UNIVERSITY OF OSLO
Department of Informatics

Estimating Resource Bounds for Software Transactions

Research Report No. 414

Thi Mai Thuong Tran, Martin Steffen, and Hoang Truong

ISBN 82-7368-376-1
ISSN 0806-3036

December 2011

Estimating Resource Bounds for Software Transactions

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 and lower bounds on the memory resource consumption in a transactional calculus. The calculus is a concurrent variant of Featherweight Java extended by transactional constructs.
The model supports nested and concurrent transactions. The analysis is compositional and takes into account implicit join synchronizations that arise when more than one thread perfom a join-synchronization when jointly committing 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). We show the soundness of the analysis.

1 Introduction

Software Transactional Memory (STM) has recently been introduced to concurrent programming languages as an alternative for locked-based synchronization. STM enables an optimistic form of synchronization for shared memory. One of the advanced transactional calculi is Transactional Featherweight Java (TFJ) [14], a transactional object calculus which supports *nested* and *multi-threaded* transactions. Multi-threaded transactions mean 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 read and writes on its own local copy of the memory and e.g., a local log is used to record these operations to allow validation or potentially rollbacks at commit time. Maintaining the logs is 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 memory (i.e., logs) that may co-exist at the same time (in parallel threads) at a given point. 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 threads running in parallel, because of the nesting of transactions. A further complication when estimating the resource consumption is that parallel threads do not run independently; instead, executing a commit in a transaction may lead to a form of 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. The analysis is compositional, i.e., syntax-directed. The language features non-lexical starting and ending a transaction, concurrency, choice and sequencing. The analysis is *multi-threaded* in the sense that, due to synchronization, it does not analyze each thread in isolation, but needs to take their interaction into account. This complicates the design of the effect system considerably, as the synchronization is implicit in the use of commit-statements and connected to the nesting structure of the transactions. To our knowledge, there is no work on taking care of issues concerning memory/resources consumption are used in such programs.

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 property is sketched in Section 5. We conclude in Section 6 with related and future work.

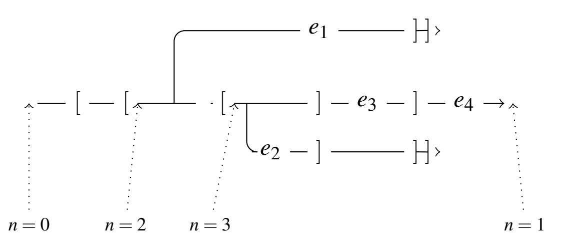
2 Compositional analysis of join synchronization

In this section, we sketch the concurrency and transaction model of the used calculus and the consequences for the analysis of the memory resource consumption. The presentation is informal and by way of examples; the syntax, semantics, and the analysis are presented more formally later.

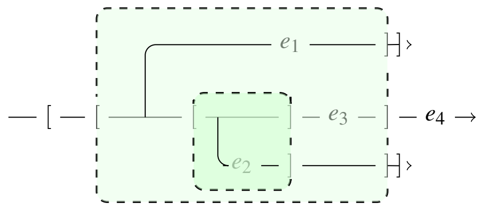
Example 1 (Joining commits). Consider the following (contrived) code snippet.

```
onacid;                                   // thread 0
  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`-statements expresses the start of a transaction and `commit` the end. Hence, the thread spawned first starts its execution at a nesting depth of 2 and the second one at depth 3. See also Fig. 1a. In the figure (and in the following) we often write [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, which makes the transaction multi-threaded. Further note that thread 1 uses two commits (after e_1) to close those transactions. Important is that parent and child thread(s) commit a shared enclosing transaction at the same time, i.e., executing a commit may lead to a 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 mark the joining commits. If the child thread, say in e_1, starts its own transactions (nested inside the surrounding ones), e.g., if $e_1 = [\ ;\ [\ ;\ [\ ;\ ]\ ;\ ]\ ;\ ]$, then these three commits are no joining commits. □



(a)



(b)

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

Our goal is to obtain 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. As a measure for the resource consumption at a given point, we take the *number* of logs co-existing at the same time. This ignores that, clearly, different logs may have different memory needs (e.g., accessing more variables in a transactional manner). Abstracting away from this difference, we concentrate on the synchronization and nesting structure underlying the concurrency model with nested and multi-threaded transactions.

Example 2 (Resource consumption). For the code of Example 1, the resource consumption can be seen as follows. Assuming that e_1 opens and closes three transactions, e_2 four, e_3 five, and e_4 six, the resource consumption after spawning the thread for e_2 and before the subsequent `commit` is $15 = 3+5+7$ in the worst case: the main thread executes inside 3 transactions, thread 1 inside 5 (three from e_1 plus two "inherited" from the parent), and thread 2 contributes 7. At the point when the main thread executes e_3, i.e., after its first commit, the resource consumption in the worst case is $12 = 5+5+2$. 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 (cf. Fig. 1 again). □

To be efficient, i.e., to be scalable and usable in practice, the analysis must be *compositional*. This syntax-directedness is common for type/effect-based analyses. In our setting, the analysis needs to cope with parallelism and synchronization. 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 $e_1 \parallel e_2$ can approximated by the *sum* of the respective contributions. The challenges we are facing in our model are:

Multi-threaded analysis: due to joining commits, threads running in parallel 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 achieve precision, the analysis must be aware of which program parts can run in parallel and which not. Assuming independent parallelism would allow to analyze each thread in isolation. Such a single-threaded analysis would still yield a sound over-approximation, but be too imprecise.

Implicit synchronization: Compositional analysis is rendered intricate by the fact that the synchronization is *not* explicitly represented syntactically. In particular, there is no clean syntactic separation between sequential composition and parallel composition. For instance writing $(e_1 \parallel e_2); e_3$ would make the join synchronization that sequentially separates the $e_1 \parallel e_2$ from e_3 explicit and would make a compositional analysis straightforward. 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. Assume that we split the code of Example 1 after the first spawn, i.e., at the semicolon at the end of line 3 and that we analyse the two parts, say e_l and e_r independently. Writing m for the effect that over-approximates the memory consumption, we need to obtain a rule for sequential composition resembling the following:

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

For compositionality, the "interface" information captured in the effects must be rich enough such that m in the conclusion can be calculated from m_1 and m_2. In particular, 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 Figure 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 contribution and the parallel contribution —the contribution of the spawned thread— of e_l. Furthermore, the parallel part of m_1 is partly synchronized with e_r by joining commits, and hence the effects must contain information about the corresponding synchronization points. Ultimately, the judgements of the effect system

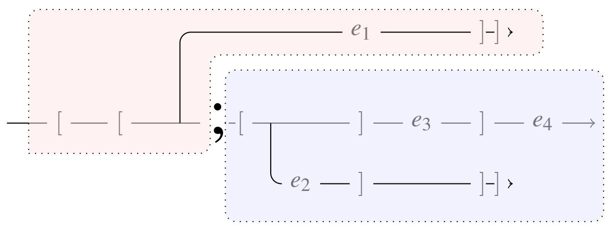


Fig. 2: Compositional analysis (sequential composition)

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	$::=$	$C(\vec{f} : \vec{T})\{\mathsf{this}.\vec{f} := \vec{f}\}$	contructors
M	$::=$	$m(\vec{x}{:}\vec{T})\{e\} : T$	methods
e	$::=$	$v \mid v.f \mid v.f := v \mid \mathtt{if}\ v\ \mathtt{then}\ e\ \mathtt{else}\ e$	
	$\mid$	$\mathtt{let}\ x{:}T = e\ \mathtt{in}\ e \mid v.m(\vec{v})$	expressions
	$\mid$	$\mathtt{new}\ C(\vec{v}) \mid \mathtt{spawn}\ e \mid \mathtt{onacid} \mid \mathtt{commit}$	
v	$::=$	$r \mid x \mid \mathtt{null}$	values

Table 1: Abstract syntax

will use a six-tuple of information that will allow a compositional analysis of sequential composition as well as parallel composition (plus the other constructs of the language). A central part of the effects is a tree-representation of the future resource consumption and joining commits, which we call jc-trees. □

3 A transactional calculus

We start by presenting the syntax and semantics of TFJ. It is, with some adaptations, taken from [14] and a variant of Featherweight Java (FJ) [13] extended with *transactions* and a construct for thread creation. The main adaptations, as in [15], 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 fairly standard and omitted here.

