This is a revised version of the technical report. We are indebted to the anonymous reviewers of earlier versions for their thorough reviews and suggestions, which has lead to extensions of the formalization and the proofs.
Compositional Static Analysis for Multithreaded Transactions with Join Synchronization

Thi Mai Thuong Tran\textsuperscript{1}, Martin Steffen\textsuperscript{1}, and Hoang Truong\textsuperscript{2}

\textsuperscript{1}Department of Informatics, University of Oslo, Norway
\textsuperscript{2}University of Engineering and Technology, VNU Hanoi

Abstract. We present an effect-based static analysis to calculate upper and lower bounds on multithreaded 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) \cite{22,11} 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 rollbacks 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) \cite{18}. The language features non-lexical starting and ending a transaction, concurrency, choice and sequencing. The analysis is compositional, i.e., syntax-directed. The analysis is \textit{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.

```plaintext
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. 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.

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 \([...[...]...] [...]...]\)), \( e_2 \), four, \( e_3 \), five, and \( e_4 \), six. The resource consumption after spawning \( e_2 \)'s thread and before the subsequent commit 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 and thus usable in practice, the analysis should be compositional. This syntax-directedness is common for type/effect-based analyses. Here, 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 \( e_1 || e_2 \) 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. It should be clear that one would (more) easily obtain a sound resource estimation assuming independent parallelism. Ignoring those synchronization points, however, would entail a loss of precision. For instance, without taking the joining commits into account, i.e., ignoring that their respective maximal values cannot occur at the same time, the resource consumption in Example 2 would have to be overapproximated by the sum of the maximal resource consumption of the 3 involved threads, yielding 19.

**Example 3.** Let us split the code of Example 1 after the first spawn, i.e., at the semicolon at the end of line 3 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:

\[
\vdash e_l :: m_1 \quad \vdash e_r :: m_2 \quad m = f(m_1, m_2)
\]

\[
\vdash e_l; e_r :: m
\]

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 \(m\) in the conclusion 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.

![Fig. 2: Compositional analysis (sequential composition \(e_l; e_r\))](image)
Table 1: Abstract syntax

(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 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 [18]. The main adaptations, as in [19], 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

FJ is a core language originally introduced to study typing issues related to Java, such as inheritance, sub-type 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 [18] 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 I shows the abstract syntax of TFJ. A program consists of a number of processes/threads $p(e)$ running in parallel, where $p$ is the thread's identifier and $e$ the expression being executed. The empty process is written $0$. The syntactic category $L$ captures class definitions. In absence of inheritance, a 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 $C\{\vec{f};\vec{T}\}\{\text{this.$\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 over-
loading. 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 implicitly assume standard values like booleans, integers, \ldots ; besides those, values
can be object references \( r \), variables \( x \) or \( \text{null} \). 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
\( \text{new} \ C(\vec{v}) \). The next two expressions deal with the basic, sequential control structures: \( \text{if } v \text{ then } e_1 \text{ else } e_2 \)
represents conditions, and the let-construct \( \text{let } x:T = e_1 \text{ in } e_2 \) 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 \( \text{let } x:T = e_1 \text{ in } e_2 \) 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: \( \text{spawn} \ e \) starts a new thread of activity which evaluates \( e \) in parallel
with the spawning thread. Specific for TFJ are the two dual constructs \text{onacid} \ and \text{commit}.
The expression \text{onacid} \ starts a new transaction and executing \text{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 re-
stricted concerning where to use general expressions \( e \). E.g., Table \ref{tab: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 \cite{18} 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: \cite{18} 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 \( \text{let } x_1 = e_1 \text{ in } (\text{let } x_2 = e_2 \text{ in } x_1.f := x_2) \), making the evaluation order explicit.
The transformation from the general syntax to the one of Table \ref{tab:1} is standard. 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 \ref{tab:2} deals with the evaluation of \textit{one expression/thread} and reducing con-
Local transitions are thus of the form

\[ E \vdash e \rightarrow E' \vdash e' \],

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 \( l_1:p_1, \ldots, l_k:p_k \), i.e., of pairs of transaction labels \( l_i \) and a corresponding log \( \rho_i \). We write \( |E| \) for the size of the local environment, i.e., the number of pairs \( l:p \) 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 \( \rho_i \) can be thought of as “local copies” of the heap. The log \( \rho_i \), a sequence of mappings from references to values, is used to keep track of changes by a thread in 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., [18] gives two different flavors, a “pessimistic”, lock-based one and an “optimistic” one).

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 \( (e_1; e) \rightarrow e_1; (e; e') \). 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 restriction 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 \( \text{read, write, and extend} \) (which look-up a reference, update a reference, resp. allocate a new reference on the heap). Note that also the \text{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 \text{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 \text{fields}(C)\ looks up fields of class \( C \) and \text{mbody}(C, m)\ looks up the method \( m \) of class \( C \). So, field look-up in R-FIELD works as follows: consulting the local environment \( E \), the \text{read}-function looks up the object referenced by \( r \); the object is \( C(u) \), i.e., it’s an instance of class \( C \), and its fields carry the
values \( \bar{u} \). The (run-time) type \( C \) of the object is further used to determine the fields \( \vec{f} \), using the object referenced by \( r \), where \textit{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 \textit{read} to look up the objects, and additionally using \textit{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 \( \textit{mbody} \) 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 \( \vec{f} \) 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 \( C(\text{null}) \), which has all fields initialized to \text{null}. The function \( \textit{extend} \) in that rule extends \( E \) by binding the fresh reference \( r \) to the newly created instance.

