Compositional Static Analysis for Implicit Join Synchronization in a Transactional Setting

Thi Mai Thuong Tran[1], Martin Steffen[1], and Hoang Truong[2]*

[1] Department of Informatics, University of Oslo, Norway
[2] University of Engineering and Technology, VNU Hanoi

Abstract. We present an effect-based static analysis to calculate upper bounds on multi-threaded and nested transactions as measure for the resource consumption in an execution model supporting implicit join synchronization. The analysis is compositional and takes into account implicit join synchronizations that arise when more than one thread jointly commit a transaction. Central for a compositional and precise analysis is to capture as part of the effects a tree-representation of the future resource consumption and synchronization points (which we call joining commit trees). The analysis is formalized for a concurrent variant of Featherweight Java extended by transactional constructs. We show the soundness of the analysis.

1 Introduction

Software Transactional Memory (STM) [13,3] has recently been introduced to concurrent programming languages as an alternative for lock-based synchronization, enabling an optimistic form of synchronization for shared memory. *Nested* and *multi-threaded* transactions are advanced features of recent transactional models. Multi-threaded transactions means that inside one transaction there can be more than one thread running in parallel. *Nesting* of transactions means that a parent transaction may contain one or more child transactions which must commit before their parent. Additionally, if a transaction commits, all threads spawned inside must join via a commit. To achieve isolation, each transaction operates via reads and writes on its own local copy of the memory, called log. It is used to record these operations to allow validation or potentially roll-backs at commit time. The logs are a critical factor of memory resource consumption of STM. As each transaction operates on its own log of the variables it accesses, a crucial factor in the memory consumption is the number of thread-local transactional memories (i.e., logs) that may co-exist at the same time in parallel threads. Note that the number of logs neither corresponds to the number of transactions running in parallel (as transactions can contain more than one thread) nor to the number of parallel threads, because of the nesting of transactions. A main complication is that parallel threads do not run independently; instead, executing a commit in a transaction may lead to a form of implicit *join synchronization* with other threads inside the same transaction.

In this paper, we develop a type and effect system for statically approximating the resource consumption in terms of the maximum number of logs of a program. It can be more generally understood as a compositional static analysis of a concurrency model with implicit join synchronization. For the concrete formulation of the analysis, we use a variant of Featherweight Java extended with transactional constructs known as Transactional Featherweight Java (TFJ) [9]. The

* This work was partly supported by the research Project QG.11.29, Vietnam National University, Hanoi.

language features non-lexical starting and ending a transaction, concurrency, choice and sequencing. The analysis is compositional, i.e., syntax-directed. The analysis is *multi-threaded* in that, due to synchronization, it does not analyze each thread in isolation, but needs to take their interaction into account. This complicates the effect system considerably, as the synchronization is implicit in the use of commit-statements and connected to the nestedness of the transactions. To our knowledge, the issue of statically and compositionally estimating the memory resource consumption in such a setting has not been addressed.

The rest of the paper is structured as follows. Section 2 starts by illustrating the execution model and sketching the technical challenges in the design of the effect system. Section 3 introduces the syntax and operational semantics. Section 4 presents an effect system for estimating the resource consumption. The soundness of the analysis is sketched in Section 5. We conclude in Section 6 with related and future work.

2 Compositional analysis of implicit join synchronization

We start by sketching the concurrency model with nested and multi-threaded transactions. The consequences for a compositional analysis of the memory resource consumption are presented informally and by way of examples.

Example 1 (Joining commits). Consider the following (contrived) code snippet.

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

The main expression of thread 0 spawns two new threads 1 and 2. The `onacid`-statement expresses the start of a transaction and `commit` the end. Hence, thread 1 starts its execution at a nesting depth of 2 and thread 2 at depth 3. See also Fig. 1a, where the values of n represent the nesting depth of open transactions at different points in the main thread. We often write in the illustrations and examples [and] for starting resp. committing a transaction. Note that e.g. thread 1 is executing *inside* the first two transactions started by its parent thread and that it uses two commits (after e_1) to close those transactions. Important is that parent and child thread(s) commit an enclosing transaction at the same time, i.e., in a form of join synchronization. We call an occurrence of a commit-statement which synchronizes in that way a *joining commit*. Fig. 1b makes the nesting of transactions more explicit and the right-hand edge of the corresponding boxes marks the joining commits. E.g., e_2 and e_3 cannot execute in parallel since e_2 is sequentialized by a joining commit before e_3 starts. □

Our goal is a compositional, static worst-case estimation of memory resource consumption for the sketched execution model. To achieve isolation, an important transactional property, each thread operates on a local copy of the needed memory which is written back to global memory when and if the corresponding transaction commits; that thread-local and transactional-local memory is called log. We measure the resource consumption at a given point by the *number* of logs co-existing at the same time. This ignores that different logs have different memory needs

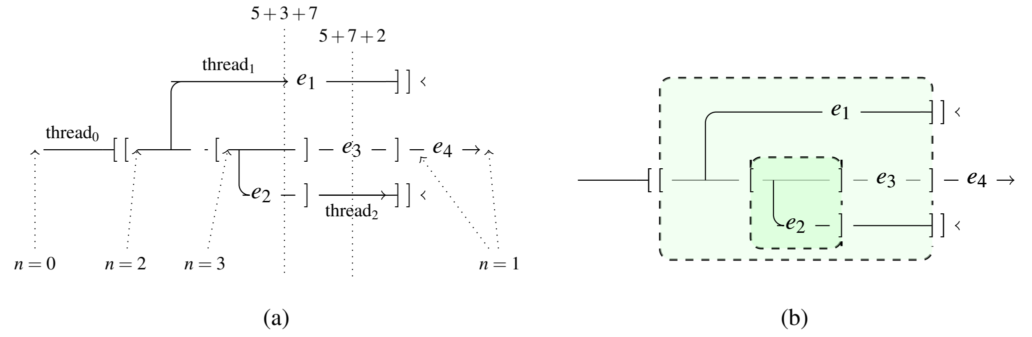


Fig. 1: Nested, multi-threaded transactions and join synchronization

(e.g., accessing more variables transactionally). Abstracting away from this difference, we concentrate on the synchronization and nesting structure underlying the concurrency model. A more fine-grained estimation of resource consumption per log is an orthogonal issue and the corresponding refinement can be incorporated. The refinement would be based on a conservative estimation of the memory consumption per *individual* transaction, which in turn depends on the resource consumption per variable used in the transaction and potentially, dependent on the transactional model, how many times variables are accessed.