3.1 Syntax

FJ is a core language originally introduced to study typing issues related to Java, such as inheritance, subtype polymorphism. A number of extensions have been developed for other language features, so FJ is today a generic name for Java-related core calculi. Following [14] and in contrast to the original FJ proposal, we ignore inheritance, subtyping, and type casts, as these features are orthogonal to the issues at hand, but include imperative features such as destructive field updates, further concurrency and transactions.

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 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. Here the vector notation is used analogously to the vector $\vec{f}$ which presents a list of fields. The vector $\vec{T}$ represents a sequence of types, $\vec{x}$ stands for a sequence of variables. When writing $\vec{x}{:}\vec{T}$ we assume that the length of $\vec{x}$ corresponds to the length of $\vec{T}$, and we refer by $x_i : T_i$ to the i'th pair of variable and type. For brevity, we do not make explicit or formalize such assumptions, when they are clear from the context.

In the syntax, v stands for values, i.e., expressions that can no longer be evaluated. In the core calculus, we leave unspecified standard values like booleans, integers, $\ldots$, so 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 `new` $C(\vec{v})$. The next two expressions deal with the basic, sequential control structures: `if` v `then` e_1 `else` e_2 represents conditions, and the let-construct `let` $x{:}T = e_1$ `in` e_2 represents sequential composition: first e_1 is evaluated, and afterwards e_2, where the eventual value of e_1 is bound to the local variable x. Consequently, standard sequential composition $e_1;e_2$ is syntactic sugar for `let` $x{:}T = e_1$ `in` e_2 where the variable x does not occur free in e_2. The let-construct, as usual, binds x in e_2. We write $fv(e)$ for the free variables of e, defined in the standard way. The language is multi-threaded: `spawn` e starts a new thread of activity which evaluates e in parallel with the spawning thread. Specific for TFJ are the two constructs `onacid` and `commit`, two dual operations dealing with transactions. The expression `onacid` starts a new transaction and executing `commit` successfully terminates a transaction by committing its effect, otherwise the transaction will be rolled back or aborted. In case of multiple threads inside the same transaction, all threads perform a join synchronization when committing the transaction.

A note on the form of expressions and the use of values may be in order. The syntax is restricted concerning where to use general expressions e. E.g., Table 1 does not allow field updates $e_1.f := e_2$, where the object whose field is being updated and the value used in the right-hand side are represented by general expressions that need to be evaluated first. It would be straightforward to relax the abstract syntax that way and indeed the proposal of TFJ from [14] allows such more general expressions. We have chosen this presentation, as it slightly simplifies the operational semantics and the (presentation of the) type and effect system later: [14] specifies the operational semantics using so-called evaluation contexts, which fixes the order of evaluation in such more complex expressions. With that slightly restricted representation, we can get away with a semantics without evaluation contexts, using simple rewriting rules (and the let-syntax). Of course, this is not a real restriction in expressivity. E.g., the mentioned expression $e_1.f := e_2$ can easily and be expressed by `let` $x_1 = e_1$ `in` (`let` $x_2 = e_2$ `in` $x_1.f := x_2$), making the evaluation order explicit. The transformation from the general syntax to the one of Table 1 is standard.

3.2 Semantics

The operational semantics of TFJ is separated in two different levels: a local and a global semantics. The local semantics is given in Table 2, dealing with the evaluation of *one* single *expression/thread* and reduce configurations of the form $E \vdash e$. Local transitions are thus of the form

$$E \vdash e \rightarrow E' \vdash e' \ , \tag{1}$$

where e is one expression and E a *local environment.* At the local level, 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* $l_1{:}\rho_1, \dots l_k{:}\rho_k$*, 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 commiting 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 keeps track of changes of the threads actions concerning transaction l_i. The exact structure of such environments and the logs have no influence on our static analysis, and indeed, the environments may be realized in different ways (e.g., [14] gives two different flavors, a "pessimistic", lock-based one and an "optimistic" one). Relevant for our effect system is only a number of *abstract properties*.

Definition 2. *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* $\Gamma = p_1{:}E_1, \dots, p_k{:}E_k$*, then* $\Gamma' = p_1{:}E'_1, \dots, p_k{:}E'_k$ *with* $|E_i| = |E'_i|$ *(for all i).*
2. *The function spawn satisfies the following condition: Assume* $\Gamma = t : E, \Gamma''$ *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 a* $\Gamma = p_1{:}E_1, \dots, p_i{:}E_i, \dots, p_k{:}E_k$ *and for a fresh l, then* $\Gamma' = p_1{:}E_1, \dots, p_i{:}E'_i, \dots, p_k{:}E_k$*, 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* $intranse(l,\Gamma) = \vec{p}$*, then*
   (a) $p \in \vec{p}$ *and*
   (b) *for all* $p_i \in \vec{p}$ *we have* $\Gamma = \dots, p_i : (E'_i, l{:}\rho_i), \dots$*.*
   (c) *for all threads* p' *with* $p' \notin \vec{p}$ *and where* $\Gamma = \dots, t'{:}(E', l'{:}\rho'), \dots$*, we have* $l' \neq l$*.*
5. *The function commit satisfies the following condition: if* $commit(\vec{p}, \vec{E}, \Gamma) = \Gamma'$ *for a* $\Gamma = \Gamma'', p{:}(E, l{:}\rho)$ *and for a* $\vec{p} = intranse(l, \Gamma)$ *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* $|E'_i| = |E_i| - 1$*.*

The operational rules are formulated exploiting the let-construct/sequential composition, and the restricted form of (abstract) syntax. The syntax for the conditional construct from Table 1, e.g., insists that the boolean condition is already evaluated (i.e., either a boolean value or value/reference to such a value), and the R-COND-rules apply when the previous evaluation has yielded already true, resp. false.

We use the let-construct to unify sequential composition, local variables, and handing over of values in a sequential composition, and rule R-LET basically expresses associativity of the sequential composition, i.e., ignoring the local variable declarations, it corresponds to a step from $(e_1;e);e'$ to $e_1;(e;e')$. Note further that the left-hand side for all local rules (and later the global ones) insists that the top-level construct is a let-construct. That is assured during run-time inductively by the form of the initial thread and the restiction on our syntax.

The first two rules deal with the basic evaluation based on substitution and specifying a left-to-right evaluation (cf.R-RED and R-LET). The two R-COND-rules deal with conditionals in an obvious way. Unlike the first four rules, the remaining ones do access the heap. Thus, in the premises of these rules, 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 an entry for a new reference on the heap). Note that also the *read*-function used in the rules 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 the transaction fails. The premises assume the class table is given implicitly where $\mathit{fields}(C)$ looks up fields of class C and $\mathit{mbody}(m,C)$ looks up the method m of class C. So, field look-up in R-FIELD works as follows: consulting the local environment E, the *read*-function looks up the object referenced by r; the object is $C(\vec{u})$, i.e., it's an instance of class C, and its fields carry the values $\vec{u}$. The (run-time) type C of the object is further used to determine the fields $\vec{f}$, using the object referenced by r, where *fields* finds the fields of the object referenced by r, and the step replaces the field access $r.f_i$ by the corresponding value u_i. Field update in rule R-UPD works similarly, again using *read* to look up the objects, and additionally using *write* to write the value r' back into the local environment, thereby changing E' to E'' (again, the exact details of the function are left abstract).

The function *mbody* in the rule R-CALL for method invocation gives back the method's formal parameters $\vec{x}$ and the method body, and invocation involves substituting $\vec{x}$ by the actual parameters $\vec{r}$ and substituting `this` by the object's identity r. Rule R-NEW, finally, takes care of object creation, using a fresh object identity r to refer to the new instance $C(\vec{\texttt{null}})$, which has all fields initialized to `null`. The function *extend* in that rule extends E by binding the fresh reference r to the newly created instance.

The five rules of the *global* semantics are given in Table 3. The semantics works on configurations of the following form:

$$\Gamma \vdash P\,, \tag{2}$$

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 that we describe 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 $t{:}E$ bindings where p is the name of a thread and E is its corresponding local environment.

Definition 3 (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:

$$\Gamma \vdash P \Longrightarrow \Gamma' \vdash P' \quad \text{or} \quad \Gamma \vdash P \Longrightarrow \mathit{error}\ . \tag{3}$$

Also the global steps make use of a number of functions accessing and changing the (this time global) environment. As before, those functions are left abstract (cf. Definition 2). 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. Rule G-SPAWN deals with starting a thread. The next three rules treat the two central commands of the calculus, those dealing directly 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. the failed attempt to commit a transaction. In G-COMM, the label of the transaction l to be committed is found (right-most) in the local context E. Furthermore, the function $\mathit{intranse}(l,\Gamma)$ 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.