\[
\begin{align*}
E \vdash \text{let } x : T = e & \quad \text{R-RED} \\
E \vdash \text{let } x_1 : T_1 = \{ \text{let } x_1 : T_1 = e_1 \text{ in } e \} & \quad \text{R-LET} \\
E \vdash \text{let } x : T & = \{ \text{if true then } e_1 \text{ else } e_2 \} & \quad \text{R-COND}_1 \\
E \vdash \text{let } x : T & = \{ \text{false} \} & \quad \text{R-COND}_2 \\
\text{read}(E,r) = E.C(\vec{a}) & \quad \text{read}(E,r) = E'.C(\vec{a}) \quad \text{fields}(C) = \vec{f} \\
\text{write}(r \leftarrow E.C(\vec{a}); e') & \quad \text{write}(r \leftarrow E'.C(\vec{a}); e') = E'' \\
E \vdash \text{let } x : T & = r.f_i \text{ in } e & \quad \text{R-LOOKUP} \\
\text{read}(E,r) = E'.C(\vec{a}) & \quad \text{R-CALL} \\
E \vdash \text{let } x : T & = r.f_i \leftarrow e & \quad \text{R-NEW} \\
\hline
\end{align*}
\]

Table 2: Semantics (local)

The rules of the \textit{global} semantics are given in Table 3. The semantics works on configurations of the form

\[
\Gamma \vdash P ,
\]

where \( P \) is a \textit{program} and \( \Gamma \) is a global environment. Besides that, we need a special configuration \textit{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 \( p_e \), we need one \( E \) for each thread \( p \). This means, \( \Gamma \) 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.

\textbf{Definition 2 (Global environment).} A global environment \( \Gamma \) 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:

\[
\Gamma \vdash P \implies \Gamma' \vdash P' \quad \text{or} \quad \Gamma \vdash P \implies \text{error} .
\]
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 [18]. As the functions’ concrete details are irrelevant for our static analysis, we refer the interested reader to [18] 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 erroneous use of the commit-statement outside any transaction. In G-COMM, $l$ is the label of the transaction to be committed and the function intranse 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 \( \text{reflect}(p, E, \Gamma) = \Gamma' \) 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 \( \text{spawn}(p, p', \Gamma) = \Gamma' \), then \( \Gamma' = \Gamma, p': E' \) s.t. \( |E| = |E'| \).

3. The function start satisfies the following condition: if \( \text{start}(l, p, \Gamma) = \Gamma' \) for \( \Gamma = p_1: E_1, \ldots, p_k: E_k \) and for a fresh $l$, then \( \Gamma' = p_1: E_1, \ldots, p_k: E_k' \) 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 \( \text{intranse}(\Gamma, l) = \beta \), then

<table>
<thead>
<tr>
<th>$\Gamma \vdash p: E$</th>
<th>$E \vdash e \rightarrow E' \vdash e'$</th>
<th>$\text{reflect}(p, E, \Gamma) = \Gamma'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash p \parallel p(e) \implies \Gamma'' \parallel p(p')$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\Gamma' \parallel p(l: \Gamma') \parallel \Gamma''$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\Gamma \vdash p \parallel {\text{let } x : T = \text{spawn } e \in e_2} \implies \Gamma'' \parallel p \parallel {\text{let } x : \Gamma = \text{null in } e_2} \parallel p'(e_1)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l$ fresh \hspace{1cm} $\text{start}(l, p, \Gamma) = \Gamma'$</td>
</tr>
</tbody>
</table>

| $\Gamma \parallel p \parallel \{\text{let } x : T = \text{onacid in } e\} \implies \Gamma'' \parallel p \parallel \{\text{let } x : T = \text{null in } e\}$ |
|------------------|-------------------------------|--------------------------------|
| $\Gamma = \Gamma'', p: E, E = E', l: \rho \hspace{1cm} \text{intranse}(\Gamma, l) = p = p_1, \ldots, p_k$ |
| $\mathrm{commit}(\beta, E, \Gamma') = \Gamma'' \parallel p_1: E_1, p_2: E_2, \ldots, p_k: E_k \in E' \hspace{1cm} E = E_1, E_2, \ldots, E_k$ |

**Table 3: Semantics (global)**
(a) $p \in \vec{p}$ and 
(b) for all $p_i \in \vec{p}$ we have $\Gamma = \ldots, p_i : (E_i', l; p_i), \ldots$ 
(c) for all threads $p'$ with $p' \notin \vec{p}$ and $\Gamma = \ldots, p' : (E', l'; p'), \ldots$, we have $l' \neq l$.

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

**Definition 4.** Let $\text{TName}$ be the type of transaction labels. Given a local environment $E$, the function $\text{l} : (\text{LEnv} \rightarrow \text{List of TName})$ is defined inductively as follows: $\text{l}(\varepsilon) = \varepsilon$, and $\text{l}(l; \ldots, E) = 1, l(E)$. Overloading the definition, we lift the function straightforward to global environments (with type $\text{l} : \text{TName} \times \text{GEnv} \rightarrow \text{List of TName}$), s.t. $\text{l}(p, (p; E), \Gamma) = l(E)$.

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 $\text{nested} : \text{TName} \times \text{GEnv} \rightarrow \text{List of TName}$ returns the list of all threads nested inside a given transaction.