Example 2 (Resource consumption). In Example 1, assume that e_1 opens and closes three nested transactions (i.e., is of the form $[\ldots[\ldots[\ldots]\ldots]\ldots]\ldots)$, e_2 four, e_3 five, and e_4 six. The resource consumption after spawning e_2's thread is at most $15 = 5+3+7$ (at the left vertical line): the main thread executes inside three transactions, thread 1 inside five (3 from e_1 plus 2 "inherited" from the parent), and thread 2 inside 7. At the point when thread 0 executes e_3, i.e., after its first commit, the worst case is $14 = 5+7+2$. Note that e_2 cannot run *in parallel* with e_3 whereas e_1 can: the commit before e_3 synchronizes with the commit after e_2 which sequentializes their execution. Thus e_1 still contributes 5, e_2 contributes only 2, and the main thread of e_3 contributes 7 (i.e, 5 from e_3 and 2 from the enclosing transactions). □

To be scalable, the analysis should be *compositional*. In principle, the resource consumption of a *sequential* composition $e_1;e_2$ is approximated by the *maximum* of consumption of its constituent parts. For e_1 and e_2 running (independently) in parallel, the consumption of $e_1 \parallel e_2$ is approximated by the *sum* of the respective contributions. The challenges in our setting are:

Multi-threaded analysis: due to joining commits, threads running in parallel do not necessarily run independently and a sequential composition `spawn` $e_1;e_2$ does not sequentialize e_1 and e_2. They may synchronize, which introduces sequentialization, and to be precise, the analysis must be aware of which program parts can run in parallel and which cannot. Assuming independent parallelism would allow us to analyze each thread in isolation. Such a single-threaded analysis would still yield a sound over-approximation, but would be too imprecise.

Implicit synchronization: Compositional analysis is rendered intricate as the synchronization is *not* explicitly represented syntactically. In particular, there is no clean syntactic separation between sequential and parallel composition. E.g., writing $(e_1 \parallel e_2);e_3$ would make the sequential separation of $e_1 \parallel e_2$ from e_3 explicit and would make a compositional analysis straightforward. Here instead, the sequentialization constraints are entailed by joining commits and it's not explicitly represented with which other threads, if any, a particular commit should synchronize.

Thus, the model has neither independent parallelism nor full sequentialization, but synchronization is affected by the nesting structure of the multi-threaded transactions.

Example 3. Let us split the code of Example 1 after the first spawn to analyze the two parts, say e_l and e_r independently. Writing m for the effect that over-approximates the memory consumption, a rule for sequential composition could resemble the following:

$$\frac{\vdash e_l :: m_1 \qquad \vdash e_r :: m_2 \qquad m = f(m_1, m_2)}{\vdash e_l; e_r :: m}$$

In the schematic rule, $\vdash e :: m$ is read as "expression e has effect m as interface specification". For compositionality, the "interface" information captured in the effects must be rich enough such that m can be calculated from m_1 and m_2. Especially, the upper bound of the overall resource consumption, i.e., the value we are ultimately interested in, is in itself non-compositional. Consider Fig. 2, which corresponds to Fig. 1a except that we separated the contributions of e_l and e_r (by the surrounding boxes). As the execution of e_l partly occurs *before* e_r and partly *in parallel*, m_1 must distinguish the sequential and the parallel contribution of e_1, i.e., the contribution of the spawned thread. Moreover, the parallel part of m_1 is partly synchronized with e_r by joining commits, and thus the effects must contain information about the corresponding synchronization points. Ultimately, the judgments of the effect system use a six-tuple of information that allows a compositional analysis of sequential and parallel composition (plus the other language constructs). A central part of the effect system to achieve compositional analysis is a tree-representation of the future resource consumption and joining commits, which we call jc-trees. □

3 A transactional calculus

Next we present the syntax and semantics of TFJ. We have chosen this calculus as the vehicle for our investigation, as it supports a quite expressive transactional concurrency model, and secondly, it allows us to present the formal semantical analysis in a concise manner. Note, however, that the

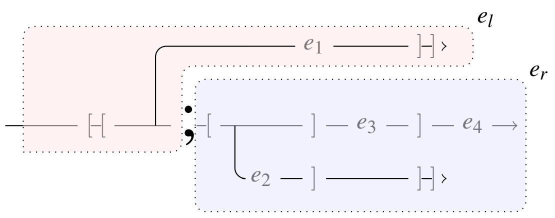


Fig. 2: Compositional analysis (sequential composition $e_l; e_r$)

P	$::= \mathbf{0} \mid P \parallel P \mid p\langle e\rangle$	processes/threads
L	$::= \mathsf{class}\ C\{\vec{f}{:}\vec{T};K;\vec{M}\}$	class definitions
K	$::= C(\vec{f}:\vec{T})\{\mathsf{this}.\vec{f} := \vec{f}\}$	constructors
M	$::= m(\vec{x}{:}\vec{T})\{e\} : T$	methods
e	$::= v \mid v.f \mid v.f := v \mid \texttt{if}\ v\ \texttt{then}\ e\ \texttt{else}\ e$	
	$\mid\ \texttt{let}\ x{:}T = e\ \texttt{in}\ e \mid v.m(\vec{v})$	expressions
	$\mid\ \texttt{new}\ C(\vec{v}) \mid \texttt{spawn}\ e \mid \texttt{onacid} \mid \texttt{commit}$	
v	$::= r \mid x \mid \texttt{null}$	values

Table 1: Abstract syntax

core of our analysis, i.e., a compositional analysis of concurrent threads with join-synchronization does not depend on the concrete choice of language. TFJ as presented here is, with some adaptations, taken from [9]. The main adaptations, as in [10], are: we added standard constructs such as sequential composition (in the form of the let-construct) and conditionals. Besides that, we did not use evaluation-context based rules for the operational semantics, which simplifies the analysis. The underlying type system (without the effects) is standard and omitted here.

3.1 Syntax

Table 1 shows the abstract syntax of TFJ. A program consists of a number of processes/threads $p\langle e\rangle$ running in parallel, where p is the thread's identifier and e the expression being executed. The empty process is written $\mathbf{0}$. The syntactic category L captures class definitions. In absence of inheritance, a class $\mathsf{class}\ C\{\vec{f}{:}\vec{T};K;\vec{M}\}$ consists of a name C, a list of fields $\vec{f}$ with corresponding type declarations $\vec{T}$ (assuming that all f_i's are different), a constructor K, and a list $\vec{M}$ of method definitions. A constructor $C(\vec{f}{:}\vec{T})\{\mathsf{this}.\vec{f} := \vec{f}\}$ of the corresponding class C initializes the fields of instances of that class, these fields are mentioned as the formal parameters of the constructor. We assume that each class has exactly one constructor, i.e., we do not allow constructor overloading. Similarly, we assume that all methods defined in a class have a different name; likewise for fields. A method definition $m(\vec{x}{:}\vec{T})\{e\} : T$ consists of the name m of the method, the formal parameters $\vec{x}$ with their types $\vec{T}$, the method body e, and finally the return type T of the method. In the syntax, v stands for values, i.e., expressions that can no longer be evaluated. In the core calculus, we implicitly assume standard values like booleans, integers, ...; besides those, values can be object references r, variables x or `null`. The expressions $v.f$ and $v_1.f := v_2$ represent field access and field update respectively. Method calls are written $v.m(\vec{v})$ and object instantiation is $\texttt{new}\ C(\vec{v})$. The next two expressions deal with the basic, sequential control structures: conditionals and sequential composition (represented by the let-construct). The language is multi-threaded: $\texttt{spawn}\ e$ starts a new thread of activity which evaluates e in parallel with the spawning thread. Specific for TFJ are the two dual constructs `onacid` and `commit`. The expression `onacid` starts a new transaction and executing `commit` successfully terminates a transaction. For a thread spawned inside a transaction, we impose the following restriction: after a joining commit with its parent, the child thread is not allowed to start another transaction. This restriction is imposed to simplify the analysis later and is not a real restriction in practice as one can transform programs easily to adhere to that convention (at the expense of spawning further threads).

