened transaction, its type is ${}^{-}1$.

- **T-SPAWN:** This rule is used to type the element **spawn**{ e }. If the type of e is T then **spawn**{ e } has the type T^{ρ}. Here symbol ${}^{\rho}$ is used to indicate that the current thread is running parallel with the parent thread.

- **T-NEW:** This rule is used to type the statements that initialize a new reference object. The function *size(x)* returns the size of the variable *x* (memory allocated for variable *x*). In this work, we assume that an integer variable needs 2 memory units, a boolean variable needs 1 memory unit. If *e* is ***shared int*** *x=0;* then type of *e* is $^{\star}2$; if *e* is ***shared bool*** *x=0;* then type of *e* is $^{\star}1$.

- **T-SEQ:** This rule represents sequential components of e_1 and e_2. When two expressions are sequential, its type is also sequential. Suppose e_1 has type T_1 and e_2 has type T_2, where T_1 has a form without $^{\rho}$, $e_1; e_2$ has type T_1T_2.

- **T-MERGE:** This rule to type expressions with two parallel elements. The symbol T_1^{ρ} indicates that the thread of type T_1 is running in parallel with its parent thread. To simplify the expression of type $T_1^{\rho}T_2$ we use the $\otimes$ operation shown in the following.

When e_1 ends with some expressions *spawn*, it means that e_1 has the type T_1^{ρ}. When its type is associated with T_2, we need to use the aggregate operation to type the string $e_1; e_2$ then it will be of the form $T_1 \otimes T_2$.

- **T-COND:** This rule is used to type for conditional expressions. We assume that e_1 and e_2 have types of T_1 and T_2, respectively. If the expression *e* is **if**(e_b) t**hen**{ e_1 } **else**{ e_2 }$ then the type of *e* is $T_1 \oslash T_2$.

- **T-WHILE:** This rule is used to type the loop expression. The symbol T^m to describe a sequence of *m* components *T* consecutively, where *m* is the maximum number of loops, and *T* is the type of the loop body expression. By this rule, our type system can type any loop expressions with a known maximum number of loops. For general loop expressions, we cannot statically determine the number of loops, because it depends on the values of variables during program execution. We plan to investigate loop-bound analysis in developing a type inference algorithm for our type system in a future work.

- **T-THREAD:** If an expression *e* has type *T*, then the thread executing it also has type *T*. *e* here is the expression that will be executed by the thread, so its type must be a canonical sequence as in Definition 5.

- **T-PAR:** This rule is used to type programs at the time the program is running. At this time, the program has many parallel threads running. If we just need to type static for the program, we do not need this rule. However, this rule helps us prove the correctness and sharpness of the type system.

For cases where expressions execute outside transactions, or it does not consume memory by the STM mechanism, we use the rule T-SKIP. This means that we can skip the typing of these expressions. The function *summem(e)* returns the total memory that expression e uses by the STM mechanism.

For convenience, we can add or remove elements of $^{tag(s)}0$ form from the string because it does not affect the semantics of the string. The set $^{T}\mathbb{N}$ can be divided into equivalence classes, in which all elements in the equivalent class describe the same transaction behavior, and each class uses the most concise string to describe the class. We call it the *canonical* string.

Definition 5 (Canonical sequence*).* *A sequence S is canonical if tag(S) does not include elements* $'\star\star'$, $'++'$, $'--'$, $'\#\#'$, $'+-'$, $'+\star-'$, $'+\#-'$, $'+\#\star-'$, *and* $[\![S(i)]\!] > 0$ *for all i.*

We can always reduce an S string without changing the way to understand it. Note that, during string reduction, the pattern $'+-'$ may not appear on the left, but we can add $^{\#}0$ to apply the function.

Note that, during reducing string we cannot reduce the string in ^{+}n, ^{+}m format, because each element ^{+}n, ^{+}m represents an open transaction, it needs to be committed with an element $^{-}1$.

For convenience of presenting operations in the following sections, we introduce several notations and their meanings:

- $s^{\#}$ represents an empty sequence or an element ${}^{\#}n$, i.e. $s^{\#}_k \in \{\epsilon\} \cup \{{}^{\#}n | n \in \mathbb{N}\}$.
- $s^{\mathbf{n}}$ (**n** stands for **n**egation) represents ${}^{-}1$ or ${}^{\neg}n$, i.e. $s^{\mathbf{n}}_k \in \{{}^{-}1\} \cup \{{}^{\neg}n | n \in \mathbb{N}^{+}\}$.
- $s^{\mathbf{c}}$ (**c** stands for **c**ommit) represents ${}^{-}1$, ${}^{\neg}n$, or $T_1 \|^0 \ldots \|^0 T_k$ i.e. $s^{\mathbf{c}}_k \in \{{}^{-}1\} \cup \{{}^{\neg}n | n \in \mathbb{N}^{+}\} \cup \{T_1 \|^0 \ldots \|^0 T_k | k \geq 2\}$.
- $[\![s^{\#}]\!]$ represents the natural number of *s* corresponding, for example, $[\![5^{\#}]\!]$ is equal to 5.