### 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 $\Gamma$, 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:

$$n_1 \vdash e :: n_2, h, l, \bar{i}, S,$$  

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 $e$ (the “highest” and the “lowest” balance during execution). The numbers so far describe the balances of the thread executing $e$. Executing $e$, however, may spawn new child threads and the remaining elements $\bar{i}$ and $S$ take their contribution into account. Roughly speaking, the information $S$ is needed to achieve compositionality wrt. sequential composition and $\bar{i}$ 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 12 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. 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. In contrast, \( \tilde{\ell} \) is needed for compositionality wrt. parallel composition. The \( \tilde{\ell} \) 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 \( \tilde{\ell} \) a joining-commit sequence, or \( \text{jc-sequence} \) for short. In Example 3, the right-hand expression \( \text{spawn } (e_2)([e_3]e_4) \) 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 \( \tilde{\ell} = 10, 7 \) (where \( 10 = ((4 + 3) + 3) \lor ((5 + 2) + 2) \) and \( 7 = 6 + 1 \)). For uniformity, we use \( \lor \) resp. \( \land \) 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 trivial, 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. To achieve a more fine-grained model would mean to add an appropriate estimation of memory consumption as non-trivial effect to those rules. The estimation could be made dependent on the type of the value accessed, but the formulation is orthogonal to the problems of synchronization and concurrency. 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 \( \tilde{\ell} \) 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 \( \tilde{u} \) 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 \( p_i, c_i \) 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:

\[
\begin{align*}
\text{par}(S,l) &= \{(p,c) \in S \mid p \leq l\} \\
\text{seq}(S,l) &= \{(p,c) \in S \mid p > l\} \\
[S]_l &= \{(l,0),(l,0),\ldots\} \\
S_{\downarrow l} &= \text{par}(S,l) \cup \text{seq}(S,l)_{\downarrow l}
\end{align*}
\]
where for $|S|_i$, the number of tuples in $S$ equals the number of $(1,0)$ in $|S|_i$.

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

1. The part $\text{par}(S_1, l_2)$ of $S_1$ unaffected by a commit in $e_2$ and thus able to run in parallel with $e_2$.
2. The part $\text{seq}(S_1, l_2)$ 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 $\text{par}(S_1, l_2)$ consists of the half of $S_1$ unaffected by any join synchronization. Even if $\text{seq}(S_1, l_2)$ 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 $\{\text{seq}(S_1, l_2)\}_{l_2}$ calculates that remaining contribution. So $|S_1|_{l_2}$ 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 $S_1 \cup l_2 \cup S_2$ 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 7 (Sequential composition of jc-sequences-\(x\)).** 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: $\vec{s} \oplus_p \vec{t} = s_0 \lor t_0, \ldots, s_p \lor t_{p+1}, \ldots, t_m$. 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 $t_0' = t_0 + |S|$, $t_1' = t_1 + |S|_{n-1}$, $\ldots$, $t_m' = t_m + |S|_{n-m}$.

The compositional calculation of the jc-sequence $\vec{u}$ (cf. Definition 7) 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 $\otimes\_n$ operation of Definition 7. Secondly, joining commits of $e_2$ may no longer be joining commits of the composed expression $\text{let } x = e_1 \text{ in } e_2$. 

![Fig. 3: Sequential composition of jc-sequences (cf. Definition 7)](image-url)
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 $e_1;e_r$, 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 $\lor$) 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 $\vec{s} \oplus_p (S_1 \ominus_{n_2} \vec{t})$. 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-balances $n_3$ 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 $\text{let } x = e_1 \text{ in } e_2$, which makes the compositional treatment of the sequential composition complicated. The treatment of
The order relation on \( \vec{c} \)-sequences (of equal length) \( \vec{s} \leq \vec{t} \) is defined pointwise and we write \( \vec{s} \lor \vec{t} \) 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. \(\sqsubseteq\) (cf. Lemma 7 which states the existence of the least upper bound).

**Definition 8 (Order).** The order relation on \( \vec{c} \)-sequences (of equal length) \( \vec{s} \leq \vec{t} \) is defined pointwise and we write \( \vec{s} \lor \vec{t} \) 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. \(\sqsubseteq\) (cf. Lemma 7 which states the existence of the least upper bound).

**Lemma 1 (Least upper bound).** The order relation \(\sqsubseteq\) on parallel weight (Definition 8) has a least upper bound.

**Proof.** Given \( S_1 \) and \( S_2 \). Given a natural number \( p \), let \( S_1^p \) be defined as the multi-set \( \{(p, c) \mid (p, c) \in S_1\} \), and analogously for \( S_2^p \). Given a fixed \( p \), assume that both multisets \( S_1^p \) and \( S_2^p \) are ordered decreasingly, i.e., \( S_1^p = \{(p, c_1), \ldots, (p, c_k)\} \) such that \((p, c_i) \sqsubseteq (p, c_{i+1})\) for all \( i \) (which means \( c_i \geq c_{i+1} \), for all \( i \)). Analogously for \( S_2^p = \{(p, c'_1), \ldots, (p, c'_m)\} \). Wlog., assume \( k \leq m \).
Now let \( S_1^p \sqcup S_2^p \) be defined as the multi-set \( \{(p, c''_1), \ldots, (p, c''_j), (p, c''_{j+1}), \ldots, (p, c''_{k+j})\} \), where \( c''_i \) is given as the maximum of \( c_i \) and \( c'_i \). Then \( S_1 \sqcup S_2 \) is defined “pointwise”, i.e., as \( (S_1^p \sqcup S_2^p) \sqcup \ldots \sqcup (S_1^n \sqcup S_2^n) \), for all values \( p \), occurring in \( S_1 \sqcup S_2 \).