3.2 Semantics

The operational semantics of TFJ is given in two different levels: a local and a global one. The local semantics of Table 2 deals with the evaluation of *one expression/thread* and reducing configurations $E \vdash e$. Local transitions are thus of the form $E \vdash e \to E' \vdash e'$, where e is one expression and E a *local environment.* Note that in the chosen presentation, the expression starts uniformly with a let and the redex is always the left expression of the let construct. Locally, the relevant commands only concern the current thread and consist of reading, writing, invoking a method, and creating new objects.

Definition 1 (Local environment). *A* local environment *E of type LEnv is a finite sequence of the form $l_1{:}\rho_1, \ldots, l_k{:}\rho_k$, i.e., of pairs of transaction labels l_i and a corresponding* log *ρ_i. We write $|E|$ for the size of the local environment, i.e., the number of pairs $l{:}\rho$ in the local environment.*

Transactions are identified by labels l, and as transactions can be nested, a thread can execute "inside" a number of transactions. So, the E in the above definition is ordered, where e.g. l_k to the right refers to the inner-most transaction, i.e., the one most recently started and committing removes bindings from right to left. For a thread with local environment E, the number $|E|$ represents the nesting depth of the thread, i.e., how many transactions the thread has started but not yet committed. The corresponding *logs* ρ_i can be thought of as "local copies" of the heap. The log ρ_i, a sequence of mappings from references to values, is used to keep track of changes by a thread in transaction l_i.

The first four rules deal straightforwardly with the basic, sequential control flow. Unlike the first four rules, the remaining ones do access the heap. Thus, the local environment E is consulted to look up object references and then *changed* in the step. The access and update of E is given *abstractly* by corresponding access functions *read*, *write*, and *extend* (which look-up a reference, update a reference, resp. allocate a new reference on the heap). Note that also the *read*-function actually *changes* the environment from E to E' in the step. The reason is that in a transaction-based implementation, read-access to a variable may be *logged*, i.e., remembered appropriately, to be able to detect conflicts and to do a roll-back if necessary. The premises assume that the class table is given implicitly where $\mathit{fields}(C)$ looks up fields of class C and $\mathit{mbody}(C,m)$ looks up the method m of class C. Otherwise, the rules for field look-up, field update, method calls, and object instantiation are standard.

The rules of the *global* semantics are given in Table 3. The semantics works on configurations of the form $\Gamma \vdash P$, where P is a *program* and Γ is a global environment. Besides that, we need a special configuration *error* representing an error state. Basically, a program P consists of a number of threads evaluated in parallel (cf. Table 1), where each thread corresponds to one expression, whose evaluation is described by the local rules. Now describing the behavior of a number of (labeled) threads or processes $p\langle e\rangle$, we need one E for each thread p. This means, Γ is a "sequence" (or rather a set) of $p{:}E$ bindings where p is the name of a thread and E is its corresponding local environment.

Definition 2 (Global enviroment). *A* global environment *Γ of type GEnv is a finite mapping, written as $p_1{:}E_1, \ldots, p_k{:}E_k$, from threads names p_i to local environments E_i (the order of bindings plays no role, and each thread name can occur at most once).*

So global steps are of the form:

$$\Gamma \vdash P \Longrightarrow \Gamma' \vdash P' \quad \text{or} \quad \Gamma \vdash P \Longrightarrow \mathit{error}\ . \tag{1}$$

$$E \vdash \texttt{let}\, x : T = v \;\texttt{in}\; e \rightarrow E \vdash e[v/x] \quad \text{R-RED}$$

$$E \vdash \texttt{let}\, x_2 : T_2 = (\texttt{let}\, x_1 : T_1 = e_1 \;\texttt{in}\; e) \;\texttt{in}\; e' \rightarrow E \vdash \texttt{let}\, x_1 : T_1 = e_1 \;\texttt{in}\; (\texttt{let}\, x_2 : T_2 = e \;\texttt{in}\; e') \quad \text{R-LET}$$

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

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

$$\frac{\mathit{read}(E,r) = E', C(\vec{u}) \quad \mathit{fields}(C) = \vec{f}}{E \vdash \texttt{let}\, x{:}T = r.f_i \;\texttt{in}\; e \rightarrow E' \vdash \texttt{let}\, x{:}T = u_i \;\texttt{in}\; e} \;\text{R-LOOKUP}$$

$$\frac{\mathit{read}(E,r) = E', C(\vec{r}) \quad \mathit{fields}(C) = \vec{f} \qquad \mathit{write}(r \mapsto (C(\vec{r})[f_i \mapsto r']), E') = E''}{E \vdash \texttt{let}\, x{:}T = r.f_i := r' \;\texttt{in}\; e \rightarrow E'' \vdash \texttt{let}\, x{:}T = r' \;\texttt{in}\; e} \;\text{R-UPD}$$

$$\frac{\mathit{read}(E,r) = E', C(\vec{u}) \quad \mathit{mbody}(C,m) = (\vec{x}, e)}{E \vdash \texttt{let}\, x{:}T = r.m(\vec{r}) \;\texttt{in}\; e' \rightarrow E' \vdash \texttt{let}\, x : T = e[\vec{r}/\vec{x}][r/\mathsf{this}] \;\texttt{in}\; e'} \;\text{R-CALL}$$

$$\frac{r\; \mathit{fresh} \quad E' = \mathit{extend}(r \mapsto C(\vec{u}), E)}{E \vdash \texttt{let}\, x{:}T = \texttt{new}\, C(\vec{u}) \;\texttt{in}\; e \rightarrow E' \vdash \texttt{let}\, x = r \;\texttt{in}\; e} \;\text{R-NEW}$$

Table 2: Semantics (local)

Also the global steps make use of a number of functions accessing and changing the (this time global) environment. As before, some semantical functions are left abstract. However, their *abstract properties* relevant for proving soundness of our analysis are given later in Definition 3 after discussing the global rules. Note further, that two specific implementations of those functions (an optimistic and a pessimistic) have been given in [9]. As the functions' concrete details are irrelevant for our *static* analysis, we refer the interested reader to [9] for possible concretizations of the semantics. Rule G-PLAIN simply *lifts* a local step to the global level, using the reflect-operation, which roughly makes local updates of a thread globally visible; the premise $\Gamma \vdash p{:}E$ means $p{:}E \in \Gamma$. Rule G-SPAWN deals with starting a thread. The next three rules treat the two central commands of the calculus, those dealing with the transactions. The first one G-TRANS covers `onacid`, which starts a transaction. The *start* function creates a new label l in the local environment E of thread p. The two rules G-COMM and G-COMM-ERROR formalize the successful commit resp. an erronous use of the commit-statement outside any transaction. In G-COMM, l is the label of the transaction to be committed and the function $\mathit{intranse}(\Gamma, l)$ finds the identities $p_1, \ldots, p_k$ of all concurrent threads in the transaction l and which *all* join in the commit. In the erroneous case of G-COMM-ERROR, the local environment E is empty; i.e., the thread executes a commit outside of any transaction, which constitutes an error.