The following are some rules to reduce a sequence of tagged numbers to a canonical sequence.

i) In a transaction, memory resources allocated to consecutive variables are equal to the total memory allocated for each variable.

$$^{\star}n\,^{\star}m \Rightarrow {}^{\star}(n+m)$$

For example, for the expression *e* is shared int x=0; shared bool y=0; where an integer variable needs 4 units of memory, the logical variable needs 1 unit of memory then the type of *e* is:

$$^{\star}4\,^{\star}1 \Rightarrow {}^{\star}(4+1) \Rightarrow {}^{\star}5.$$

ii) Inside an open transaction, if we initialize one more variable with memory to use is *m* then the memory needed for that transaction increases by *m*.

$$^{+}n\,^{\star}m \Rightarrow {}^{+}(n+m)$$

For example, for the expression e is onacid shared int x=0; shared int y=0; then the type of *e* is ${}^{+}0\,{}^{\star}2\,{}^{\star}2 \Rightarrow {}^{+}2\,{}^{\star}2 \Rightarrow {}^{+}4$.

iii) For nested transactions, the memory that they need to use is the total memory of the component transactions.

$$\begin{array}{llr} ^{+}n\, s^{\#}\, ^{-}1 & \Rightarrow {}^{\#}([\![s^{\#}]\!]+n) & (1) \\ ^{+}n\, s^{\#}_1 s^{\#}_2\, ^{-}1 & \Rightarrow {}^{\#}([\![s^{\#}_1]\!]+n)\, {}^{\#}([\![s^{\#}_2]\!]+n) & (2) \end{array}$$

The formula (1) is used to type program segments that behave as described in segment AB, formula (2) is used to type program segments that behave as described in segment BC in the Figure 2.

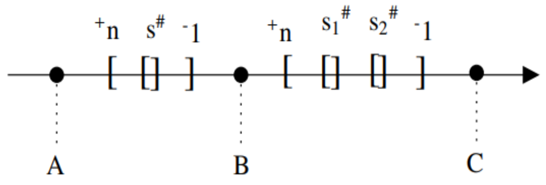


Figure 2. Nested transactions.

iv) Two consecutive terms have types ${}^{\#}n, {}^{\#}m$, then their type is the type of term that has a greater value, so we have the formula to reduce the string as follows.

$$^{\#}m\,^{\#}n \Rightarrow {}^{\#}max(m,n)$$

In this case, the behavior of the program segment is similar to that described in the Figure 3.

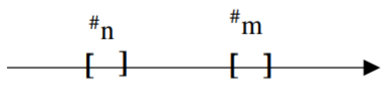


Figure 3. Two consecutive terms.

v) Symbol $^{\rho}$ is used to mark that the expression is executed in parallel to its parent thread.

$$(T^{\rho})^{\rho} = T^{\rho}$$

In case the expression does not contain joint commits, we can ignore them:

$$({}^{\#}n)^{\rho} = {}^{\#}n$$

For expression of the form $T_1^{\rho} T_2$, then T_2 is the remainder of the parent thread after spawning the child thread executing T_1. Since T_2 has joint commits with T_1, we can join T_1 with T_2 to make it ready for joint commit.

$$T_1^{\rho} T_2 \Rightarrow T_1 \otimes T_2$$

The $\otimes$ operation is defined rule (6). In this case, the behavior of the program segment can be described in the Figure 4.

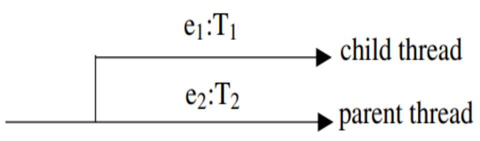


Figure 4. Two parallel threads.

vi) The $\otimes$ operation is used to combine the type of threads in parallel.