That \( S_1 \sqcup S_2 \) thus defined is the least upper bound wrt. \(\sqsubseteq\) rests on the following observation. Assume one particular value of \( p \) fixed and let \( S_1 \) and \( S_2 \) both contain only elements of the form \((p, c)\). Let’s interpret \( S_1 \) and \( S_2 \) not as multi-sets but lists in the following way: \( S_1^{ord} = \{(p, c_1), \ldots, (p, c_k) \mid (p, c) \in S_1\} \) for all \( 1 \leq i \leq k \) and \( c_j \geq c_{j+1} \), for all \( 1 \leq j \leq k - 1 \); analogously for \( S_2^{ord} \). Let furthermore define \(\sqsubseteq\) as order on lists as follows: \(\{(p, c_1), \ldots, (p, c_k)\} \sqsubseteq \{(p, c'_1), \ldots, (p, c'_j), (p, c'_{j+1}), \ldots\}\) iff \( c_i \leq c'_i \), for all \( 1 \leq i \leq k \). It’s easy to see, that \( S_1 \sqsubseteq S_2 \) iff \( S_1^{ord} \sqsubseteq S_2^{ord} \). It’s furthermore easy to see that the least upper bound wrt. \(\sqsubseteq\) exists and corresponds to the above-given definition of \(\sqsubseteq\).

Coming back to rule T-COND for conditionals: 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. Similarly, the common \( \vec{c} \)-sequence and the common parallel weight is determined by the corresponding least upper bounds of the two branches. When spawning a new thread \( e \) (cf. rule T-SPAWN), the pre-condition \( m_1 \) 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 \( m_1 \), as well. The \( \vec{c} \)-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 \( m_1 \) 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 \( m \)’s type in the given class table (the function is standard and its definition is omitted here). The appropriate adaption of the interface information concerning \( \vec{t} \) and \( S \) is defined as follows:

**Definition 9 (Shift).** Given a natural number \( n \), the addition \( \vec{t} + n \) on a \( \vec{c} \)-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 \( \{(p_1 + n, c_1), \ldots, (p_k + n, c_k)\} \).
Example 4. The example illustrates our type and effect system by giving the derivation for Example 1 as follows (focusing on the \( \vec{l} \) - and \( S \)-part, only):

\[
\begin{array}{c}
0^\downarrow \vdash [[:\text{spawn}(e_1)]]: \{1, \{2,3\}\} \\
2^\downarrow \vdash [[:\text{spawn}(e_2)]]: \{1\}; e_1; e_2: \{10, \{1,0\}\} \\
0^\downarrow \vdash [[:\text{spawn}(e_1)]]: [[:\text{spawn}(e_1)]]: \{1\}; e_1; e_2: \{10, \{1,0\}\}
\end{array}
\]

The overall resource consumption then is \( 15 = 7 \lor (10 + 1) \lor (8 + 1) \).

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 \( \text{jc-sequences} \). Now that more than one thread is involved, that data-structure is generalized to \( \text{jc-trees} \) which are basically finitely branching, finite trees where the nodes are labeled by a transaction label and an integer.

With \( t \) as \( \text{jc-tree} \), the judgments at the global level are of the following form:

\[
\Gamma \vdash P :: t.
\]

Definition 10 (Jc-tree). Joining commit trees (or \( \text{jc-trees for short} \)) are defined as tree of type \( \text{JCTree} = \text{Node} \times \text{Nat} \times \text{Lab} \times (\text{List of JCTree}) \), with typical element \( t \). We write \( \vec{t} \) for lists of \( \text{jc-trees} \). We write also \( [] \) for the empty list, and \( \text{Node}(n, l, \vec{t}) \) for a \( \text{jc-tree} \) whose root carries the natural number \( n \) as weight and \( l \) as label, and with children \( \vec{t} \).

Definition 11 (Weight). The weight of a \( \text{jc-tree} \) is given inductively as \( |\text{Node}(n, l, \vec{t})| = n \lor \sum_{i=1}^{\vec{t}}(|l_i|) \). The initial weight of a join tree \( t \), written \( |t|_1 \), is the weight of its leaves.

Definition 12 (Parallel merge). We define the following two functions \( \otimes_1 \) of type \( \text{JCTree} \times \text{JCTree} \rightarrow \text{JCTree} \) and \( \otimes_2 \) of type \( \text{JCTree}^2 \rightarrow \text{JCTree} \) by mutual induction. In abuse of notation, we will write \( \otimes \) for both in the following.

\[
\begin{align*}
\text{Node}(n_1, l_1, f_1) \otimes_1 (\text{Node}(n_2, l_2, f_2) &:: f) = \text{Node}(n_1 + n_2, l_1 \otimes_2 f_1 \otimes_2 f_2, f) \\
\text{Node}(n_1, l_1, f_1) \otimes_1 (\text{Node}(n_2, l_2, f_2) &:: f) = \text{Node}(n_2, l_2, f_2) \cdot (\text{Node}(n_1, l_1, f_1) \otimes_1 f) \quad l_1 \neq l_2
\end{align*}
\]

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

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 13 (Lifting). The function \( \text{lift} \) of type \( \text{LEnv} \times \text{Nat}^+ \rightarrow \text{JCTree} \) is given inductively as:

\[
\begin{align*}
\text{lift}([], [\vec{n}]) &= \text{Node}(n, \bot, [], [\vec{n}]) \\
\text{lift}((l; \rho :: E), \vec{s} :: n) &= \text{Node}(n, l, [\text{lift}(E, \vec{s})])
\end{align*}
\]
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 $|\vec{s}| = |\text{lift}(E, \vec{s})|$.