$$E \vdash \mathtt{let}\ x : T = v\ \mathtt{in}\ e \rightarrow E \vdash e[v/x] \qquad \text{R-RED}$$

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

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

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

$$\frac{\mathit{read}(r,E) = E', C(\vec{u}) \quad \mathit{fields}(C) = \vec{f}}{E \vdash \mathtt{let}\ x{:}T = r.f_i\ \mathtt{in}\ e \rightarrow E' \vdash \mathtt{let}\ x{:}T = u_i\ \mathtt{in}\ e}\ \text{R-LOOKUP} \qquad \frac{\mathit{read}(r,E) = E', C(\vec{r}) \quad \mathit{write}(r \mapsto C(\vec{r}) \downarrow_i^{r'}, E') = E''}{E \vdash \mathtt{let}\ x{:}T = r.f_i := r'\ \mathtt{in}\ e \rightarrow E'' \vdash \mathtt{let}\ x{:}T = r'\ \mathtt{in}\ e}\ \text{R-UPD}$$

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

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

Table 2: Semantics (local)

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

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

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

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

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

Table 3: Semantics (global)

Definition 4. *Let TrName be the type of transaction labels. Given a local environment E, the function* $l : (\mathit{LEnv} \to \mathit{List\ of\ TrName})$ *is defined inductively as follows:* $l(\varepsilon) = \varepsilon$*, and* $l(l{:}\_, E) = l, l(E)$*. Overloading the definition, we lift the function straightforwardly to global environments (with type* $l : \mathit{TName} \times \mathit{GEnv} \to \mathit{List\ of\ TrName}$*), s.t.* $l(p, (p{:}E), \Gamma) = l(E)$*.*

The first definition, extracting the list of transaction labels from a local environment E is a straightforward projection, simply extracting the sequence of transaction labels. As for the *order* of the transactions: As said, the most recent, the innermost transaction label is to the right. Given a transaction, the following function determines the threads for which the given transaction is (properly) "nested" in a global environment, i.e., those threads which execute *inside* the given transaction but where the transaction is *not the current, directly enclosing* transaction.

Definition 5 (Nesting). *Given a global environment, the function* $\mathit{nested} : \mathit{TrName} \times \mathit{GEnv} \to \mathit{List\ of\ TName}$ *returns the list of all threads nested inside a given transaction.*

4 Type and effect system

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

$$n_1 \vdash e :: n_2, h, l, \vec{t}, S \tag{4}$$

The elements 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 represent the maximum resp., the minimum value of the balance *during* the execution of e (the "highest" and the "lowest" balance during execution). The numbers so far describe the balances of the thread executing e. During the execution of e, however, new child threads may be created via the spawn-expression and the remaining elements $\vec{t}$ and S take their contribution into account. 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 potential *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 $e_r = [\texttt{spawn}\, e_2]e_3]e_4$ has one joining commit and the jc-sequence $\vec{t} = 10, 7$.

The last component 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 itself. That reflects the fact that in the operational semantics, a child thread is contained in the surrounding transactions and furthermore, the transactional log of the parent is copied into the newly spawned thread.

The derivation rules locally for expressions are shown in Table 4. The rules for variable, the null reference, for field look-up and field update, and for object instantiation are trivial, as they neither affect the balance nor is any other thread involved. Initiating a transaction (cf. rule T-ONACID) increases the balance by one and accordingly the highest balance and the total sum, whereas the minimum value stays constant. 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 6 (Parallel weight). *Let S be a multi-set of the form $\{(p_1,c_1),\ldots,(p_k,c_k)\}$ where the p_i and c_i are natural numbers, and l be a natural number. Then we define the following functions:*

$$\begin{array}{cc} par(S,l) = \{(p,c) \in S \mid p \le l\} & seq(S,l) = \{(p,c) \in S \mid p > l\}\,. \\ \lfloor S \rfloor_l = \{(l,0),(l,0),\ldots\} & S \downarrow_l = par(S,l) \cup \lfloor seq(S,l) \rfloor_l \end{array} \tag{5}$$

Furthermore, the overall parallel weight of S is defined as $|S| = \sum_i (p_i + c_i)$.

Definition 7 (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:*

$$\vec{s} \oplus_p \vec{t} = s_0, \ldots, s_k \vee t_0 \ldots \vee t_p, t_{p+1}, \ldots, t_m \tag{6}$$

Given a parallel weight S and a $n \geq m \geq 0$, *then* $\otimes_n$ *is defined as*

$$S \otimes_n \vec{t} = t'_0, t'_1, \ldots, t'_m \tag{7}$$

where $t'_0 = t_0 + |S|$, $t'_1 = t_1 + |S_1 \downarrow_{n-1}|$, $\ldots t'_m = t_m + |S_1 \downarrow_{n-m}|$.

To determine the spawned weight S in T-LET, the spawned weight S_1 of e_1 needs to be split into two halves (cf. Definition 6).

1. The part $par(S_1, l_2)$ of S_1 *not* affected 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). This distinction is based on comparing the inherited nesting depth p_i with the minimal balance l_2 of e_2. The parallel weight $par(S_1, l_2)$ 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. The reason is that 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 $S_1 \downarrow_{l_2}$ contains the resource 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 $S_1 \downarrow_{l_2} \cup S_2$ as overall parallel weight. The correctness of the calculation in T-LET depends on the restriction on the language that once a spawned thread commits a transaction inherited from its parent thread, it will not open another transaction.

Now to the compositional calculation of the jc-sequence $\vec{u}$ (cf. Definition 7): the calculation takes care of two phenomena: 1) The parallel weight S_1 at the end of e_1 has to be taken into account, since that may increase the resource consumption of the jc-sequence $\vec{t}$. This is formalized by the $\otimes_{\_}$ operation of Definition 7. 2) 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 $\vec{s} \oplus_{n_2 - l_1} \vec{t}$ "merges" an appropriate number of elements from $\vec{t}$ (using $\vee$) depending on how many joining commits disappear in the

$$\frac{}{n \vdash x :: n,n,n,[n],\emptyset}\ \textsc{T-Var} \qquad \frac{}{n \vdash \texttt{null} :: n,n,n,[n],\emptyset}\ \textsc{T-Null} \qquad \frac{}{n \vdash v.f :: n,n,n,[n],\emptyset}\ \textsc{T-Lookup}$$

$$\frac{}{n \vdash v_1.f_i := v_2 :: n,n,n,[n],\emptyset}\ \textsc{T-Upd} \qquad \frac{}{n \vdash \texttt{new}\, C :: n,n,n,[n],\emptyset}\ \textsc{T-New}$$

$$\frac{}{n \vdash \texttt{onacid} :: n+1,n+1,n,[n+1],\emptyset}\ \textsc{T-Onacid} \qquad \frac{n \geq 1}{n \vdash \texttt{commit} :: n-1,n,n-1,[n;n-1],\emptyset}\ \textsc{T-Commit}$$

$$\frac{\begin{array}{c} n_1 \vdash e_1 :: n_2,h_1,l_1,\vec{s},S_1 \qquad n_2 \vdash e_2 :: n_3,h_2,l_2,\vec{t},S_2 \\ h = h_1 \vee h_2 \qquad l = l_1 \wedge l_2 \qquad \vec{s} = s_1,\ldots,s_k \qquad \vec{t} = t_1,\ldots,t_m \qquad k,m \geq 1 \qquad p = n_2 - l_1 \\ t'_1 = t_1 + |S_1| \qquad t'_2 = t_2 + |S_1 \downarrow_{n_2-1}| \qquad \ldots \qquad t'_m = t_m + |S_1 \downarrow_{n_2-(m-1)}| \\ S = S_1 \downarrow_{l_2} \cup S_2 \qquad \vec{u} = \vec{s} \oplus_p (S_1 \ogreaterthan_{n_2} \vec{t}) = s_1,\ldots,s_{k-1},s_k \vee t'_1 \vee \ldots \vee t'_p,t'_{p+1},\ldots,t'_m \end{array}}{n_1 \vdash \texttt{let}\, x{:}T = e_1 \,\texttt{in}\, e_2 :: n_3,h,l,\vec{u},S}\ \textsc{T-Let}$$

$$\frac{n \vdash e_1 :: n',h_1,l_1,\vec{s},S \qquad n \vdash e_2 :: n',h_2,l_2,\vec{t},S}{n \vdash \texttt{if}\ v\ \texttt{then}\, e_1\, \texttt{else}\, e_2 :: n',h_1 \vee h_2,l_1 \wedge l_2,\vec{s} \vee \vec{t},S}\ \textsc{T-Cond}$$

$$\frac{n_1 \vdash e :: 0,h,l,\vec{s},S}{n_1 \vdash \texttt{spawn}\, e :: n_1,n_1,n_1,[n_1+s_0],S \cup \{(n_1,h-n_1)\}}\ \textsc{T-Spawn} \qquad \frac{mtype(C,m) :: n_1 \to n_2,h,l,\vec{t},S}{n_1 \vdash v.m(\vec{v}) :: n_2,h,l,\vec{t},S}\ \textsc{T-Call}$$

Table 4: Effect system

composition. This number p is given by $n_2 - l_1$. So in rule T-LET, the overall $\vec{u}$ is given as $\vec{s} \oplus_p (S_1 \ogreaterthan_{n_2} \vec{t})$.