$$\begin{array}{rcl}
{}^{\sharp}m \otimes {}^{\sharp}n & \Rightarrow & {}^{\sharp}(m+n) \\
{}^{\sharp}m \otimes \epsilon & \Rightarrow & {}^{\sharp}m \\
\epsilon \otimes {}^{\sharp}m & \Rightarrow & {}^{\sharp}m \\
s_1^{\sharp} s_1^{\mathbf{n}} T_1 \otimes s_2^{\sharp} s_2^{\mathbf{n}} T_2 & \Rightarrow & (s_1^{\sharp} \otimes s_2^{\sharp})\, {}^{\neg}(\llbracket s_1^{\mathbf{n}} \rrbracket + \llbracket s_2^{\mathbf{n}} \rrbracket) \\
 & & (T_1 \otimes T_2) \\
(T_1 \otimes T_2) T_3 & \Rightarrow & T_1 \otimes (T_2 T_3) \\
T_1 \otimes {}^{\sharp}s\, T_2^{\rho} & \Rightarrow & T_1 \otimes {}^{\sharp}s\, T_2 \\
T_1^{\rho} \otimes T_2^{\rho} & \Rightarrow & (T_1 \otimes T_2)^{\rho}
\end{array}$$

During this operation, we look for joint commits from left to right to merge them. In case T_1 contains joint commits but T_2 does not then we do not merge them and wait for the next expression.

vii) The $\oslash$ operation is used to type conditional statements. In this case, we will choose the element which has a larger value.

$$\begin{array}{rcl}
{}^{\sharp}m \oslash {}^{\sharp}n & \Rightarrow & {}^{\sharp}\max(m,n) \\
{}^{\sharp}m \oslash \epsilon & \Rightarrow & {}^{\sharp}m \\
\epsilon \oslash {}^{\sharp}m & \Rightarrow & {}^{\sharp}m \\
s_1^{\sharp}\, {}^{\mp}n_1 T_1 \oslash s_2^{\sharp}\, {}^{\mp}n_2 T_2 & \Rightarrow & (s_1^{\sharp} \oslash s_2^{\sharp})\, {}^{\mp}(max(n_1, n_2) \\
 & & (T_1 \oslash T_2) \\
T_1^{\rho} \oslash T_2^{\rho} & \Rightarrow & (T_1 \oslash T_2)^{\rho}
\end{array}$$

In this rule, for brevity, we use the symbol ${}^{\mp}n$ instead of the symbol ${}^{+}n$ or ${}^{-}n$ or ${}^{\neg}n$.

viii) The following rules are used for threads to joint commit. The value n in element ${}^{\neg}n$ represents the number of threads inside the latest opened transaction. This case is described in the Figure 5., and we can combine them with the following rules.

$$ {}^{+}n_1\, (s^{\sharp}\, {}^{\neg}n_2\, S)^{\rho} \Rightarrow (\, {}^{\sharp}(\llbracket s^{\sharp} \rrbracket + n_1 * n_2) S)^{\rho} $$

The following rule is similar to the above, but we are interested in nested transactions, the same as described in the Figure 6.

$$\begin{array}{l}
{}^{+}n_1\, s_1^{\sharp}\, (s_2^{\sharp}\, {}^{\neg}n_2\, S)^{\rho} \Rightarrow \\
{}^{\sharp}(\llbracket s_1^{\sharp} \rrbracket + n_1)(\, {}^{\sharp}(\llbracket s_2^{\sharp} \rrbracket + n_1 * n_2) S)^{\rho}
\end{array}$$

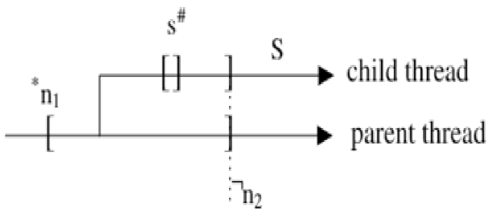


Figure 5. Joint commit parallel threads (case 1).

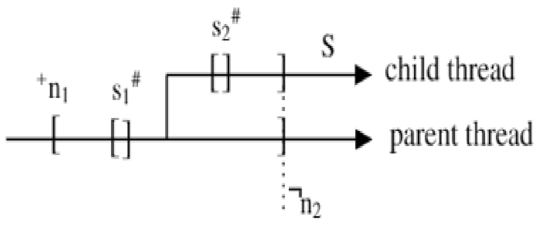


Figure 6. Joint commit parallel threads (case 2).

The above rules are used to type the program when it is not running (static). If your purpose is only to determine the type of program, the above rules are sufficient. However, to prove the correctness and sharpness of the type system, we need the following two rules. The two rules (9), (10) are to type the program when the program is running (dynamic), and already have threads running in parallel.

ix) The following rules are used for threads to joint commit between threads in running time.

$$\begin{array}{rcl}
\epsilon \parallel^{0} T & \Rightarrow & T \\
T \parallel^{0} \epsilon & \Rightarrow & T \\
{}^{\sharp}m \parallel^{0} {}^{\sharp}n & \Rightarrow & {}^{\sharp}(m+n) \\
T_1 s_1^{\mathbf{c}} s_1^{\sharp} \parallel^{k} T_2 s_2^{\mathbf{c}} s_2^{\sharp} & \Rightarrow & (T_1 \parallel^{k-1} T_2)(s_1^{\mathbf{c}} \parallel^{0} s_2^{\mathbf{c}}) \\
 & & (s_1^{\sharp} \parallel^{0} s_2^{\sharp})
\end{array}$$

Listing 2. Example of applying rules (9), (10)

```
onacid;
 shared bool x=0;
 spawn{
  onacid;
   shared int y=0;
  commit;
 commit;
 }
 onacid;
  shared int z=0;
 commit;
commit;
```

where $k>0$ and $T_1 s_1^c s_1^\#$, $T_2 s_2^c s_2^\#$ is canonical forms.

x) Similar to rule (8)., the rules below apply at the time that the program is running.