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. Remember that the resource consumption is measured in terms of numbers of logs co-existing simultaneously. We start by defining the actual resource consumption of a program:

Definition 14 (Resource consumption). The weight of a local environment $E$, written $|E|$ is defined as its length, i.e., the number of its $l,p$-bindings. The weight of a global environment $\Gamma$, 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.

Lemma 2. $(S_1 \cup S_2) \ominus_n \vec{t} = S_1 \ominus_n (S_2 \ominus_n \vec{t})$.

Proof. Straightforward. □

Lemma 3. Let $S$ be a parallel weight and $n_1$ and $n_2$ two non-negative numbers.

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

Proof. By straightforward calculation. □

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 4 (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 $f(S_1, l, S_2) = S_1 \uparrow_l \cup S_2$. Then

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

Proof. By straightforward but slightly tedious calculation. □

Lemma 5 (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

$$\vec{s} \oplus p_1 (S_1 \ominus_n \vec{t} \oplus_p (S_2 \ominus_n \vec{u})) = (\vec{s} \oplus_p (S_1 \ominus_n \vec{t})) \oplus_p ((S_2 \ominus_n \vec{t}) \ominus_n \vec{u}).$$

<table>
<thead>
<tr>
<th>$E \vdash e :: n.l.\vec{s}$</th>
<th>$t = \text{lift}(E, \vec{t})$</th>
<th>$\Gamma_1 \vdash P_1 : t_1$</th>
<th>$\Gamma_2 \vdash P_2 : t_2$</th>
<th>$\Gamma_1, \Gamma_2 \vdash P_1 \parallel P_2 : t_1 \otimes t_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P : E \vdash p(e) :: t$</td>
<td>$\text{T-THREAD}$</td>
<td>$\text{T-PAR}$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5: Effect system
Proof. We are given \( s = s_0, \ldots, s_k, t = t_0, \ldots, t_m, \) and \( u = u_0, \ldots, u_q. \) Further we set

\[
\begin{align*}
l_1 &= n_1 - |s| + 1 = n_1 - k \\
l_2 &= n_2 - |t| + 1 = n_2 - m \\
l_3 &= n_3 - |u| + 1 = n_3 - q \\
p_1 &= n_2 - l_1 \\
p_2 &= n_3 - l_2
\end{align*}
\]

where the \( l_i, n_i \) and the 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 (7) \( p_1 \leq m \) 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:

\[
\begin{align*}
\bar{s} \oplus_{p_1} (S_1 \ominus n_2) (\bar{t} \oplus_{p_2} (S_2 \ominus n_3 \bar{u})) &= (8) \\
&= \bar{s} \oplus_{p_1} (S_1 \ominus n_2) (\bar{t} \oplus_{p_2} (u_0 + |S_2|, u_1 + |S_2 - n_3 - 1|, \ldots, u_q + |S_2 - n_3 - q|)) \\
&= \bar{s} \oplus_{p_1} (S_1 \ominus n_2) (\bar{t} \oplus_{p_2} \bar{u}) \\
&= \bar{s} \oplus_{p_1} (S_1 \ominus n_2) (t_0, t_1, \ldots, t_m \lor u_0 \lor u_1 \lor \ldots \lor u_{p_2}, u_{p_2 + 1}, \ldots, u_q) \\
&= \bar{s} \oplus_{p_1} (S_1 \ominus n_2) (t_0, t_1, \ldots, t_m, u_{p_2 + 1}, \ldots, u_q) \\
&= \bar{s} \oplus_{p_1} (t_0, t_1, \ldots, t_m, u_{p_2 + 1}, \ldots, u_q) \\
&= s_0, s_1 \lor t_0 \lor t_1 \lor \ldots \lor t_{p_2} \lor \ldots \lor t_{p_2 + 1} \lor \ldots \lor t_m, u_{p_2 + 1}, \ldots, u_q
\end{align*}
\]

and

\[
\begin{align*}
(s \oplus_{p_1} (S_1 \ominus n_2 \bar{t})) \oplus_{p_2} (S_2 \cup S_1 \downarrow l_2 \ominus n_3 \bar{u}) &= (9) \\
&= (s \oplus_{p_1} (t_0 + |S_1|, t_1 + |S_1 - n_2 - 1|, \ldots, t_m + |S_1 - n_2 - m|)) \oplus_{p_2} (S_2 \cup S_1 \downarrow l_2 \ominus n_3 \bar{u}) \\
&= (s \oplus_{p_1} (t_0', t_1', \ldots, t_m')) \oplus_{p_2} (S_2 \cup S_1 \downarrow l_2 \ominus n_3 \bar{u}) \\
&= (s_0, s_1 \lor t_0' \lor t_1' \lor \ldots \lor t_{p_1'} \lor \ldots \lor t_{p_1'} \lor \ldots \lor t_m', u_{p_1'} \lor \ldots \lor u_{q'}) \\
&= (s_0, s_1 \lor t_0' \lor t_1' \lor \ldots \lor t_{p_1'} \lor \ldots \lor t_{p_1'} \lor \ldots \lor t_m', u_{p_1'} \lor \ldots \lor u_{q'}) \\
&= (s_0, s_1 \lor t_0' \lor t_1' \lor \ldots \lor t_{p_1'} \lor \ldots \lor t_{p_1'} \lor \ldots \lor t_m', u_{p_1'} \lor \ldots \lor u_{q'}) \\
&= s_0, s_1 \lor t_0' \lor t_1' \lor \ldots \lor t_{p_1'} \lor \ldots \lor t_{p_1'} \lor \ldots \lor t_m', u_{p_1'} \lor \ldots \lor u_{q'}