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

1. *The function reflect satisfies the following condition: if reflect$(p, E, \Gamma) = \Gamma'$ and $\Gamma = p_1{:}E_1, \ldots, p_k{:}E_k$, then $\Gamma' = p_1{:}E'_1, \ldots, p_k{:}E'_k$ with $|E_i| = |E'_i|$ (for all i).*
2. *The function spawn satisfies the following condition: Assume $\Gamma = p{:}E, \Gamma''$ and $p' \notin \Gamma$ and spawn$(p, p', \Gamma) = \Gamma'$, then $\Gamma' = \Gamma, p'{:}E'$ s.t. $|E| = |E'|$.*
3. *The function start satisfies the following condition: if start$(l, p_i, \Gamma) = \Gamma'$ for $\Gamma = p_1{:}E_1, \ldots, p_i{:}E_i, \ldots, p_k{:}E_k$ and for a fresh l, then $\Gamma' = p_1{:}E_1, \ldots, p_i{:}E'_i, \ldots, p_k{:}E_k$, with $|E'_i| = |E_i| + 1$.*
4. *The function intranse satisfies the following condition: Assume $\Gamma = \Gamma'', p{:}E$ s.t. $E = E', l{:}\rho$ and intranse$(\Gamma, l) = \vec{p}$, then*
   (a) *$p \in \vec{p}$ and*
   (b) *for all $p_i \in \vec{p}$ we have $\Gamma = \ldots, p_i : (E'_i, l{:}\rho_i), \ldots$.*

$$\frac{\Gamma \vdash p : E \qquad E \vdash e \to E' \vdash e' \qquad \mathit{reflect}(p, E', \Gamma) = \Gamma'}{\Gamma \vdash P \parallel p\langle e \rangle \Longrightarrow \Gamma' \vdash P \parallel p\langle e' \rangle}\ \text{G-PLAIN}$$

$$\frac{p' \mathit{\ fresh} \qquad \mathit{spawn}(p, p', \Gamma) = \Gamma'}{\Gamma \vdash P \parallel p\langle \mathtt{let}\ x : T = \mathtt{spawn}\ e_1\ \mathtt{in}\ e_2 \rangle \Longrightarrow \Gamma' \vdash P \parallel p\langle \mathtt{let}\ x : T = \mathtt{null}\ \mathtt{in}\ e_2 \rangle \parallel p'\langle e_1 \rangle}\ \text{G-SPAWN}$$

$$\frac{l \mathit{\ fresh} \qquad \mathit{start}(l, p, \Gamma) = \Gamma'}{\Gamma \vdash P \parallel p\langle \mathtt{let}\ x : T = \mathtt{onacid}\ \mathtt{in}\ e \rangle \Longrightarrow \Gamma' \vdash P \parallel p\langle \mathtt{let}\ x : T = \mathtt{null}\ \mathtt{in}\ e \rangle}\ \text{G-TRANS}$$

$$\frac{\begin{array}{c}\Gamma = \Gamma'', p{:}E \qquad E = E', l{:}\rho \qquad \mathit{intranse}(\Gamma, l) = \vec{p} = p_1 \dots p_k \\ \mathit{commit}(\vec{p}, \vec{E}, \Gamma) = \Gamma' \quad p_1{:}E_1, p_2{:}E_2, \dots p_k{:}E_k \in \Gamma \quad \vec{E} = E_1, E_2, \dots, E_k\end{array}}{\Gamma \vdash P \parallel \dots \parallel p_i\langle \mathtt{let}\ x : T_i = \mathtt{commit}\ \mathtt{in}\ e_i \rangle \parallel \dots \Longrightarrow \Gamma' \vdash P \parallel \dots \parallel p_i\langle \mathtt{let}\ x : T_i = \mathtt{null}\ \mathtt{in}\ e_i \rangle \parallel \dots}\ \text{G-COMM}$$

$$\frac{\Gamma = \Gamma'', p{:}E \qquad E = \emptyset}{\Gamma \vdash P \parallel p\langle \mathtt{let}\ x : T = \mathtt{commit}\ \mathtt{in}\ e \rangle \Longrightarrow \mathit{error}}\ \text{G-COMM-ERROR}$$

Table 3: Semantics (global)

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

5. *The function commit satisfies the following condition: if $commit(\vec{p}, \vec{E}, \Gamma) = \Gamma'$ for $\Gamma = \Gamma''$, $p{:}(E, l{:}\rho)$ and for a $\vec{p} = intranse(\Gamma, l)$ then $\Gamma' = \dots, p_j{:}E'_j, \dots, p_i{:}E'_i, \dots$ where $p_i \in \vec{p}, p_j \notin \vec{p}$, $p_j{:}E_j \in \Gamma$, with $|E'_j| = |E_j|$ and $|E'_i| = |E_i| - 1$.*

4 Effect system

Next we present our analysis as an effect system. The underlying types T include names C of classes, basic types B (natural numbers, booleans, etc.) and Void. The underlying type system for judgments of the form $\Gamma \vdash e : T$ ("under type assumptions Γ, expression e has type T") is standard and therefore omitted here.

Thread-local effects, sequential composition, and joining commits On the local level, the judgments of the effect part are of the following form:

$$n_1 \vdash e :: n_2, h, l, \vec{t}, S\ , \tag{2}$$

where n_1, n_2, h, and l are natural numbers with the following interpretation. n_1 and n_2 are the pre- and post-condition for the expression e, capturing the current nesting depth: starting at a nesting depth of n_1, the depth is n_2 after termination of e. We call the numbers n_1 resp. n_2 the current balance of the thread before and after execution. Starting from the pre-condition n_1, the numbers h and l approximate the maximum resp., the minimum value of the balance *during* the execution of e. Executing e, however, may spawn new child threads and the remaining elements $\vec{t}$ and S take their contribution into account. Roughly speaking, the information S is needed to achieve compositionality wrt. sequential composition and $\vec{t}$ for compositionality wrt. parallel composition.