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 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. For the parallel contributions of e_1 and e_2, they are *not* necessarily separated by the syntax: threads spawned in e_1 can run in parallel with e_2. In this case, the contributions of e_1 and e_2 need to be treated *additively* as they may occur at the same time in the worst case. If potential parallelism were the *only* relationship between the spawned threads of e_1 and the subsequent e_2, the situation would still be comparatively simple. In the model of nested and concurrent transactions, however, threads do not run uncoordinated in parallel: A commit executed by a thread spawned inside a transaction *synchronizes* via a join with the corresponding commit of the spawning thread. This may lead to a sequentiality constraint between the effects of e_1 and e_2 such that the overall effect is not calculated additively, by taking the corresponding least upper bound. This kind of sequentiality concerning the effects of the spawned threads of e_1 and the effects of e_2 are not reflected syntactically in the sequential composition `let` $x = e_1$ `in` e_2, which makes the compositional treatment of the sequential composition complicated.

The treatment of conditionals in rule T-COND is comparatively simple: the maximal balance is given as least upper bound and dually the minimal balance as greatest lower bound of the corresponding values of the two branches. Besides that, both arms of the conditional must agree wrt. their post-balance and their parallel weight S. The definition of least upper bound of $\vec{s}$ and $\vec{t}$ of vectors of the same length, written $\vec{s} \vee \vec{t}$, is defined pointwise. Similarly we write $\vec{s} \leq \vec{t}$ (for vectors of the same length, if the $\leq$-relation holds point-wise. When spawning a new thread to execute an expression 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 is determined taking 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 the thread executing e, 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 directly given by the interface information of method m in class C, which is looked up using *mtype*.

Example 4. The example illustrates our type and effect system by giving the derivation for Example 1 in Section 2 as follows (focusing on the $\vec{t}$- and S-part, only):

$$\dfrac{\dfrac{\vdots}{0 \vdash [\ [\ ;\texttt{spawn}\,(e_1\ ]\ ]\ ) :: [7], \{(2,3)\}} \qquad \dfrac{\vdots}{2 \vdash [\ ;(\texttt{spawn}\,(e_2\ ]\ ]\ ]\ );\ ]\ ;e_3\ ]\ ;e_4 :: [10,8], \{(1,0)\}}}{0 \vdash [\ [\ ;\texttt{spawn}\,(e_1;\ ]\ ]\ )\ ;\ [\ ;(\texttt{spawn}\,(e_2;\ ]\ ]\ ]\ );\ ]\ ;e_3\ ]\ ;e_4 :: t, \{(1,0),(1,0)\}}$$

The overall resource consumption then is $t = 15 = 7 \vee (10 + |\{(2,3)\}|) \vee (8 + |\{(1,0)\}|)$.

Global effects, parallel composition, and joining commit trees The rest of the section is concerned 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 are 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:

$$\Gamma \vdash P :: t \tag{8}$$

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.*

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

Table 5: Effect system

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.*

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

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

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

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

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 will be 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 $|\vec{s}| = |\mathit{lift}(E,\vec{s})|$.

5 Correctness

This section establishes the soundness of the analysis, i.e., that the estimation given back by the type and effect system overapproximates the actual potential resource consumption for all reachable configurations. Remember that the resource consumption is measured in terms of numbers of logs co-existing simultaneously (cf. Definition 12). We start by defining the actual resource consumption of a program:

Definition 12 (Resource consumption). *The weight of a local environement 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, which are later needed in the inductive proof of subject reduction.

Lemma 1. $(S_1 \cup S_2) \ogreaterthan_n \vec{t} = S_1 \ogreaterthan_n (S_2 \ogreaterthan_n \vec{t})$.

Proof. Straightforward. $\square$

Lemma 2. *Let S be a parallel weight and n_1 and n_2 two non-negative numbers.*

1. $S \downarrow_{n_1} \downarrow_{n_2} = S \downarrow_{n_2} \downarrow_{n_1}$.
2. *If $n_2 \leq n_1$, then $S \downarrow_{n_1} \downarrow_{n_2} = S \downarrow_{n_2}$.*

Proof. By straightforward calculation. $\square$

The next two lemma shows that the way the resource consumption is calculated in the let-rule is associative, which is a crucial ingredient in subject reduction.

Lemma 3 (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*

$$f(f(S_1, l_2, S_2), l_3, S_3) = f(S_1, l_2 \wedge l_3, f(S_2, l_3, S_3)) \ .$$

Proof. By straightforward but slightly tedious calculation. $\square$

Lemma 4 (Associativity of $\oplus$ and $\ogreaterthan$). $\vec{s} \oplus_{p_1} (S_1 \ogreaterthan_{n_2} (\vec{t} \oplus_{p_2} (S_2 \ogreaterthan_{n_3} \vec{u}))) = (\vec{s} \oplus_{p_1} (S_1 \ogreaterthan_{n_2} \vec{t})) \oplus_{p_2} (S_2 \cup S_1 \downarrow_{l_2} \ogreaterthan_{n_3} \vec{u})$.

Proof. We are given $\vec{s} = s_0, \ldots, s_k$, $\vec{t} = t_0, \ldots, t_m$, and $\vec{u} = u_0, \ldots, u_q$. Further we set

$$\begin{aligned} l_1 &= n_1 - |s| + 1 = n_1 - k \\ l_2 &= n_2 - |t| + 1 = n_2 - m \\ l_3 &= n_3 - |u| + 1 = n_3 - q \\ p_1 &= n_2 - l_1 \\ p_2 &= n_3 - l_2 \end{aligned} \tag{9}$$

where the l_i, n_i and relation connecting them with the p_i reflect the use of those quantities in the T-LET type rule. We distinguish according to the relationship between the low points l_1, l_2, and l_3.