+) $^{+}n_1\, s^{\sharp}(\,^{\sharp}n_2 \parallel^0 T) \Rightarrow$
$(\,^{\sharp}(\llbracket s^{\sharp}\rrbracket + n_2) \parallel^0 {}^{+}n_1\, T)$
when $^{\sharp}n_2 \parallel^0 T$ is canonical forms.

+) $^{+}n\, s^{\sharp}(S \parallel^0 T) \Rightarrow s^{\sharp}(\,^{+}n\, S \parallel^0 T)$
when $S \parallel^0 T$ are canonical form and S no form $^{\sharp}n$.

To explain more clearly these two rules, we consider the example in Listing 2.

We apply the rules T-ONACID, T-COMMIT, T-INIT, T-SPAWN to type the program. We then apply rule (9) to reduce them, and get the type of the program is

$$\begin{aligned}&{}^{+}0\ {}^{*}1\ (\,{}^{+}0\ {}^{*}2\ {}^{-}1\ {}^{-}1\,)^{\rho}\ {}^{+}0\ {}^{*}2\ {}^{-}1\ {}^{-}1\\ \Rightarrow\ &{}^{+}1\ (\,{}^{\sharp}2\ {}^{-}1\,)^{\rho}\ {}^{\sharp}2\ {}^{-}1\end{aligned}$$

After the spawn statement runs, the first thread has an open transaction, and their memory is duplicated. The type of the rest of the two threads are thread 1: $^{\#}2^{-}1$; thread 2: $^{\#}2^{-}1$.

Apply rule T-PAR, and apply rule (9) to reduce them, we have:

$$\begin{aligned}(\,{}^{\sharp}2\ {}^{-}1\,) \parallel^1 (\,{}^{\sharp}2\ {}^{-}1\,) &\Rightarrow (\,{}^{\sharp}2 \parallel^0 {}^{\sharp}2\,)(\,{}^{-}1 \parallel^0 {}^{-}1\,)\\ &\Rightarrow {}^{\sharp}4\,(\,{}^{-}1 \parallel^0 {}^{-}1\,)\end{aligned}$$

This is the type of program at the time after executing the statement `spawn`. Besides, it has an open transaction and its memory is being duplicated. So the type of this transaction is: $^{+}1^{+}1$.

We add this sequence to the above type sequence, and apply rule 10. to reduce them, we have:

$$\begin{aligned}{}^{+}1\ {}^{+}1\ {}^{\sharp}4\,(\,{}^{-}1 \parallel^0 {}^{-}1\,) &\Rightarrow {}^{+}1\ {}^{\sharp}4\,(\,{}^{+}1\ {}^{-}1 \parallel^0 {}^{-}1\,)\\ &\Rightarrow {}^{+}1\ {}^{\sharp}4\,(\,{}^{\sharp}1 \parallel^0 {}^{-}1\,)\\ &\Rightarrow {}^{\sharp}4\,(\,{}^{\sharp}1 \parallel^0 {}^{+}1\ {}^{-}1\,)\\ &\Rightarrow {}^{\sharp}5 \parallel^0 {}^{\sharp}1\\ &\Rightarrow {}^{\sharp}6\end{aligned}$$

So the program type is $^{\#}6$, or the maximum memory the program needs is 6 units. Since the type in this work reflects the behavior of a term of a program, so the type of a well-type program is a string containing only one element $^{\#}n$, where n is the maximum memory that the program requires during executing it.

Definition 6 (Well-typed). *A program is* well-typed *if it has type T and T* $\Rightarrow^* s^\#$ [6].

Definition 7 (Resource consumption). *If Γ, P is a running state of the program and $P{:}T$, then the maximum resource consumption during executing P is:*

$$\llbracket \Gamma, P \rrbracket = \begin{cases} \llbracket s^{\sharp} \rrbracket & \text{if } S\,T \Rightarrow^* s^{\sharp} \\ error & otherwise \end{cases}$$

where $S = {}^{+}n_1\ {}^{+}n_2 \ldots {}^{+}n_k$ and $\llbracket S \rrbracket = \llbracket \Gamma \rrbracket$ [6].