The S-part represents the resources of threads being spawned in e, more precisely their resource consumption *after* e. S is needed when considering e in a sequential composition with a trailing expression. E.g., in the sequential composition of Figure 2, the S of the left expression corresponds to the part of the left box which overlaps with the trailing expression on the right. Depending on the nesting depth at the point of being spawned, a thread may or may not be synchronized by a joining commit in the trailing expression. E.g., splitting the program of Figure 1a after the second spawn and before the first commit, this commit affects only the thread of e_2 but not the one of e_1. To distinguish the two situations, S must contain, for each thread, the thread's nesting depth at the point it is spawned. Thus, S is of the form $\{(p_1,c_1),(p_2,c_2),\ldots\}$, i.e., a multi-set of pairs of natural numbers. For all spawned threads, S keeps its maximal contribution to the resource consumption at the point after e, i.e., (p_i,c_i) represents that the thread i can have maximally a resource need of p_i+c_i, where p_i represents the contribution of the spawning thread ("parent"), i.e., the nesting depth at the point when the thread is being spawned, and c_i the additional contribution of the child threads themselves. In contrast, $\vec{t}$ is needed for compositionality wrt. parallel composition. The $\vec{t}$ is a sequence of non-negative numbers, representing the maximal, overall ("total") resource consumption during the execution of e, including the contribution of all threads (the current and the spawned ones) separated by joining commits of the main thread. We call $\vec{t}$ a joining-commit sequence, or *jc-sequence* for short. In Example 3, the right-hand expression $[\texttt{spawn}\ (e_2]]])]e_3]e_4$ has one joining commit, i.e., the jc-sequence is of length 2. Assuming that the execution of the expression starts at nesting depth 2 (as is the case at the end of the left-hand expression) the jc-sequence is $\vec{t} = 10, 7$ (where $10 = ((4+3)+3) \vee ((5+2)+2)$ and $7 = 6+1$). For uniformity, we use $\vee$ resp. $\wedge$ not only for the least upper bound resp. greatest lower bound in general, but also for the maximum, resp. the minimum of natural numbers.

The rules for expressions are shown in Table 4. The rules for variables, the null reference, for field look-up and field update, and for object instantiation are omitted (cf. [11]), as they neither affect the balance nor is any other thread involved. Note that not "counting" the resource consumption of these operations reflects the decision, as stated earlier, that we simply use the number of logs running in parallel as measure for memory consumption. The committing in rule T-COMMIT similarly keeps the maximal value constant. Considered in isolation, the `commit` is a joining commit, and hence $\vec{t}$ has two elements, where the resource consumption is decreased by one after the commit.

The treatment of sequential composition is more complicated, for the reasons explained in Section 2. In particular, calculating the jc-sequence $\vec{u}$ and the parallel weight S for the composed expression from the corresponding information in the premises is intricate. The following two definitions formalize the necessary calculations:

Definition 4 (Parallel weight). *Let S be a multi-set of the form $\{(p_1,c_1),\ldots,(p_k,c_k)\}$ where the p_i, c_i, and l are natural numbers. The overall parallel weight of S is defined as $|S| = \sum_i (p_i+c_i)$. Furthermore we define the following functions:*

$$\begin{aligned} par(S,l) &= \{(p,c)\in S \mid p\le l\} \qquad & seq(S,l) &= \{(p,c)\in S \mid p > l\}\ . \\ \lfloor S\rfloor_l &= \{(l,0),(l,0),\ldots\} & S\downarrow_l &= par(S,l)\cup\lfloor seq(S,l)\rfloor_l \end{aligned} \tag{3}$$

where for $\lfloor S\rfloor_l$, the number of tuples in S equals the number of $(l,0)$ in $\lfloor S\rfloor_l$.

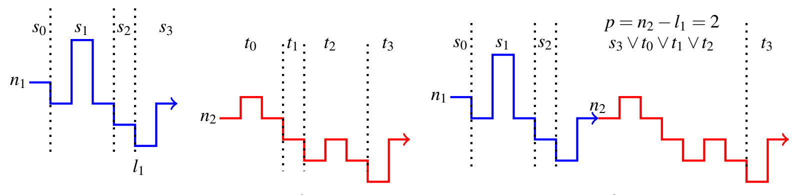
To determine S in T-LET, the spawned weight S_1 of e_1 is split into two halves:

1. The part $par(S_1,l_2)$ of S_1 unaffected by a commit in e_2 and thus able to run in parallel with e_2.
2. The part $seq(S_1,l_2)$ of S_1 affected by a commit in e_2 via a join synchronization.

The parallel weight S_1 of e_1 is a multi-set of pairs (p_i, c_i), one pair for each spawned thread, where the first element p_i of the pair represents the balance of the parent thread at the time of the spawning, i.e., the nesting depth inherited from the parent thread. Whether the contribution (p_i, c_i) of a thread spawned in e_1 counts as being composed in parallel or affected by a join synchronization with e_2 depends on whether e_2 does a commit which closes a transaction *containing* the thread of (p_i, c_i). The $par(S_1, l_2)$ consists of the half of S_1 unaffected by any join synchronization. Even if $seq(S_1, l_2)$ in contrast synchronizes via joining commits in e_2, it still contributes to the resource consumption *after* e_2, because transactions may be nested, and after the joining synchronization, the rest of a spawned thread still consumes resources corresponding to the not-yet-committed parent transactions. The operation $\lfloor seq(S_1, l_2) \rfloor_{l_2}$ calculates that remaining contribution. So $\lfloor S_1 \rfloor_{l_2}$ contains the consumption *after* e_1 of threads spawned *during* e_1. In the conclusion of T-LET, that estimation is added to e_2's own contribution S_2 by multi-set union, resulting in $S_1 \downarrow_{l_2} \cup S_2$ overall. The correctness of the calculation in T-LET depends on the restriction that once a spawned thread commits a transaction inherited from its parent thread, it will not open another transaction. Note, however, that corresponds to the standard semantics of the explicit join-construct, e.g., in Java, letting the caller wait for the termination of the thread it intends to "join".

Definition 5 (Sequential composition of jc-sequences). *Let $\vec{s} = s_0, \ldots, s_k$, $\vec{t} = t_0, \ldots, t_m$, and $m \geq p \geq 0$. Then $\vec{s} \oplus_p \vec{t}$ is defined as: $\vec{s} \oplus_p \vec{t} = s_0, \ldots, (s_k \vee t_0 \ldots \vee t_p), t_{p+1}, \ldots, t_m$. Given a parallel weight S and a $n \geq m \geq 0$, then $\otimes_n$ is defined as $S \otimes_n \vec{t} = t'_0, t'_1, \ldots, t'_m$ where $t'_0 = t_0 + |S|$, $t'_1 = t_1 + |\lfloor S \rfloor_{n-1}|$, $\ldots$, $t'_m = t_m + |\lfloor S \rfloor_{n-m}|$.*

The compositional calculation of the jc-sequence $\vec{u}$ (cf. Definition 5) takes care of two phenomena: Firstly, the parallel weight S_1 at the end of e_1 may increase the resource consumption of the jc-sequence $\vec{t}$. This is formalized by the $\otimes_{\_}$ operation of Definition 5. Secondly, joining commits of e_2 may no longer be joining commits of the composed expression `let` $x = e_1$ `in` e_2. For instance, in Example 3, the (only) joining commit of e_r (the one separating e_3 from e_4) is no longer a joining commit of $e_l; e_r$, as it cannot synchronize with anything outside the composed expression. The calculation of the composed jc-sequence from the constituent ones as $\vec{s} \oplus_p \vec{t}$ "merges" an appropriate number of elements from $\vec{t}$ (using $\vee$) depending on how many joining commits disappear in the composition. This number p is given by $n_2 - l_1$. See also the illustration in Fig. 3, where the respective joining commits are indicated by the vertical, dotted lines. So in rule T-LET, the overall $\vec{u}$ is given as $\vec{s} \oplus_p (S_1 \otimes_{n_2} \vec{t})$. The calculation of the remaining effects in T-LET is straightforward: given the balance n_1 as pre-condition, the post-condition n_2 of e_1