Case: $l_2 \leq l_1$ and $l_3 \leq l_2$
The assumption $l_2 \leq l_1$ implies with the equations (9) $p_1 \leq m$ and $l_3 \leq l_2$ implies $p_2 \leq q$. Expanding the definitions for the left-hand and the right-hand side of the equation of the lemma gives the following two chains of equations:

$$\begin{aligned} &\vec{s} \oplus_{p_1} (S_1 \ogreaterthan_{n_2} (\vec{t} \oplus_{p_2} (S_2 \ogreaterthan_{n_3} \vec{u}))) = \\ &\qquad \vec{s} \oplus_{p_1} (S_1 \ogreaterthan_{n_2} (\vec{t} \oplus_{p_2} (u_0 + |S_2|, u_1 + |S_2 \downarrow_{n_3 - 1}|, \ldots, u_q + |S_2 \downarrow_{n_3 - q}|))) = \\ &\qquad \vec{s} \oplus_{p_1} (S_1 \ogreaterthan_{n_2} (\vec{t} \oplus_{p_2} \vec{u}')) = \\ &\qquad \vec{s} \oplus_{p_1} (S_1 \ogreaterthan_{n_2} (t_0, t_1, \ldots, t_m \vee u'_0 \vee u'_1 \vee \ldots \vee u'_{p_2}, u'_{p_2+1}, \ldots, u'_q)) = \\ &\qquad \vec{s} \oplus_{p_1} (S_1 \ogreaterthan_{n_2} (t_0, t_1, \ldots, \tilde{t}_m, u'_{p_2+1}, \ldots, u'_q)) = \\ &\qquad \vec{s} \oplus_{p_1} (t_0 + |S_1|, t_1 + |S_1 \downarrow_{n_2 - 1}|, \ldots, \tilde{t}_m + |S_1 \downarrow_{n_2 - m}|, = \\ &\qquad\qquad u'_{p_2+1} + S_1 \downarrow_{n_2 - (m+1)}, \ldots, u'_q + S_1 \downarrow_{n_2 - (m+q-p_2)}) \\ &\qquad \vec{s} \oplus_{p_1} (t''_0, t''_1, \ldots, \tilde{t}''_m, u''_{p_2+1}, \ldots, u''_q) = \\ &\qquad \vec{s} \oplus_{p_1} (t''_0, t''_1, \ldots, t''_{p_1}, t''_{p_1+1}, \ldots, \tilde{t}''_m, u''_{p_2+1}, \ldots, u''_q) = \\ &\qquad s_0, \ldots, s_k \vee t''_0 \vee t''_1 \vee t''_{p_1}, t''_{p_1+1}, \ldots, \tilde{t}''_m, u''_{p_2+1}, \ldots, u''_q = \end{aligned} \tag{10}$$

and

$$\begin{array}{ll}
(\vec{s} \oplus_{p_1} (S_1 \oslash_{n_2} \vec{t})) \oplus_{p_2} (S_2 \cup S_1 \downarrow_{l_2} \oslash_{n_3} \vec{u}) \ = & \\
\quad (\vec{s} \oplus_{p_1} (t_0 + |S_1|, t_1 + |S_1 \downarrow_{n_2-1}|, \ldots, t_m + |S_1 \downarrow_{n_2-m}|)) \oplus_{p_2} (S_2 \cup S_1 \downarrow_{l_2} \oslash_{n_3} \vec{u}) & = \\
\quad (\vec{s} \oplus_{p_1} (t''_0, t''_1, \ldots, t''_m)) \oplus_{p_2} (S_2 \cup S_1 \downarrow_{l_2} \oslash_{n_3} \vec{u}) & = \\
\quad (s_0, \ldots, s_{k-1}, s_k \vee t''_0 \vee t''_1 \vee \ldots \vee t''_{p_1}, t''_{p+1}, \ldots, t''_m) \oplus_{p_2} (S_2 \cup S_1 \downarrow_{l_2} \oslash_{n_3} \vec{u}) & = \\
\quad (s_0, \ldots, s_{k-1}, s_k \vee t''_0 \vee t''_1 \vee \ldots \vee t''_{p_1}, t''_{p+1}, \ldots, t''_m) \oplus_{p_2} (S_1 \downarrow_{l_2} \oslash_{n_3} (S_2 \oslash_{n_3} \vec{u})) & = \\
\quad (s_0, \ldots, s_{k-1}, s_k \vee t''_0 \vee t''_1 \vee \ldots \vee t''_{p_1}, t''_{p+1}, \ldots, t''_m) \oplus_{p_2} (S_1 \downarrow_{l_2} \oslash_{n_3} \vec{u}') & \\
\quad (s_0, \ldots, s_{k-1}, s_k \vee t''_0 \vee t''_1 \vee \ldots \vee t''_{p_1}, t''_{p+1}, \ldots, t''_m) \oplus_{p_2} \vec{u}''' & = \\
\quad s_0, \ldots, s_{k-1}, s_k \vee t''_0 \vee t''_1 \vee \ldots \vee t''_{p_1}, t''_{p+1}, \ldots, t''_{m-1}, t''_m \vee u'''_0 \vee \ldots \vee u'''_{p_2}, u'''_{p_2+1}, \ldots, u'''_q & = \\
\quad s_0, \ldots, s_{k-1}, s_k \vee t''_0 \vee t''_1 \vee \ldots \vee t''_{p_1}, t''_{p+1}, \ldots, t''_{m-1}, t'''_m, u'''_{p_2+1}, \ldots, u'''_q &
\end{array} \tag{11}$$

In the calculation, we used the following abbreviations:

$$\begin{aligned}
\vec{u}' &= S_2 \oslash_{n_3} \vec{u} = (u_0 + |S_2|, u_1 + |S_2 \downarrow_{n_3-1}|, \ldots, u_q + |S_2 \downarrow_{n_3-q}|))) \\
\tilde{t}_m &= t_m \vee u'_0 \vee u'_1 \vee \ldots \vee u'_{p_2} \\
t''_0, t''_1, \ldots, t''_{m-1} &= (t_0 + |S_1|, t_1 + |S_1 \downarrow_{n_2-1}|, \ldots, t_{m-1} + |S_1 \downarrow_{n_2-(m-1)}|, \\
\tilde{t}''_m &= \tilde{t}_m + |S_1 \downarrow_{n_2-m}| \\
u''_{p_2+1} \ldots, u''_q &= u'_{p_2+1} + S_1 \downarrow_{n_2-(m+1)}, \ldots, u'_q + S_1 \downarrow_{n_2-(m+q-p_2)})
\end{aligned}$$

$$\begin{aligned}
t''_m &= t_m + |S_1 \downarrow_{n_2-m}| \\
\vec{u}''' &= S_1 \downarrow_{l_2} \oslash_{n_3} \vec{u}' \\
t'''_m &= t''_m \vee u'''_0 \vee \ldots \vee u'''_{p_2} \\
S'_1 &= S_1 \downarrow_{l_2} \\
S'_2 &= S_1 \downarrow_{l_2} \cup S_2
\end{aligned}$$

To see that (10) and (11) are equal, we need to establish the following two equation. The required equality $\tilde{t}''_m = t'''_m$ is shown as follows:

$$\begin{array}{lll}
\tilde{t}''_m & = \tilde{t}_m + |S_1 \downarrow_{n_2-m}| & \\
& = (t_m \vee u'_0 \vee u'_1 \vee \ldots \vee u'_{p_2}) + |S_1 \downarrow_{n_2-m}| & \text{(distributivity)} \\
& = \ ((t_m + |S_1 \downarrow_{n_2-m}|) \vee (u'_0 + |S_1 \downarrow_{n_2-m}|) \vee & (l_2 = n_2 - m) \\
& \qquad\qquad (u'_1 + |S_1 \downarrow_{n_2-m}|) \vee \ldots \vee (u'_{p_2} + |S_1 \downarrow_{n_2-m}|) & \\
& = \ (t_m + |S_1 \downarrow_{l_2}|) \vee (u'_0 + |S_1 \downarrow_{l_2}|) \vee (u'_1 + |S_1 \downarrow_{l_2}|) \vee & \text{(Lemma 2)} \\
& \qquad\qquad \ldots \vee (u'_{p_2} + |S_1 \downarrow_{l_2}|) & \\
& = \ (t_m + |S_1 \downarrow_{l_2}|) \vee (u'_0 + |S_1 \downarrow_{l_2}|) \vee (u'_1 + |S_1 \downarrow_{l_2} \downarrow_{n_3-1}|) \vee & \\
& \qquad\qquad \ldots \vee (u'_{p_2} + |S_1 \downarrow_{l_2} \downarrow_{n_3-p_2}|) & \\
& = (t_m + |S_1 \downarrow_{n_2-m}|) \vee u'''_0 \vee \ldots \vee u'''_{p_2} & \\
& = t''_m \vee u'''_0 \vee \ldots \vee u'''_{p_2} & \\
& = t'''_m &
\end{array}$$

For the application of Lemma 2, observe that for all indices $n_3 - j$, we have $n_3 - j \geq l_2$. For the required equality $u''_{p_2+1}, \ldots, u''_q = u'''_{p_2+1}, \ldots, u'''_q$, we argue as follows:

$$\begin{aligned}
u''_{p_2+1},\dots,u''_q &= u'_{p_2+1} + |S_1 \downarrow_{n_2-(m+1)}|,\dots,u'_q + |S_1 \downarrow_{n_2-(m+q-p_2)}| && \text{(by definition)}\\
&= u'_{p_2+1} + |S_1 \downarrow_{l_2} \downarrow_{n_2-(m+1)}|,\dots,u'_q + |S_1 \downarrow_{l_2} \downarrow_{n_2-(m+q-p_2)}| && \text{(Lemma 2)}\\
&= u'_{p_2+1} + |S_1 \downarrow_{l_2} \downarrow_{l_2-1}|,\dots,u'_q + |S_1 \downarrow_{l_2} \downarrow_{n_2-(m+q-p_2)}| && (l_2 = n_2 - m)\\
&= u'_{p_2+1} + |S_1 \downarrow_{l_2} \downarrow_{l_2-1}|,\dots,u'_q + |S_1 \downarrow_{l_2} \downarrow_{l_2-(q-p_2)}| && (l_2 = n_2 - m)\\
&= u'_{p_2+1} + |S_1 \downarrow_{l_2} \downarrow_{n_3-(p_2+1)}|,\dots,u'_q + |S_1 \downarrow_{l_2} \downarrow_{n_3-q}| && (l_2 = n_3 - p_2)\\
&= u'''_{p_2+1},\dots u'''_q
\end{aligned}$$

The remaining cases are similar. □

The order on trees is defined "pointwise" 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 5 (Lifting of ordering). *If $\vec{s} \leq \vec{t}$ (as comparison between jc-sequences), then $\mathit{lift}(E,\vec{s}) \leq \mathit{lift}(E,\vec{t})$ (as comparison between jc trees).*

Proof. Obvious. □

Lemma 6 (Lifting and commit). $\mathit{lift}(E, l{:}\rho, [n,\vec{u}]) \geq \mathit{lift}(E,\vec{u})$.

