A type system for calculating the maximum log memory used by transactional programs

Nguyen Ngoc Khai - K22, Software Engineering, Faculty of Information Technology, University of Engineering and Technology, Vietnam National University, Hanoi

Abstract

We developed a type system for estimating an upper bound of memory that multi-threaded and nested transactional programs may require for their transaction logs. In our previous works, we only estimated the maximum number of logs that can coexist by static type and effect systems. This work extends our previous language that allows ones to specify also the size of logs in a transaction, and then develop a type system that can infer the memory bound required by the transaction logs.

Introduction

This work addresses the problem to determine the memory bound of a transactional program during the compile time to ensure that the program can run smoothly without memory overflow errors. To describe the problem more precisely, we use a core language based on [2]. Our language here focuses on features that allows the programmers to mix creating new threads and transactions. We can formulate the problem as follows. Given the memory requirement of each transaction in the program, compute the maximal memory requirement for the whole program, and determine where in the execution of the program the maximal memory requirement is reached. The figure below represent a nested multi-threaded program.

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

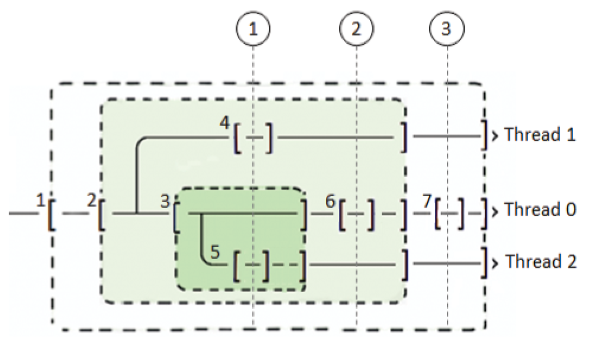

Figure 1: A nested multi-threaded program

Abstract Language TM (Transactional Multi-threaded)

Syntax of the language TM

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

Figure 2: TM syntax

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 size. We formally define the local and global environments as follows.

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 and denoted by ϵ.*

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

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

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

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

$$\frac{i = 1, 2}{\Gamma, P \parallel p(e_1 + e_2) \Rightarrow \Gamma, P \parallel p(e_i)} \text{ S-COND} \qquad \frac{\Gamma = \Gamma'' \cup \{p : E\} \quad [\![E]\!] = 0}{\Gamma, P \parallel p(\mathbf{commit}; e) \Rightarrow error} \text{ S-ERROR}$$

Table 1: TM dynamic semantics

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. logs.

Types

Inspired from our previous works [4], our types are finite sequences over the set of so called tagged numbers. A tagged number is a pair of a tag and a natural number. 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 ${}^{T}\mathbb{N} = \{ {}^{+}n\,, {}^{-}n\,, {}^{\sharp}n\,, {}^{\neg}n \mid n \in \mathbb{N}^{+}\}$.

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 is to reduce a sequence in ${}^{T}\bar{\mathbb{N}}$ to a canonical one. Note the pattern '$+-$' does not appear at the left, but we can insert ${}^{\sharp}0$ to apply. The last two patterns, '$+\neg$' and '$+\sharp\neg$', will be handled by the function jc later.

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

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

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

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

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

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

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:*

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

Typing rules

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

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

The second kind of type S^{ρ} is used for terms inside a `spawn` expression which needs to be synchronized with their parent thread. The treatment of two cases are different, so we denote $kind(T)$ 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 expression 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.

$$\frac{}{-n \vdash \mathbf{onacid(n)} : {}^{+}n} \text{ T-ONACID} \qquad \frac{}{n \vdash \mathbf{commit} : {}^{-}1} \text{ T-COMMIT}$$

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

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

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

Table 2: Typing rules

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

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

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.

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 when combined with the type of the expression 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.*

Conclusion

We have presented a novel type system that can estimate the maximum memory used by transaction logs for a minimal language whose features are multi-threaded and nested transactions. Although the type system in this work have similar type structures to the ones in our previous works [1, 4, 5], the semantics of type elements and typing rules are novel and the size information obtained from well-typed programs are of practical value.

Our next tasks are to implement a type inference tool and to apply the core language to some real-world languages.

References

[1] Marc Bezem, Dag Hovland, and Hoang Truong. A type system for counting instances of software components. *Theor. Comput. Sci.*, 458:29–48, 2012.

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

[3] 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.

[4] 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 *Distributed Computing and Internet Technology - 12th International Conference, ICDCIT 2016, Bhubaneswar, India, January 15-18, 2016, Proceedings*, pages 157–168, 2016.

[5] 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.