(a) $\vec{s}$ of e_1 (b) $\vec{t}$ of e_2 (c) $\vec{s} \oplus_p \vec{t}$ of $e_1; e_2$

Fig. 3: Sequential composition of jc-sequences (cf. Definition 5)

$$\frac{}{n \vdash \texttt{onacid}\!:: n+1, n+1, n, [n+1], \emptyset}\ \text{T-ONACID} \qquad \frac{n \geq 1}{n \vdash \texttt{commit}\!:: n-1, n, n-1, [n; n-1], \emptyset}\ \text{T-COMMIT}$$

$$\frac{\begin{array}{c} n_1 \vdash e_1 :: n_2, h_1, l_1, \vec{s}, S_1 \qquad n_2 \vdash e_2 :: n_3, h_2, l_2, \vec{t}, S_2 \\ h = h_1 \vee h_2 \qquad l = l_1 \wedge l_2 \qquad p = n_2 - l_1 \qquad S = S_1 \downarrow_{l_2} \cup S_2 \qquad \vec{u} = \vec{s} \oplus_p (S_1 \otimes_{n_2} \vec{t}) \end{array}}{n_1 \vdash \texttt{let}\ x{:}T = e_1\ \texttt{in}\ e_2 :: n_3, h, l, \vec{u}, S}\ \text{T-LET}$$

$$\frac{n \vdash e_1 :: n', h_1, l_1, \vec{s}, S_1 \qquad n \vdash e_2 :: n', h_2, l_2, \vec{t}, S_2}{n \vdash \texttt{if}\ \ v\ \ \texttt{then}\ e_1\ \texttt{else}\ e_2 :: n', h_1 \vee h_2, l_1 \wedge l_2, \vec{s} \vee \vec{t}, S_1 \sqcup S_2}\ \text{T-COND} \qquad \frac{n_1 \vdash e :: 0, h, 0, \vec{s}, S}{n_1 \vdash \texttt{spawn}\ e :: n_1, n_1, n_1, [n_1 + s_0], S \cup \{(n_1, h - n_1)\}}\ \text{T-SPAWN}$$

$$\frac{mtype(C,m) :: n_1' \to n_2', h, l, \vec{t}, S \qquad n_1' \leq n_1 \qquad n = n_1 - n_1'}{n_1 \vdash v.m(\vec{v}) :: n_2' + n, h + n, l + n, \vec{t} + n, S + n}\ \text{T-CALL}$$

Table 4: Effect system

serves as pre-condition for the subsequent e_2, whose post-balance n_3 gives the corresponding final post-balance. The values h and l are calculated by the least upper bound, resp., the greatest lower bound of the corresponding numbers of e_1 and e_2. The treatment of h, l, and of the current balance is simple because the syntax of sequential composition reflects and separates the contributions of e_1 and e_2. The treatment of conditionals in rule T-COND is comparatively simple, after having defined an appropriate order on the jc-sequences and the parallel weights.

Definition 6 (Order). *The order relation on jc-sequences (of equal length) $\vec{s} \leq \vec{t}$ is defined pointwise and we write $\vec{s} \vee \vec{t}$ for the corresponding least upper bound. For parallel weights, the order $S_1 \sqsubseteq S_2$ is defined as follows. For pairs of natural numbers and in abuse of notation, $(p_1, c_1) \sqsubseteq (p_2, c_2)$ iff $p_1 = p_2$ and $c_1 \leq c_2$. Then for $S_1 = \{(p_1, c_1), \ldots, (p_k, c_k)\}$ and $S_2 = \{(p_1', c_1'), \ldots, (p_k', c_k'), (p_{k+1}', c_{k+1}'), \ldots\}$, $S_1 \sqsubseteq S_2$ if $(p_i, c_i) \sqsubseteq (p_i', c_i')$, for all $1 \leq i \leq k$. We write $S_1 \sqcup S_2$ for the corresponding least upper bound of S_1 and S_2 wrt. $\sqsubseteq$.*

When spawning a new thread e (cf. rule T-SPAWN), the pre-condition n_1 remains unchanged, as the effect of e as determined by the premise does not concern the current, i.e., spawning thread. Likewise, the maximal and minimal value are simply n_1, as well. The jc-sequence of total resource consumption takes into account the contribution s_0 of the spawned thread before *its* first joining commit plus the resource consumption n_1 of the current thread. Finally, the parallel weight S of the spawned expression is increased by the maximal value h of e's thread, where that contribution is split into the "inherited" part n_1 and the rest $h - n_1$. The effect of a method call $v.m(\vec{v})$ (cf. T-CALL) is given by the interface information of method m in class C appropriately increased by the difference n of the balance n_1 at the call-site and the specified pre-condition n_1'; the interface information for the method is looked up using *mtype* in the given class table (the function is standard and its definition is omitted here). The appropriate adapation of the interface information concerning $\vec{t}$ and S is defined as follows:

Definition 7 (Shift). *Given a natural number n, the addition $\vec{t} + n$ on a jc-sequence $\vec{t}$ is defined point-wise. For parallel weights $S = \{(p_1, c_1), \ldots, (p_k, c_k)\}$, $S + n$ is defined as $\{(p_1 + n, c_1), \ldots, (p_k + n, c_k)\}$.*

Global effects, parallel composition, and joining commit trees The rest of the section is concerned with formalizing the resource analysis on the global level, in essence, capturing the parallel composition of threads (cf. Table 5 below). The key is again to find an appropriate representation of the resource effects which is compositional wrt. parallel composition. At the local level, one key was to capture the synchronization point of a thread in what we called *jc-sequences*. Now that more than one thread is involved, that data-structure is generalized to *jc-trees* which are basically finitely branching, finite trees where the nodes are labeled by a transaction label and an integer. With t as jc-tree, the judgments at the global level are of the following form: $\Gamma \vdash P :: t$.