Proof. Straightforward. □

Lemma 7 (Monotonicity). *If $t_1 \leq t_1$ and $t_2 \leq t'_2$, then $(t_1 \otimes t_2) \leq (t'_1 \otimes t'_2)$.*

Proof. By straightforward calculation. □

Next we prove preservation of well-typedness under reduction, i.e., subject reduction. The proof is split into two parts, preservation under the local reduction rules and preservation on the global level.

Lemma 8 (Subject reduction (local)). *If $n_1 \vdash e_1 :: n_2, h_1, l_1, \vec{s}, S$ and $E_1 \vdash e_1 \rightarrow E_2\ e_2$, then $n_1 \vdash e_2 :: n_2, h_2, l_2, \vec{t}, S$ s.t. $h_2 \leq h_1$, $l_2 \geq l_1$, and $\vec{t} \leq \vec{s}$.*

Proof. In induction on the derivation of the local reduction steps using the rules from Table 2. The cases for field lookup, field update, and object instantiation are immediate. In the proof we concentrate on the parallel weights and the jc-sequences, as the other parts (pre- and post-balance, high and low points) are straightforward.

Case: R-RED: $E \vdash \mathtt{let}\ x : T = v\ \mathtt{in}\ e \rightarrow E \vdash e[v/x]$

The assumption of well-typedness gives

$$\dfrac{n_1 \vdash v :: n_1, n_1, n_1, [n_1], \emptyset \qquad n_1 \vdash t :: n_2, h_2, l_2, \vec{s}, S}{n_1 \vdash \mathtt{let}\ x = v\ \mathtt{in}\ t :: n_2, h_2, l_2, \vec{s}, S}\ \text{T-LET}$$

The $\vec{s}$ in the conclusion is justified by the observation that s_0, the first element of $\vec{s}$, is $\geq n_1$. The result follows from the fact that $n_1 \vdash t : n_2, h_2, l_2, \vec{s}, S$ implies $n_1 \vdash t[v/x] : n_2, h_2, l_2, s, S$, as required.

Case: R-COND$_1$: $E \vdash \texttt{let}\, x : T = (\texttt{if true then}\ e_1\ \texttt{else}\ e_2)\ \texttt{in}\, e \rightarrow E \vdash \texttt{let}\, x : T = e_1\ \texttt{in}\, e$
By well-typedness we are given

$$\frac{n \vdash e_1 :: n', h_1, l_1, \vec{s}, S \qquad n \vdash e_2 :: n', h_2, l_2, \vec{t}, S}{n \vdash \texttt{if}\ v\ \texttt{then}\, e_1\ \texttt{else}\, e_2 :: n', h_1 \vee h_2, l_1 \wedge l_2, \vec{s} \vee \vec{t}, S}$$

The case follows from the fact that $\vec{s} \leq \vec{s} \vee \vec{t}$. The case for R-COND$_2$ works symmetrically.

Case: R-LET: $E \vdash \texttt{let}\, x_2 : T_2 = (\texttt{let}\, x_1 : T_1 = e_1\ \texttt{in}\, e_2)\ \texttt{in}\, e_3 \rightarrow E \vdash \texttt{let}\, x_1 : T_1 = e_1\ \texttt{in}\ (\texttt{let}\, x_2 : T_2 = e_2\ \texttt{in}\, e_3)$
We are given:

$$\frac{\dfrac{n_1 \vdash e_1 :: n_2, h_1, l_1, \vec{s}, S_1 \qquad n_2 \vdash e_2 :: n_3, h_2, l_2, \vec{t}, S_2}{n_1 \vdash \texttt{let}\, x_1 = e_1\ \texttt{in}\, e_2 :: n_3, h_1 \vee h_2, l_1 \wedge l_2, \vec{v}, S_1 \downarrow_{l_2} \cup S_2} \qquad n_3 \vdash e_3 :: n_4, h_3, l_3, \vec{u}, S_3}{n_1 \vdash \texttt{let}\, x_2 = (\texttt{let}\, x_1 = e_1\ \texttt{in}\, e_2)\ \texttt{in}\, e_3 :: n_4, (h_1 \vee h_2) \vee h_3, (l_1 \wedge l_2) \wedge l_3, \vec{w}, (S_1 \downarrow_{l_2} \cup S_3) \downarrow_{l_3} \cup S_3}$$

where $\vec{w} = (\vec{s} \oplus_{p_1} (S_1 \otimes_{n_2} \vec{t})) \oplus_{p_2} (S_2 \cup S_1 \downarrow_{l_2} \otimes_{n_3} \vec{u})$ and we need to prove

$$\frac{n_1 \vdash\ e_1 :: n_2, h_1, l_1, \vec{s}, S_1 \qquad \dfrac{n_2 \vdash e_2 :: n_3, h_2, l_2, \vec{t}, S_2 \qquad n_3 \vdash e_3 :: n_4, h_3, l_3, \vec{u}, S_3}{n_2 \vdash \texttt{let}\, x_2 = e_1\ \texttt{in}\, e_2 :: n_4, h_2 \vee h_3, l_2 \wedge l_3, \vec{v}', S_2 \downarrow_{l_3} \cup S_3}}{n_1 \vdash \texttt{let}\, x_1 = e_1\ \texttt{in}\ (\texttt{let}\, x_2 = e_2\ \texttt{in}\, e_3) :: n_4, h_1 \vee (h_2 \vee h_3), l_1 \wedge (l_2 \wedge l_3), \vec{w}', S_1 \downarrow_{l_2 \wedge l_3} \cup (S_2 \downarrow_{l_3} \cup S_3)}$$

where $\vec{w}' = \vec{s} \oplus_{p_1} (S_1 \otimes_{n_2} (\vec{t} \oplus_{p_2} (S_2 \otimes_{n_3} \vec{u})))$. For high and low points, we use associativity of $\vee$ and $\wedge$ For parallel weights, we use associativity from Lemma 3. Finally, $\vec{w} = \vec{w}'$ follows from Lemma 4.

Case: R-LOOKUP, R-UPD, and R-NEW
Trival, as not transactions are involved and no threads are spawned.

Case: R-CALL
Straightforward.

Lemma 9 (Subject reduction).

$$\Gamma \vdash P :: t \textit{ and } \Gamma \vdash P \Longrightarrow \Gamma' \vdash P' \textit{ implies } \Gamma' \vdash P' :: t' \textit{ where } t' \leq t.$$

Proof. By induction on the derivation/derivation tree of the reduction step $\Gamma \vdash P \Longrightarrow \Gamma' \vdash P'$ by the rules of the semantics.

Case: G-PLAIN
A consequence of subject reduction for the local level (Lemma 8), the compatibility of the orders for the sequences on the local level and the trees on the global level (Lemma 5) and fact that the reflect-function does not change the length of the local environments (cf. Definition 2).