4.3. Characteristic of the Type System

In this section, we present the characteristics of type systems and apply them to prove the correctness and sharpness of type systems.

A type of the expression e has the characteristic that its environment is only sufficient for open transactions to commit in e as described by its type. We have the following theorem.

Theorem 1 (Type judgment property). *Assume e:T, and its environment is Γ, if $ST \Rightarrow^* s^\#$ then $[\![s^\#]\!] \geq [\![S]\!]$, where $S = {}^+n_1{}^+n_2 \ldots {}^+n_k$, and $[\![S]\!] = [\![\Gamma]\!]$ [6].*

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

During program execution by the semantic rules in the Table 2, if the program changes from state Γ, P to Γ', P' and $[\![\Gamma, P]\!] = n$, then $[\![\Gamma', P']\!] = n'$ and $n \geq n'$ for any rules. We formally express this characteristic in the following theorem.

Lemma 1 (subject reduction). *If $\Gamma, P \Longrightarrow \Gamma', P'$ by R rule and $[\![\Gamma, P]\!] = n$ then $[\![\Gamma', P']\!] = n'$ and $n \geq n'$ for $\forall$ R [6].*

Proof (Sketch). The proof is done by checking one by one of all the semantics rules in Table 2.

Lemma 2 (Preservation). *Given a well-typed P_0, its type is T and $T \Rightarrow^* s^\#$. For any state Γ, P of the program, we have $[\![\Gamma', P']\!] \leq [\![s^\#]\!]$ [6].*

Proof. By inductions on transitions of the semantics.

- Initial state: $[\![\emptyset, P]\!] = [\![s^\#]\!] \leq [\![s^\#]\!]$.
- If $\Gamma, P \Longrightarrow \Gamma', P'$, *assume that* $[\![\Gamma, P]\!] \leq [\![s^\#]\!]$, *by Lemma 1, we have* $[\![\Gamma', P']\!] \leq [\![\Gamma, P]\!] \leq [\![s^\#]\!]$.

The correctness of the type system is understood that a well-typed program does not use more memory than the amount expressed in its type.

Theorem 2 (Correctness). *Given a well-typed program P_0, its type is T, and $T \Rightarrow^* s^\#$ then the resource consumption of the program during running cannot exceed $]\!] \leq [\![s^\#]\!]$ [6].*

Proof. Let Γ, P be a state of the program, by the Lemma 2, we have $[\![\Gamma, P]\!] \leq [\![s^\#]\!]$. By Definition $[\![\Gamma, P]\!]$ and Theorem 1 we infer $[\![\Gamma]\!] \leq [\![s^\#]\!]$.

This theorem asserts that, if a program is well-typed then maximum memory usage of the program will not exceed the value expressed in its type.

5. Typing the Example Program

In this section, we apply the rules in Table 3. to build a type inference tree for the example program in List 1.

In this work, our type system can type any term, and then integrate them into the program's type. However, in order to facilitate the analysis of sharpness in Section 6.1, we divide it into two steps as follows: First, we type the program segment e_{29}^{10} (the program segment contains the conditional statement). Then we combine it with the rest of the program to get the type of the program.

By applying the rules in Table 3, we can build a type inference tree for the program segment e_{29}^{10} as shown in the Figure 7.

We inherit this result, and do the same for the rest of the program, we get the type inference tree of the program as shown in the Figure 8.

Note that the variables `a` and `b` at line 1 are local variables, so we omit it.

Through this type inference tree, we can conclude that, in the worst case scenario, the program can use up to 12 memory units for shared variables.

6. Discussion

In this section, to explain more clearly about our work, we discuss sharpness, and evaluations of our proposed solution.

6.1. Sharpness

The sharpness of type system is understood that if given a well-typed program then there is always a path from the initial state Γ_0, P_0 to the state Γ, P such that the memory consumption of the program at state Γ, P to be equal to the value shown in their type expression.

In this work, our typing rules ensure that the memory bound is always greater than or equal to the total memory required by the program (i.e., ensuring the correctness of the type system). However, there are some instances where the bound found by these rules is greater than the total memory that the program needs to use (i.e. bound is not sharp).

During proving the correctness and sharpness of the type system, we realized that, for expressions that contain conditional statements, the memory bound found may not guarantee sharpness. To better understand this problem, we consider the program segment e_{29}^{10} in the example in List 1. Applying the rules in Table 3., we build the type inference tree for the program segment e_{29}^{10} in the Figure 7.

Through the type inference tree, we realized that the type of e_{29}^{10} is $^{\#}5$. This means that it is a well-typed program segment and the maximum memory consumed by it is 5 units.