\end{align*}
\]
In the calculation, we used the following abbreviations:

\[
\bar{u} = S_2 \otimes n_3, \quad \underline{u} = (u_0 + |S_2|, u_1 + |S_2|_{n_3-1}, \ldots, u_q + |S_2|_{n_3-q})
\]

\[
\bar{t}_m = t_m \lor u_0' \lor \cdots \lor u_{p_2}'
\]

\[
t'_m, t''_m \ldots t''''_m = (t_0 + |S_1|, t_1 + |S_1|_{n_2-1}, \ldots, t_{m-1} + |S_1|_{n_2-(m-1)})
\]

\[
u''_{p_2+1}, \ldots, u''_q = u''_{p_2+1} + S_1 \downarrow_{n_2-(m+1)}, \ldots, u''_q + S_1 \downarrow_{n_2-(q-p_2)}
\]

\[
t''_m = t_m + S_1 \downarrow_{n_2-m}
\]

\[
\bar{t}''_m = S_1 \downarrow_{n_2-m} \otimes n_3 \underline{u}
\]

\[
t''''_m = t'_m \lor u''_0 \lor \cdots \lor u''_{p_2}
\]

\[
S'_1 = S_1 \cup S_2
\]

To see that \(S_1\) and \(S_2\) are equal, we need to establish the following two equation. The required equality \(t''''_m = t'''_m\) is shown as follows:

\[
t''''_m = t''''_m = (t_m + |S_1|_{n_2-m}) \lor \cdots \lor (u''_{p_2} + |S_1|_{n_2-m})
\]

\[
= (t_m + |S_1|_{l_2}) \lor (u''_0 + |S_1|_{l_2}) \lor (u''_1 + |S_1|_{n_3-1}) \lor \cdots \lor (u''_{p_2} + |S_1|_{n_2-(q-p_2)})
\]

\[
= (t_m + |S_1|_{l_2}) \lor (u''_0 + |S_1|_{l_2}) \lor \cdots \lor (u''_{p_2} + |S_1|_{n_2-m})
\]

\[
= t''''_m \lor u''_0 \lor \cdots \lor u''_{p_2}
\]

For the application of Lemma 3, 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:

\[
u''_{p_2+1}, \ldots, u''_q = u''_{p_2+1} + |S_1|_{n_2-(m+1)}, \ldots, u''_q + |S_1|_{n_2-(m+q-p_2)}
\]

\[
= u''_{p_2+1} + |S_1|_{l_2} \downarrow_{n_2-(m+1)}, \ldots, u''_q + |S_1|_{l_2} \downarrow_{n_2-(m+q-p_2)}
\]

\[
= u''_{p_2+1} + |S_1|_{l_2} \downarrow_{l_2-1}, \ldots, u''_q + |S_1|_{l_2} \downarrow_{l_2-(q-p_2)}
\]

\[
= u''_{p_2+1} + |S_1|_{l_2} \downarrow_{l_2-(p_2+1)}, \ldots, u''_q + |S_1|_{l_2} \downarrow_{n_3-q}
\]

The remaining cases are similar. □

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 15 (Order on trees).** We define the binary relation $\leq$ on jc trees inductively as follows: $\text{Node}(n, l, \vec{s}) \leq \text{Node}(m, l, \vec{t})$ 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 6 (Lifting of ordering).** If $\vec{s} \leq \vec{t}$ (as comparison between jc-sequences), then $\text{lift}(E, \vec{s}) \leq \text{lift}(E, \vec{t})$ (as comparison between jc trees).

*Proof.* Obvious. $\square$

**Lemma 7 (Lifting and commit).** $\text{lift}(E, l::n :: \vec{u}) \geq \text{lift}(E, \vec{u})$.

*Proof.* Straightforward. $\square$

**Lemma 8 (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. $\square$

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 9 (Subject reduction (local)).** If $n \vdash e_1 :: n_2, h_1, l_1, \vec{s}, S_1$ 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 $S_2 \sqsubseteq S_1$.

*Proof.* In induction on the derivation of the local reduction steps using the rules from Table 2. The cases for field look-up, 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: $E \vdash \text{let } x : T = v \text{ in } e \rightarrow E \vdash e[v/x]$  

The assumption of well-typedness gives

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

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 $n_1 \vdash t[v/x] : n_2, h_2, l_2, \vec{s}, S$, as required.

**Case:** R-COND1: $E \vdash \text{let } x : T = (\text{if } \text{true then } e_1 \text{ else } e_2) \text{ in } e \rightarrow E \vdash \text{let } x : T = e_1 \text{ in } e$  

By well-typedness, we are given

$$
\frac{n \vdash e_1 :: n', h_1, l_1, \vec{s}, S_1 \quad n \vdash e_2 :: n', h_2, l_2, \vec{t}, S_2 \quad S = S_1 \cup S_2}{n \vdash \text{if } v \text{ then } e_1 \text{ else } e_2 :: n', h_1 \lor h_2, l_1 \land l_2, \vec{s} \lor \vec{t}, S} \quad \text{S-COND1}
$$

The case follows from the fact that $\vec{s} \leq \vec{s} \lor \vec{t}$ and that $S_1 \sqsubseteq S_1 \sqsubseteq S_2$ (cf. Definition 8 and Lemma 1). The case for R-COND2 works symmetrically.