Case: G-SPAWN
We are given $\Gamma \vdash p\langle \texttt{let}\, x : T = \texttt{spawn}\, e_1\ \texttt{in}\, e_2\rangle \Longrightarrow \Gamma' \vdash p\langle \texttt{let}\, x : T = \texttt{null}\ \texttt{in}\, e_2\rangle \parallel p'\langle e_1\rangle$. Well-typedness of the configuration before the steps gives

$$\dfrac{\dfrac{\dfrac{n_1 \vdash e_2 :: 0,h_2,0,\vec{u},S_2}{n_1 \vdash \texttt{spawn}\, e_2 :: n_1,n_1,n_1,[n_1+u_0],S_2 \cup \{(n_1,h_2-n_1)\}} \qquad n_1 \vdash e_1 :: n_2,h_1,0,\vec{v},S_1 \qquad S = S_1 \cup S_2' \downarrow_0}{n_1 \vdash \texttt{let}\, x = \texttt{spawn}\, e_2\ \texttt{in}\, e_1 :: 0,h_1,0,\vec{s},S}}{p_1{:}E \vdash p_1\langle \texttt{let}\, x = \texttt{spawn}\, e_1\ \texttt{in}\, e_2\rangle :: \mathit{lift}(E,\vec{s})}$$

were $n_1 = |E|$. For the configuration after the step, we can derive with rules T-PAR, T-THREAD, T-LET, and T-NULL:

$$\dfrac{\dfrac{n_1 \vdash \texttt{null} :: n_1,n_1,n_1,[n_1],\emptyset \qquad n_1 \vdash n_2,h_1,0,\vec{v},S_1}{p_1{:}E \vdash p_1\langle \texttt{let}\, x = \texttt{null}\ \texttt{in}\ e_1\rangle :: \mathit{lift}(E,\vec{v})} \qquad \dfrac{n_1 \vdash e_2 :: 0,h_2,0,\vec{u},S_2}{p_2{:}E \vdash p_2\langle e_2\rangle :: \mathit{lift}(E,\vec{u})}}{p_1{:}E,p_2{:}E \vdash p_1\langle \texttt{let}\, x = \texttt{null}\ \texttt{in}\ e_1\rangle \parallel p_2\langle e_2\rangle :: \mathit{lift}(E,\vec{v}) \otimes \mathit{lift}(E,\vec{u})}$$

where $S_2' = S_2 \cup \{(n_1,h_2-n_1)\}$. We need to prove that $\mathit{lift}(E,\vec{s}) = \mathit{lift}(E,\vec{v}) \otimes \mathit{lift}(E,\vec{u})$. The proof of this equation follows straightforwardly from Definition 10 of $\otimes$. Note that the two trees are both linear and their nodes are labeled by the same labels (cf. the definition of the *lift*-function).

Case: G-TRANS
We are given $p{:}E \vdash p\langle \texttt{let}\, x = \texttt{onacid}\ \texttt{in}\, e\rangle \Longrightarrow p{:}E' \vdash p\langle \texttt{let}\, x = \texttt{null}\ \texttt{in}\, e\rangle$. Well-typedness of the configuration before the step gives:

$$\dfrac{\dfrac{n_1 \vdash \texttt{onacid} :: n_1+1,n_1+1,n_1,[n_1+1],\emptyset \qquad n_1+1 \vdash e :: 0,h,0,\vec{s},S}{n_1 \vdash \texttt{let}\, x = \texttt{onacid}\ \texttt{in}\, e :: 0,h,0,(s_0 \vee s_1,s_2,\ldots),S}}{p{:}E \vdash p\langle \texttt{let}\, x = \texttt{onacid}\ \texttt{in}\, e\rangle :: \mathit{lift}(E,(s_0 \vee s_1,s_2,\ldots))}$$

Note that the *start*-function used in the G-TRANS-step to update the local environment assures that $|E'| = |E|+1$ (cf. Definition 2(3)). Note further that in the application of rule T-LET, we know that $n+1 \geq s_0$, and thus $n+1 \vee s_0 \vee s_1$ equals to $s_0 \vee s_1$. For the configuration after the step, we can derive with T-THREAD, T-LET, and T-NULL

$$\dfrac{\dfrac{n_1+1 \vdash \texttt{null} :: n_1+1,n_1+1,n_1+1,[n_1+1],\emptyset \qquad n_1+1 \vdash e :: 0,h,0,\vec{s},S}{n_1+1 \vdash \texttt{let}\, x\ \texttt{null}\ \texttt{in}\, e :: 0,h,0,(s_0 \vee s_1,s_2,\ldots),S}\ \text{T-LET}}{p{:}E' \vdash p\langle \texttt{let}\, x = \texttt{null}\ \texttt{in}\, e\rangle :: \mathit{lift}(E,(s_0 \vee s_1,s_2,\ldots))}$$

Case: G-COMM
We are given $\Gamma \vdash \ldots \parallel p_i\langle \texttt{let}\, x = \texttt{commit}\ \texttt{in}\ e_i\rangle \parallel \ldots \Longrightarrow \Gamma' \vdash \parallel \ldots p_i\langle \texttt{let}\, x = \texttt{null}\ \texttt{in}\ e_i\rangle \parallel \ldots$. Well-typedness of the configuration before the step gives for each p_i

$$\dfrac{\dfrac{n_i \vdash \texttt{commit} :: n_i-1,n_i,n_i-1,[n_i,n_i-1],\emptyset \qquad n_i-1 \vdash e_i :: 0,h,0,\vec{u}_i,S}{n_i \vdash \texttt{let}\, x = \texttt{commit}\ \texttt{in}\, e_i :: 0,h_i,0,[n_i,\vec{u}_i],S}}{p_i{:}E_i \vdash p_i\langle \texttt{let}\, x = \texttt{commit}\ \texttt{in}\, e_i\rangle :: \mathit{lift}(E,[n_i,\vec{u}_i])}$$

Note that $(n_i - 1 \vee u_{i_0}) = u_{i_0}$ since $u_{i_0} \geq n_i - 1$ and $|\vec{u}_i| = n_i$ because all the `onacids` are committed at the end (cf. T-THREAD). By T-THREAD, T-LET, and T-NULL we can derive

$$\cfrac{\cfrac{n_i - 1 \vdash \texttt{null} :: n_i - 1, n_i - 1, n_i - 1, [n_i - 1], \emptyset \qquad n_i - 1 \vdash e_i :: 0, h_i, 0, \vec{u}_i, S}{n_i - 1 \vdash \texttt{let}\, x = \texttt{null in}\ e_i :: 0, h_i, 0, \vec{u}_i, S}}{p_i{:}E'_i \vdash p_i\langle \texttt{let}\, x = \texttt{null in}\ e_i \rangle :: \mathit{lift}(E'_i, \vec{u}_i)}$$

where $E_i = E'_i, l{:}\rho$, i.e., $|E'_i| = |E_i - 1|$ and (cf. Definition 2 and rule G-COMM). By Lemma 6, $(\mathit{lift}(E'_i, \vec{u}_i) \leq (\mathit{lift}(E_i, [n_i, \vec{u}_i])$, and therefore by monotonicity from Lemma 7, $\bigotimes_i(\mathit{lift}(E'_i, \vec{u}_i) \leq (\bigotimes_i \mathit{lift}(E_i, [n_i, \vec{u}_i])$, as required.

Case: G-COMM-ERROR
Omitted, since the formulation of subject reduction covers only non-erronous states. A type and effect system which prevents statically that such erroneous steps ("commit errors") occur has been formalized in [15]). □

The next lemma states a simple property of the initial weight of join-trees.

Lemma 10. $|t_1 \otimes t_2|_1 = |t_1|_1 + |t_2|_1$

Proof. Straightforward from the definition. □

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

Lemma 11. *If $\Gamma \vdash P :: t$, then $|\Gamma| \leq |t|_1$.*

Proof. By induction on the derivation of $\Gamma \vdash P :: t$.

Case: T-THREAD
Only one thread, current resource consumption is $|E|$. The weight estimated by t (which basically is a sequence) larger than the first element of t (or of s). That's easy to see by the local typing rules.

Case: T-PAR
We are given

$$\frac{\Gamma_1 \vdash P_1 :: t_1 \qquad \Gamma_2 \vdash P_2 :: t_2}{\Gamma_1, \Gamma_2 \vdash P_1 \parallel P_2 :: t_1 \otimes t_2}$$

Using induction on the two sub-goals gives $|\Gamma_1| \leq |t_1|_1$ and $|\Gamma_2| \leq |t_2|_1$ and the result follows by Lemma 10 and the fact that $|\Gamma_1, \Gamma_2| = |\Gamma_1| + |\Gamma_2|$. □

This brings us to the final result as a corollary of subject reduction and the previous lemma: the resource consumption calculated is a static over-approximation for all reachable configurations of the program.