Now, we analyze the program segment through the Figure 1. and code in Listing 1., we have the following cases:

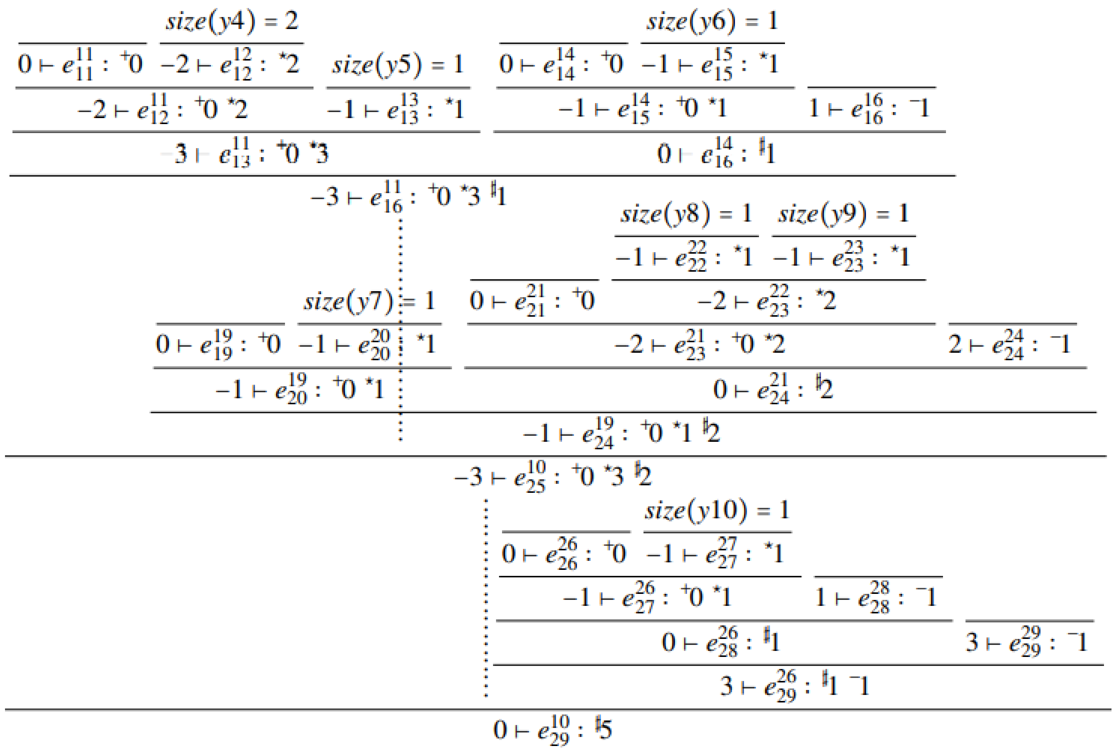


Figure 7. Typing the program segment at lines 10-29 in Listing. 1.

- If $a > b$, the conditional statement will execute e_{16}^{11}, not e_{24}^{19}, so e_{29}^{10} can rewrite into e_{16}^{11}; e_{29}^{26}, and the maximum memory that the program segment needs to use is 4 units.
- If $a \leq b$, the conditional statement will execute e_{24}^{19}, not e_{16}^{11}, so e_{29}^{10} can rewrite into e_{24}^{19}; e_{29}^{26}, and the maximum memory that the program segment needs to use is 2 units.

We realize that the program segment has no path to use up to 5 units (in other words, in this case, the bound is not sharp).

Thus, from the above example, we realize that, if we just based on the expressions in the conditional statement, we can not find the memory bound correctly because it depends on the expression before and after conditional statement.

Currently, we have done a review of the cases that we recognize, and we have found that the memory bound found by the type system is sharp for all rules, except for the branching statement.

6.2. Evaluation of Our Solution

In this work, our proposed method is a static estimation method, based on type theory, which can be proved mathematically to ensure correctness, so the results are reliable.

A special feature of our type system is that it can give type to an uncompleted program, or a program segment. This feature is very useful, because it can give a preview of memory usage patterns while the programmers are typing code.

We tried to find studies close to our work to compare results, but we couldn't find any, so comparing our results with other studies has yet to be done.

In this work, we use an abstract language with the aim to focus on analyzing the behavior of copying shared variables of the STM mechanism. For future work, we plan to apply to real languages and compare with actual memory bounds.