Case: R-LET: $E \vdash \text{let } x_2 : T_2 = (\text{let } x_1 : T_1 = e_1 \text{ in } e_2) \text{ in } e_3 \rightarrow E \vdash \text{let } x_1 : T_1 = e_1 \text{ in } (\text{let } x_2 : T_2 = e_2 \text{ in } e_3)$

We are given:

\[
\begin{align*}
n_1 \vdash e_1 :: n_2, h_1, l_1, \bar{S}_1 & \quad n_2 \vdash e_2 :: n_3, h_2, l_2, \bar{S}_2 \\
n_1 \vdash \text{let } x_1 = e_1 \text{ in } e_2 :: n_3, h_1 \lor h_2, l_1 \land l_2, \bar{S}_1 \cup \bar{S}_2 & \quad n_3 \vdash e_3 :: n_4, h_3, l_3, \bar{u}, \bar{S}_3 \\
n_1 \vdash \text{let } x_2 = (\text{let } x_1 = e_1 \text{ in } e_2) \text{ in } e_3 :: n_4, (h_1 \lor h_2) \lor h_3, (l_1 \land l_2) \land l_3, \bar{w}, (S_1 \downarrow \uplus S_3) \downarrow l_1 \cup S_3
\end{align*}
\]

where $\bar{w} = (\bar{\bar{x}} \oplus p_1 (S_1 \ominus n_2 \bar{\bar{r}})) \ominus p_2 (S_2 \ominus n_3 \bar{\bar{u}})$ and we need to prove:

\[
\begin{align*}
n_1 \vdash e_1 :: n_2, h_1, l_1, \bar{S}_1 & \quad n_2 \vdash \text{let } x_2 = e_1 \text{ in } e_2 :: n_4, h_2 \lor h_3, l_1 \land l_3, \bar{w}', (S_1 \downarrow l_1 \cup (S_2 \downarrow l_1 \cup S_3))
\end{align*}
\]

where $\bar{w}' = \bar{\bar{x}} \oplus p_1 (S_1 \ominus n_2 (\bar{\bar{r}} \oplus p_2 (S_2 \ominus n_3 \bar{\bar{u}})))$. For high and low points, we use associativity of $\lor$ and $\land$. For parallel weights, we use associativity from Lemma 8. Finally, $\bar{w} = \bar{w}'$ follows from Lemma 9.

Case: R-LOOKUP, R-UPDATE, and R-NEW

Trivial, as no transactions are involved and no threads are spawned.

Case: R-CALL

Straightforward.

\[\square\]

Lemma 10 (Subject reduction).

$\Gamma \vdash P :: t$ and $\Gamma \vdash P \rightarrow \Gamma' \vdash P'$ implies $\Gamma' \vdash P' :: t'$ where $t' \leq t$.

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

Case: G-PLAIN

A consequence of subject reduction for the local level (Lemma 7), the compatibility of the orders for the sequences on the local level and the trees on the global level (Lemma 9) and fact that the reflect-function does not change the length of the local environments (cf. Definition 3).

Case: G-SPAWN

We are given $\Gamma \vdash p(\text{let } x : T = \text{spawn } e_1 \text{ in } e_2) \rightarrow \Gamma' \vdash p(\text{let } x : T = \text{null in } e_2) \parallel p'(e_1)$.

Well-typedness of the configuration before the steps gives:

\[
\begin{align*}
n_1 \vdash e_2 :: 0, h_2, 0, \bar{u}, S_2 & \quad n_1 \vdash \text{spawn } e_2 :: n_1, n_1, [n_1 + u_0], S_2 \cup \{(n_1, h_2 - n_1)\}
\end{align*}
\]

\[
\begin{align*}
n_1 \vdash \text{let } x = \text{spawn } e_2 \text{ in } e_1 :: 0, h_1, 0, \bar{x}, S & \quad S = S_1 \cup S_2 \downarrow 0
\end{align*}
\]

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

\[
\begin{align*}
n_1 \vdash \text{null} :: n_1, n_1, [n_1], \emptyset & \quad n_1 \vdash n_2, h_1, 0, \bar{v}, S_1 & \quad n_1 \vdash e_2 :: 0, h_2, 0, \bar{u}, S_2
\end{align*}
\]

\[
\begin{align*}
p_1 : E \vdash p_1(\text{let } x = \text{null in } e_1) :: \text{lift}(E, \bar{v}) & \quad p_2 : E \vdash p_2(e_2) :: \text{lift}(E, \bar{u})
\end{align*}
\]

\[
\begin{align*}
p_1 : E, p_2 : E \vdash p_1(\text{let } x = \text{null in } e_1) \parallel p_2(e_2) :: \text{lift}(E, \bar{v}) \otimes \text{lift}(E, \bar{u})
\end{align*}
\]
where \( S'_i = S_2 \cup \{(n_1, h_2 - n_1)\} \). We need to prove that \( \text{lift}(E, \mathcal{S}) = \text{lift}(E, \mathcal{V}) \otimes \text{lift}(E, \mathcal{U}) \). The proof of this equation follows straightforwardly from Definition 12 of \( \otimes \). Note that the two trees are both linear and their nodes are labeled by the same labels (cf. the definition of the \( \text{lift} \)-function).