Definition 8 (Jc-tree). Joining commit trees *(or jc-trees for short) are defined as tree of type* $\mathsf{JCtree} = \mathsf{Node\ of\ Nat} \times \mathsf{Lab} \times (\mathsf{List\ of JCtree})$*, with typical element* t*. We write* $\vec{t}$ *for lists of jc-trees. We write also* $[]$ *for the empty list, and* $\mathsf{Node}(n,l,\vec{t})$ *for a jc-tree whose root carries the natural number* n *as weight and* l *as label, and with children* $\vec{t}$*.*

Definition 9 (Weight). *The* weight *of a jc-tree is given inductively as* $|\mathsf{Node}(n,l,\vec{t})| = n \vee \sum_{i=1}^{|\vec{t}|}(|t_i|)$*. The* initial *weight of a join tree* t*, written* $|t|_1$*, is the weight of its leaves.*

Definition 10 (Parallel merge). *We define the following two functions* $\otimes_1$ *of type* $\mathsf{JCtree} \times \mathsf{JCForest} \to \mathsf{JCForest}$ *and* $\otimes_2$ *of type* $\mathsf{JCForest}^2 \to \mathsf{JCForest}$ *by mutual induction. In abuse of notation, we will write* $\otimes$ *for both in the following.*

$$
\begin{aligned}
t \otimes_1 [] &= [t] \\
\mathsf{Node}(n_1,l,f_1) \otimes_1 (\mathsf{Node}(n_2,l,f_2) :: f) &= \mathsf{Node}(n_1+n_2,l,f_1 \otimes_2 f_2) :: f \\
\mathsf{Node}(n_1,l_1,f_1) \otimes_1 (\mathsf{Node}(n_2,l_2,f_2) :: f) &= \mathsf{Node}(n_2,l_2,f_2) :: (\mathsf{Node}(n_1,l_1,f_1) \otimes_1 f) \qquad l_1 \neq l_2
\end{aligned}
$$

$$
\begin{aligned}
[] \otimes_2 f &= f \\
t :: f_1 \otimes_2 f_2 &= f_1 \otimes_2 (t \otimes_1 f_2)
\end{aligned}
$$

Remember from Definition 1, that local environments are of the form $l_1{:}\rho_1,\ldots,l_k{:}\rho_k$. In the semantics, the transaction labelled l_k is the inner-most one.

Definition 11 (Lifting). *The function lift of type LEnv* $\times\ \mathsf{Nat}^+ \to \mathsf{JCtree}$ *is given inductively as:*

$$
\begin{aligned}
\mathit{lift}([],[n]) &= \mathsf{Node}(n,\bot,[]) \\
\mathit{lift}((l{:}\rho :: E),\vec{s} :: n) &= \mathsf{Node}(n,l,[\mathit{lift}(E,\vec{s})]) \ .
\end{aligned}
$$

Note that the function is undefined if $|E| \neq |\vec{s}| - 1$. It is an invariant of the semantics, that $|E| = |\vec{s}| - 1$, and hence the function is well-defined for all reachable configurations. Defining the weight (and in abuse of notation) of a jc-sequence $\vec{s}$ as the maximum of their elements, we obviously have $|\vec{s}| = |\mathit{lift}(E,\vec{s})|$.

$$
\frac{|E| \vdash e :: n,h,l,\vec{s},S \qquad t = \mathit{lift}(E,\vec{s})}{p{:}E \vdash p\langle e\rangle \ :: \ t}\ \textsc{T-Thread}
\qquad
\frac{\Gamma_1 \vdash P_1 : t_1 \qquad \Gamma_2 \vdash P_2 : t_2}{\Gamma_1,\Gamma_2 \vdash P_1 \parallel P_2 : t_1 \otimes_2 t_2}\ \textsc{T-Par}
$$

Table 5: Effect system

5 Correctness

This section establishes the soundness of the analysis, i.e., that the static estimation over-approximates the actual potential resource consumption for all reachable configurations. We start by defining the actual resource consumption of a program:

Definition 12 (Resource consumption). *The weight of a local environment E, written $|E|$ is defined as its length, i.e., the number of its $l{:}\rho$-bindings. The weight of a global environment Γ, written $|\Gamma|$ is defined as the sum of weights of its local environments.*

The following lemmas establish a number of facts about the operations used in the calculation of resource consumption needed later. The proofs, omitted here for lack of space, can be found in the technical report [11]. The next two lemmas show that the way the resource consumption is calculated in the let-rule is associative, which is a crucial ingredient in subject reduction.

Lemma 1 (Associativity of parallel weight). *Let S_1, S_2 be parallel weights and l be a non-negative natural number. Define the function f as $f(S_1,l,S_2) = S_1 \downarrow_l \cup S_2$. Then $f(f(S_1,l_2,S_2),l_3,S_3) = f(S_1,l_2 \wedge l_3, f(S_2,l_3,S_3))$.*

Lemma 2 (Associativity of $\oplus$ and $\otimes$). *Let $l_1 = n_1 - |s| + 1$, $l_2 = n_2 - |\vec{t}| + 1$, $p_1 = n_2 - l_1$, and $p_2 = n_3 - l_2$. Then $\vec{s} \oplus_{p_1} (S_1 \otimes_{n_2} (\vec{t} \oplus_{p_2} (S_2 \otimes_{n_3} \vec{u}))) = (\vec{s} \oplus_{p_1} (S_1 \otimes_{n_2} \vec{t})) \oplus_{p_2} ((S_2 \cup S_1 \downarrow_{l_2}) \otimes_{n_3} \vec{u})$.*

The order on trees is defined "point-wise" in that the smaller tree must be a sub-tree (respecting the labelling) of the larger one and furthermore each node of the smaller tree with weight w_1 is represented by the corresponding node with a weight $w_2 \geq w_1$.

Definition 13 (Order on trees). *We define the binary relation $\leq$ on jc trees inductively as follows: $\mathsf{Node}(n,l,\vec{s}) \leq \mathsf{Node}(m,l,\vec{t})$ if $n \leq m$ and for each tree s_i in $\vec{s}$, there exists a t_j in $\vec{t}$ such that $s_i \leq t_j$. (Note that the labels l in a jc tree are unique.)*

Lemma 3 (Lifting of ordering). *If $\vec{s} \leq \vec{t}$ (as comparison between jc-sequences), then lift$(E,\vec{s}) \leq$ lift$(E,\vec{t})$ (as comparison between jc trees).*

Lemma 4 (Lifting and commit). *lift$(E, l{:}\rho, n :: \vec{u}) \geq$ lift$(E,\vec{u})$.*

Lemma 5 (Monotonicity). *If $t_1 \leq t_1'$ and $t_2 \leq t_2'$, then $(t_1 \otimes t_2) \leq (t_1' \otimes t_2')$.*

Next we prove preservation of well-typedness under reduction, i.e., subject reduction, split into two parts, preservation under local resp. global reduction steps.

Lemma 6 (Subject reduction (local)). *If $n_1 \vdash e_1 :: n_2, h_1, l_1, \vec{s}, S_1$ and $E_1 \vdash e_1 \rightarrow E_2 \vdash e_2$, then $n_1 \vdash e_2 :: n_2, h_2, l_2, \vec{t}, S_2$ s.t. $h_2 \leq h_1$, $l_2 \geq l_1$, $\vec{t} \leq \vec{s}$, and $S_2 \sqsubseteq S_1$.*

Lemma 7 (Subject reduction). *If $\Gamma \vdash P :: t$ and $\Gamma \vdash P \Longrightarrow \Gamma' \vdash P'$ then $\Gamma' \vdash P' :: t'$ where $t' \leq t$.*

The next lemma states a basic correctness property of our analysis, namely that for well-typed configurations, the actual resource consumption $|\Gamma|$ is over-approximated via the result $|t|$ of the analysis. We prove a slightly stronger statement namely that the actual resource consumption is approximated by the initial weight $|t|_1$.