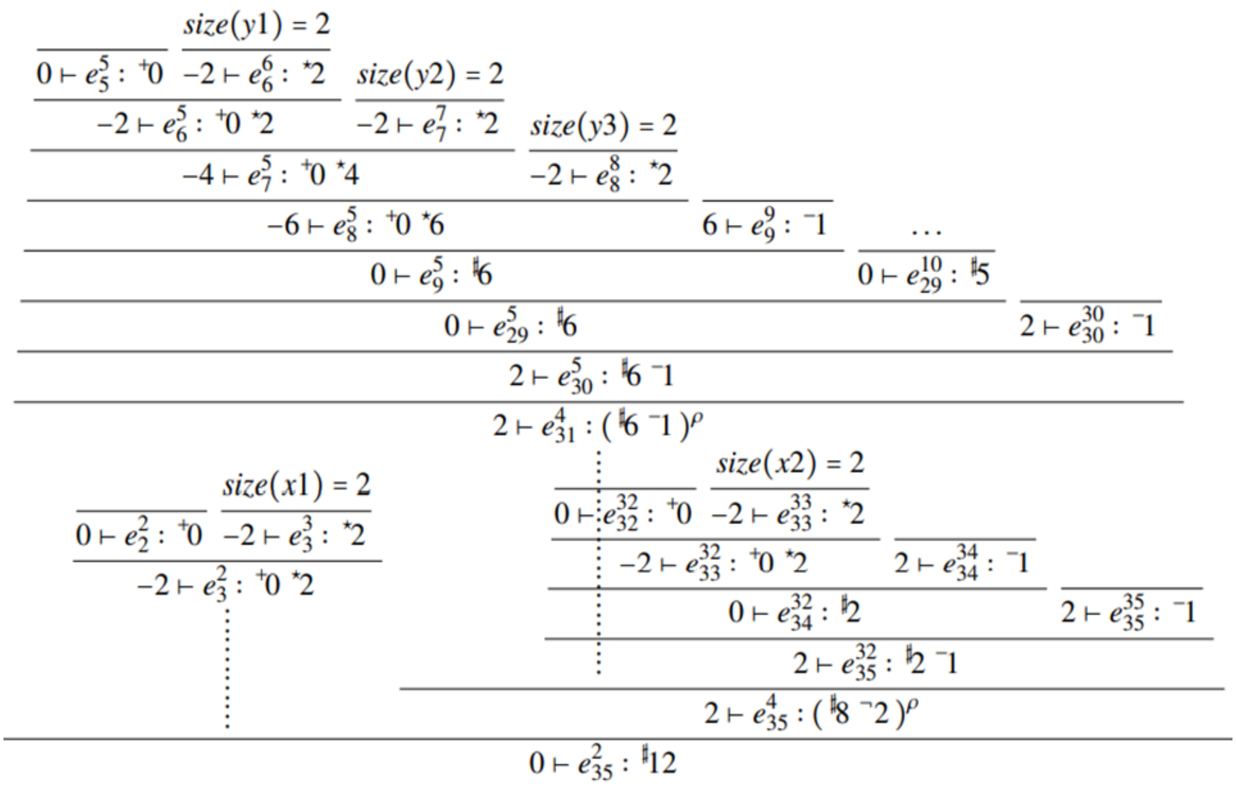


Figure 8. Typing the example program in Listing 1.

7. Conclusions

We present a multi-thread language based on the STM mechanism and a type system for estimating the maximum memory for its programs. In this work, the language and type system are more detailed and rigorous than our previous work, so they are closer to reality.

We added some discussion about the sharpness of the memory bound found by the type system, and evaluations of our proposed solution. This helps users better understand our solution and apply it more effectively.

In our future work, we plan to solve the problems of sharpness of the found memory bound and general loop typing. In this work, our language is still in abstract form to focus on presenting the features of the STM mechanism. We will apply these results to the actual language in future work.

References

[1] H. Weiwu, S. Weisong, T. Zhimin, L. Ming, A Lock-Based Cache Coherence Protocol for Scope Consistency, Journal of Computer Science and Technologydoi: https://doi.org/10.1007/BF02946599.

[2] J. R. Larus, R. Rajwar, Transactional Memory, Commun. ACM 51, (2006), pp. 80–88.

[3] T. Harris, J. Larus, R. Rajwar, Transactional Memory, 2Nd Edition, 2nd Edition, Morgan and Claypool Publishers, 2010.

[4] F. Klein, A. Baldassin, J. Moreira, P. Centoducatte, S. Rigo, R. Azevedo, Stm Versus Lock-Based Systems: An Energy Consumption Perspective, Proceedings Of The 16th ACM/IEEE International Symposium On Low Power Electronics And Design, ISLPED '10, ACM, New York, NY, USA, 2010, pp. 431–436. doi:10.1145/1840845.1840940.
URL http://doi.acm.org/10.1145/1840845.1840940

[5] A. H. Truong, N. K. Nguyen, D. V. Hung, D.-H. Dang, Calculating Statically Maximum Log Memory Used By Multi-Threaded Transactional Programs, Theoretical Aspects of Computing - ICTAC 2016 - 13th International Colloquium, Taipei, Taiwan, ROC, 2016, Proceedings, Lecture Notes in Computer Science, Springer, 2015, pp. 3–27 (to appear).