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

Proof. An immediate consequence of subject reduction (Lemma 9) and Lemma 11. □

6 Conclusion

We formalized a static, compositional effect-based analysis to estimate the resource bounds for an object-oriented calculus supporting nested and multi-threaded transactions (TFJ). 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. 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. The main challenge is that the execution model of TFJ has neither independent parallelism nor full sequentialization. Instead, synchronization is affected by the nesting structure of the multi-threaded transactions. This means, the synchronization structure is not syntax-directed, which complicates the analysis. To our knowledge, this is the first static analysis taking care of memory resource consumption for *transactional* software programs. Abstracting away from the specifics of memory consumption, the effect system presented here can be seen as a careful, compositional account of a parallel model based on join-synchronization (in particular that of TFJ). It is promising to use compositional techniques as explored here also to achieve different program analyses in a similar manner for programs based on fork/join parallelism.

Related work Estimating memory usage has be studied, however, in various other settings. Concerning functional languages, Hughes and Pareto [12] introduce a strict, first order functional language with explicit regions and give a type system with space effects which guarantees that well-typed programs use at most the space specified by the programmers. [9] is a treatment of time as a resource. Their system certifies a time limit for a complete functional program, by using annotations by the programmer of time limits for each individual function. Hofmann and Jost [10] 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 technique. [18] deals with 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. [6] 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 [11], 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. Their bounds are not precise and can be over-approximated. In [5] the authors present an algorithm to statically compute memory consumption of a method as a non-linear function of method's parameters. The bounds are not precise. Their work is not type-based and the language does not include explicit de-allocation. Braberman *et al.* [3] calculate non-linear symbolic approximation of memory bounds for Java-like methods and then apply mathematical results for optimization problem to find the concrete memory bound. However the bounds are not easily precise due to various factors. A similar technique is also presented in [8].

For low-level languages, [4] uses program logics to infer precise memory consumption of sequential bytecode programs with resource annotation by pre- and postconditions. The language does not have explicit de-allocation. In [2], Albert *et al.* compute memory consumption of a program as a function of its input data. They also refine program's functions by using escape analysis [7] to collect objects that do not escape their scopes. The bytecode language has neither explicit de-allocation nor scope. Later in [1] they introduce a more powerful method to calculate precise peak heap memory consumption that take into account implicit de-allocation (garbage collected memory). Pham *et al.* [16] propose a fast algorithm with small memory footprint to statically calculate heap memory for a class of JavaCard programs.

The main difference of this 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 a restricted to sequential language— but also the complex and implicit synchronization structure entailed by the transactional model. The work in [17], 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* construct in the investigated calculus.

Future work We plan to refine the effect system by annotating 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. Extending the language with exception handling is also one possibility. The result of our analysis could be an input for a "hybrid" model which can switch between transaction-based and lock-based modes depending on resource consumption.

References

1. E. Albert, S. Genaim, and M. G.-Z. Gil. Live heap space analysis for languages with garbage collection. In *International Symposium on Memory Management*, 2009.
2. E. Albert, S. Genaim, and M. Gomez-Zamalloa. Heap space analysis for Java bytecode. In *ISMM '07*, New York, NY, USA, 2007. ACM.
3. D. Aspinall, R. Atkey, K. MacKenzie, and D. Sannella. *Symbolic and Analytic Techniques for Resource Analysis of Java Bytecode*. 2010.
4. G. Barthe, M. Pavlova, and G. Schneider. Precise analysis of memory consumption using program logics. In *SEFM '05*, Washington, DC, USA, 2005. IEEE.
5. 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.
6. W.-N. Chin, H. H. Nguyen, S. Qin, and M. C. Rinard. Memory usage verification for OO programs. In C. Hankin and I. Siveroni, editors, *SAS*, volume 3672 of *LNCS*. Springer, 2005.
7. J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for Java. *SIGPLAN Not.*, 34(10), 1999.
8. P. Clauss, F. J. Fernandez, D. Garbervetsky, and S. Verdoolaege. Symbolic polynomial maximization over convex sets and its application to memory requirement estimation. *IEEE Transactions on Very Large Scale Integration Systems*, 17, 2009.
9. K. Crary and S. Weirich. Resource bound certification. In *POPL '00*. ACM, 2000.
10. M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *POPL '03*, New York, NY, USA, 2003. ACM.
11. M. Hofmann and S. Jost. Type-based amortised heap-space analysis (for an object-oriented language). In P. Sestoft, editor, *ESOP'06*, volume 3924 of *LNCS*. Springer, 2006.

12. J. Hughes and L. Pareto. Recursion and dynamic data-structures in bounded space: towards embedded ML programming. *SIGPLAN Not.*, 34(9), 1999.
13. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA) '99*, pages 132–146. ACM, 1999. In *SIGPLAN Notices*.
14. S. Jagannathan, J. Vitek, A. Welc, and A. Hosking. A transactional object calculus. *Science of Computer Programming*, 57(2):164–186, Aug. 2005.
15. T. Mai Thuong Tran and M. Steffen. Safe commits for Transactional Featherweight Java. In D. Méry and S. Merz, editors, *Proceedings of the 8th International Conference on Integrated Formal Methods (iFM 2010)*, volume 6396 of *Lecture Notes in Computer Science*, pages 290–304 (15 pages). Springer-Verlag, Oct. 2010. An earlier and longer version has appeared as UiO, Dept. of Comp. Science Technical Report 392, Oct. 2009 and appeared as extended abstract in the Proceedings of NWPT'09.
16. T.-H. Pham, A.-H. Truong, N.-T. Truong, and W.-N. Chin. A fast algorithm to compute heap memory bounds of Java Card applets. In *SEFM'08*, 2008.
17. H. Truong and M. Bezem. Finding resource bounds in the presence of exlicit deallocation. In *ICTAC'05*, volume 3722 of *Lecture Notes in Computer Science*, pages 227–241. Springer-Verlag, 2005.
18. L. Unnikrishnan, S. D. Stoller, and Y. A. Liu. Optimized live heap bound analysis. In *VMCAI 2003*, London, UK, 2003. Springer.

Index

$\Gamma \vdash P \Longrightarrow \Gamma' \vdash P'$ (global transition), 11
$S = \{(p_0, c_0), \ldots, (p_k, c_k)\}$ (parallel weight), 13
$E \vdash e \rightarrow E' \vdash e'$ (local transition), 9
Γ (global environment), 11
$\Gamma \vdash P$ (global configuration), 11
$\Gamma \vdash e : T$ (typing judgment), 13
$par(S, l)$, 14
$S \downarrow_l$, 14
$seq(S, l)$, 14
$|S|$, 14
$|S|$ (resource consumption of S), 14
$\oplus_p$, 14
$\otimes_l$, 14
lift (lifting), 17
$|t|_1$ (initial weight of t), 17
$|t|$ (weight of t), 17
$p\langle e \rangle$ (thread), 7
ρ (log), 9
$n_1 \vdash e :: n_2, h, l, \vec{t}, S$ (effect judgment), 13
$t_1 \otimes t_2$ (parallel composition), 17

balance, 13

class definition, 7
compositionality, 4
configuration
– global, 11
constructor, 7

E (local environment), 9
environment
– global, 11
– local, 9
extend, 10

field, 7
$fv(e)$ (free variables), 8

global transition, 11

implicit synchronization, 6
intranse, 12

jc-sequence, 13
– composition, 14
jc-tree
– parallel composition, 17
join, 15
joining commit, 4
joining commit sequence, 13
judgment, 13

l (transaction label), 9
lifting, 17
local environment, 9
local transition, 9

method definition, 7
mtype, 15

nested, 12

`onacid`, 11
overloading, 8

parallel weight, 13
– associativity, 18

read, 10
resource consumption, 18

sequential composition, 6
sequential composition and let-construct, 8
subject reduction, 21, 22

TFJ, 7
transaction
– multi-threaded, 4
– nested, 4
transaction label, 9
transaction labels, 12
Transactional Featherweight Java, 7

weight, 17
write, 10

List of Tables

1 Abstract syntax 7
2 Semantics (local) 11
3 Semantics (global) 12
4 Effect system 15
5 Effect system 17