Lemma 8. *If $\Gamma \vdash P :: t$, then $|\Gamma| \leq |t|_1$.*

The final result as corollary of subject reduction and the previous lemma: the statically calculated result is an over-approximation for all reachable configurations:

Theorem 1 (Correctness). *Given an initial configuration $\Gamma_0 \vdash p_0\langle e_0\rangle$ and $\Gamma_0 \vdash p_0\langle e_0\rangle :: t$ (with Γ_0 as empty global context). If $\Gamma_0 \vdash p_0\langle e_0\rangle \Longrightarrow^* \Gamma \vdash P$, then $|\Gamma| \leq |t|$.*

6 Conclusion

We have formalized a static, compositional effect-based analysis to estimate the resource bounds for a transactional model with nested and multi-threaded transactions. The analysis focuses on transactional memory systems where thread-local copies of memory resources (logs) caused by nested and multi-threaded transactions is our main concern. As usual, the challenge in achieving a sound static analysis lies in obtaining the following three goals at the same time: 1) compositionality, 2) precision, and 3) soundness. Without compositionality, the analysis is guaranteed not to scale for large programs, therefore not usable in practice. Without precision, compositionality and soundness can trivially be achieved by overly abstracting all details and ultimately rejecting all programs as potentially erroneous. Of course without soundness, it is pointless to formally analyze programs. Achieving all three goals in a satisfactory manner requires human ingenuity. In our setting the effect system can, in a *compositional* way, statically approximate the maximum number of logs that co-exist at run-time. This allows to infer the memory consumption of the transactional constructs in the program. To achieve a higher degree of precision in the approximation, it is important to take the underlying concurrency model and its synchronization into account. The main challenge is that the execution model has neither independent parallelism nor full sequentialization. To our knowledge, this is the first static analysis taking care of memory resource consumption for such a concurrency model. Abstracting away from the specifics of memory consumption and the concrete concurrent calculus, the effect system presented here can be seen as a careful, compositional account of a parallel model based on join-synchronization. It is promising to use our compositional techniques as explored here also to achieve different program analyses in a similar manner for programs based on fork/join parallelism. We expect that adapting our techniques to a model with *explicit* join synchronization, as e.g., in Java, leads to a simplification, as the synchronization is syntactically represented in the program code.

Related work Estimating memory, or more generally, resource usage has been studied, in various other settings. To specify upper bounds for the memory usage of dynamic, recursive data types, the notion of sized types have been introduced in [8]. Their system, a type and effect system as well, certifies a time limit for functional (and single-threaded) programs, relying on annotations by the programmer specifying time limits for each individual function. Hofmann and Jost [6] use a linear type system to compute linear bounds on heap space for a first-order functional language. One significant contribution of this work is the inference mechanism through linear programming techniques. Extensions from linear to polynomial resource bounds are presented in [5] and [4]. [15] deals with a first-order, call-by-value, garbage-collected functional language. Their approach is based on program analysis and model checking and not type-based. For imperative and object-oriented languages Wei-Ngan Chin et al. [2] treat explicit memory management in a core object-oriented language. Programmers have to annotate the memory usage and size relations for methods as well as explicit de-allocation. In [7], Hofmann and Jost combine amortized analysis, linear programming and functional programming to calculate the heap space bound as a function of input for an object oriented language. In [1] the authors present an algorithm to statically compute memory consumption of a method as a non-linear function of the method's parameters. The bounds are not precise. The main difference of our work in comparison to the above related ones is in that we are dealing not only with a multi-threaded analysis —many of the cited works are restricted to sequential languages— but also the complex and implicit synchronization structure entailed by the transactional model. The work in [14], as here, provides resource estimations in a concurrent (component-based) setting. The concurrency model in that work, however, is considerably simpler, as sequential and parallel composition are *explicit* constructs in the investigated calculus. Simpler is also the treatment in [16], which presents an analysis which is which does not treat parallel composition in a compositional manner, i.e., the compositional treatment is single-threaded. As a consequence, in that work, the effects do not capture the tree-like join-synchronization as here, at the expense of compositionality for parallel composition.

Current and future work We formalized the calculus and the type system in the Coq theorem prover (and using the OTT semantical framework [12]) and are currently working on a formalization of the correctness proof with the longer-term goal to use Coq's program extraction to obtain a formally correct implementation of the effect type system. Besides that, we plan to refine the effect system by deriving more detailed information about the logs (e.g. memory cells per log, or number of variables per log and so on) to infer memory consumption more precisely (which is an orthogonal problem, as mentioned). Furthermore, a challenging step is to automatically *infer* interface information concerning the resource consumption for method declarations.

References

1. V. Braberman, D. Garbervetsky, and S. Yovine. A static analysis for synthesizing parametric specifications of dynamic memory consumption. *Journal of Object Technology*, 5(5), 2006.
2. W.-N. Chin, H. H. Nguyen, S. Qin, and M. C. Rinard. Memory usage verification for OO programs. In *Proceedings of SAS '05*, volume 3672 of *LNCS*. Springer, 2005.
3. T. Harris, J. R. Larus, and R. Rawja. *Transactional Memory*. Morgan & Claypool, second edition, 2010.
4. J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate amortized resource analysis. In *Proceedings of POPL '11*. ACM, Jan. 2011.
5. J. Hoffmann and M. Hofmann. Amortized resource analysis with polynomial potential. a static inference of polynomial bounds for functional programs. In *Proceedings of ESOP 2010*, volume 6012 of *LNCS*. Springer, 2010.
6. M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of POPL '03*. ACM, Jan. 2003.
7. M. Hofmann and S. Jost. Type-based amortised heap-space analysis (for an object-oriented language). In *Proceedings of ESOP 2006*, volume 3924 of *LNCS*. Springer, 2006.
8. J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *Proceedings of POPL '96*. ACM, Jan. 1996.
9. S. Jagannathan, J. Vitek, A. Welc, and A. Hosking. A transactional object calculus. *Science of Computer Programming*, 57(2), Aug. 2005.
10. T. Mai Thuong Tran and M. Steffen. Safe commits for Transactional Featherweight Java. In *Proceedings of iFM 2010*, volume 6396 of *LNCS*. Springer, Oct. 2010.
11. T. Mai Thuong Tran, M. Steffen, and H. Truong. Estimating resource bounds for software transactions. Technical report 414, University of Oslo, Dept. of Informatics, Dec. 2011.
12. P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strniša. Ott: Effective tool support for the working semanticist. *Journal of Functional Programming*, 20(1), 2010.
13. N. Shavit and D. Toitu. Software transactional memory. In *22nd POPL*. ACM, Jan. 1995.
14. H. Truong and M. Bezem. Finding resource bounds in the presence of explicit deallocation. In *ICTAC'05*, volume 3722 of *LNCS*. Springer, 2005.
15. L. Unnikrishnan, S. D. Stoller, and Y. A. Liu. Optimized live heap bound analysis. In *Proceedings of VMCAI 2003*, volume 2575 of *LNCS*. Springer, 2003.
16. T. V. Xuan, H. T. Anh, T. Mai Thuong Tran, and M. Steffen. A type system for finding upper resource bounds of multi-threaded programs with nested transactions. In *ACM Proceedings of the 3rd ACM International Symposium on Information and Communication Technology SoICT*. ACM, 2012.