[6] N. K. Nguyen, A. H. Truong, A Compositional Type Systems for Finding Log Memory Bounds Of Transactional Programs, Proceedings of the Eighth International Symposium on Information and Communication Technology, SoICT 2017, ACM, New York, NY, USA, 2017, pp. 409–416.
http://doi.acm.org/10.1145/3155133.3155183

[7] A. Truong, D. V. Hung, D. Dang, X. Vu, A Type System for Counting Logs of Multi-Threaded Nested Transactional Programs, N. Bjørner, S. Prasad, L. Parida (Eds.), Distributed Computing and Internet Technology - 12th International Conference, ICDCIT 2016, Proceedings, Vol.9581 of LNCS, Springer, 2016, pp. 157–168.
http://dx.doi.org/10.1007/978-3-319-28034-9

[8] X. Vu, T. M. T. Tran, A. Truong, M. Steffen, A Type System for Finding Upper Resource Bounds of Multi-Threaded Programs With Nested Transactions, Symposium on Information and Communication Technology 2012, SoICT '12, Halong City, Quang Ninh, Viet Nam, August 23-24, 2012, pp.21–30.
http://doi.acm.org/10.1145/2350716.2350722

[9] J. C. Mitchell, Type Systems for Programming Languages, Handbook Of Theoretical Computer Science, Volume B: Formal Models and Sematics, 1990.

[10] N. Shavit, D. Touitou, Software Transactional Memory, Symposium on Principles of Distributed Computing, 1995, pp. 204–213. doi:10.1145/224964.224987.

[11] B. Carlstrom, J. Chung, H. Chafi, A. McDonald, C. Minh, L. Hammond, C. Kozyrakis, K. Olukotun, Executing Java Programs with Transactional Memory, Science of Computer Programming 63 (2006) 111–129. doi:10.1016/j.scico.2006.05.006.

[12] T. Harris, K. Fraser, Language Support for Lightweight Transactions, SIGPLAN Not. 49 (4) (2014) 64–78.
http://doi.acm.org/10.1145/2641638.2641654

[13] A. Welc, S. Jagannathan, A. L. Hosking, Transactional Monitors for Concurrent Objects, M. Odersky (Ed.), ECOOP 2004 – Object-Oriented Programming, Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, pp. 518–541.

[14] J. Vitek, S. Jagannathan, A. Welc, A. L. Hosking, A Semantic Framework for Designer Transactions, in: D. Schmidt (Ed.), Programming Languages and Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, pp. 249–263.

[15] T. Harris, S. Marlow, S. Peyton-Jones, M. Herlihy, Composable Memory Transactions, Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '05, ACM, New York, NY, USA, 2005, pp. 48–60. doi:10.1145/1065944.1065952.
URL http://doi.acm.org/10.1145/1065944.1065952

[16] B. Dongol, R. Jagadeesan, J. Riely, Transactions in Relaxed Memory Architectures, Proc. ACM Program. Lang. 2 (POPL) (2017) 18:1–18:29.
http://doi.acm.org/10.1145/3158106

[17] L. Briand, I. Wieczorek, Resource Estimation in Software Engineering, 2002.
doi:10.1002/0471028959.sof282.

[18] M. Bogaerts, New Upper Bounds for the Size of Permutation Codes via Linear Programming, Electr. J. Comb. 17.

[19] J. Hoffmann, A. Das, S.-C. Weng, Towards Automatic Resource Bound Analysis for Ocaml, SIGPLAN Not. 52 (1) (2017) 359–373. http://doi.acm.org/10.1145/3093333.3009842

[20] J. Hoffmann, Z. Shao, Automatic Static Cost Analysis for Parallel Programs, J. Vitek (Ed.), Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Vol. 9032 of LNCS, Springer, 2015, pp. 132–157. http://dx.doi.org/10.1007/978-3-662-46669-8-6

[21] E. Albert, P. Gordillo, A. Rubio, I. Sergey, Gastap: A Gas Analyzer for Smart Contracts, ArXiv abs/1811.10403.

[22] T. Chen, X. Li, X. Luo, X. Zhang, Under-Optimized Smart Contracts Devour Your Money, 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2017, pp. 442-446.

[23] T. A. Hoang, N. N. Khai, A Type System For Counting Logs of a Minimal Language With Multithreaded and Nested Transactions, Journal of Science of HNUE.

[24] N. N. Khai, T. A. Hoang, A Type System For Inferring The Log Memory Of Transactional Program From Shared Variables., Journal of Science and Technology section on Information and Communication Technology.