Case: G-TRANS
We are given \( p:E \vdash p(\text{let } x = \text{onacid in } e) \Rightarrow p:E' \vdash p(\text{let } x = \text{null in } e) \). Well-typedness of the configuration before the step gives:

\[
\begin{align*}
n_1 \vdash \text{onacid} : n_1 + 1, n_1 + 1, n_1 \vdash n_1 + 1 \vdash e : 0, h, 0, \mathcal{S} \\
n_1 \vdash x = \text{onacid} : 0, h, 0, \mathcal{S} \\
p:E \vdash p(\text{let } x = \text{onacid} : 0, h, 0, \mathcal{S})
\end{align*}
\]

Note that the \( \text{start}(\_\_\_\_\_\_\_\_\_) \)-function used in the G-TRANS-step to update the local environment assures that \( |E'| = |E| + 1 \) (cf. Definition 13). Note further that in the application of rule G-LET, we know that \( n + 1 \geq 0 \), and thus \( n + 1 \geq s_0 \) equals to \( s_0 \) for \( s_0 \). For the configuration after the step, we can derive with T-THREAD, T-LET, and T-NULL:

\[
\begin{align*}
n_1 + 1 \vdash \text{null} : n_1 + 1, n_1 + 1, n_1 + 1, n_1 + 1, \vdash n_1 + 1 \vdash e : 0, h, 0, \mathcal{S} \\
n_1 + 1 \vdash \text{let } x = \text{null in } e : 0, h, 0, \mathcal{S} \\
p:E' \vdash p(\text{let } x = \text{null in } e) : \text{lift}(E, \mathcal{S})
\end{align*}
\]

Case: G-COMM
We are given \( \Gamma \vdash \ldots \vdash p_i(\text{let } x = \text{commit in } e_i) \Rightarrow \ldots \Rightarrow \Gamma' \vdash \ldots \vdash p_i(\text{let } x = \text{null in } e_i) \). Well-typedness of the configuration before the step gives for each \( p_i \):

\[
\begin{align*}
n_i \vdash \text{commit} : n_i - 1, n_i - 1, n_i - 1, n_i - 1, \vdash n_i - 1 \vdash e_i : 0, h, 0, \mathcal{U}_i, \mathcal{S} \\
n_i \vdash \text{let } x = \text{commit in } e_i : 0, h, 0, \mathcal{U}_i, \mathcal{S} \\
p_i:E_i \vdash p_i(\text{let } x = \text{commit in } e_i) : \text{lift}(E_i, \mathcal{U}_i, \mathcal{S})
\end{align*}
\]

Note that \( n_i - 1 \geq u_0 \) since \( u_0 \geq n_i - 1 \) and \( |\mathcal{U}_i| = n_i \) because all the onacids are committed at the end (cf. T-THREAD). By T-THREAD, T-LET, and T-NULL we can derive:

\[
\begin{align*}
n_i - 1 \vdash \text{null} : n_i - 1, n_i - 1, n_i - 1, n_i - 1, \vdash n_i - 1 \vdash e_i : 0, h, 0, \mathcal{U}_i, \mathcal{S} \\
n_i - 1 \vdash \text{let } x = \text{null in } e_i : 0, h, 0, \mathcal{U}_i, \mathcal{S} \\
p_i:E_i' \vdash p_i(\text{let } x = \text{null in } e_i) : \text{lift}(E_i', \mathcal{U}_i, \mathcal{S})
\end{align*}
\]

where \( E_i = E_i', I : p \), i.e., \( |E_i'| = |E_i| - 1 \) and (cf. Definition 3 and rule G-COMM). By Lemma 7 \( (\text{lift}(E_i', \mathcal{U}_i)) \leq (\text{lift}(E_i, \mathcal{U}_i)) \), and therefore by monotonicity from Lemma 8 \( \otimes_i(\text{lift}(E_i', \mathcal{U}_i)) \leq (\otimes_i(\text{lift}(E_i, \mathcal{U}_i))) \), as required.

Case: G-COMM-ERROR
Omitted, since the formulation of subject reduction covers only non-erroneous states. A type and effect system which prevents statically that such erroneous steps ("commit errors") occur has been formalized in [19].

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

Lemma 11. \( |t_1 \otimes t_2| = |t_1| + |t_2| \)
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 over-approximated 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 12.** 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

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

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 [11] and the fact that $|\Gamma_1, \Gamma_2| = |\Gamma_1| + |\Gamma_2|$. \hfill \Box

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 $\Gamma_0 \vdash p_0(e_0)$ and $\Gamma_0 \vdash p_0(e_0) :: t$ (with $\Gamma_0$ as empty global context). If $\Gamma_0 \vdash p_0(e_0) \Rightarrow^* \Gamma \vdash P$, then $|\Gamma| \leq |t|$.

**Proof.** An immediate consequence of subject reduction (Lemma [10] and Lemma [12]) \hfill \Box

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. Instead, synchronization is affected by the nesting structure of the multi-threaded transactions, i.e., 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 such a concurrency model. Abstractive 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 [17], originally for a lazy, stream-based function language, resp. in [16] for a strict functional language, both first-order. The corresponding static type systems with space effects guarantee that well-typed programs use at most the space specified by the programmers. Sized types have been used further in [6] and [7], [10] treat execution time as resource. 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 [14] 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 [13] and [12]. [24] 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. [8] 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 [15], 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 the 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 a 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 [?]}. For low-level languages, [4] uses program logics to infer precise memory consumption of sequential byte-code programs with resource annotation by pre- and post-conditions. 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 [9] to collect objects that do not escape their scopes. The byte-code 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.* [20] propose a fast algorithm with small memory footprint to statically calculate heap memory for a class of JavaCard programs. 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 [23], 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 [25], which presents an analysis which is not compositional. 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 [21]) 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). That would involve to refine the rules which access the memory by reading and writing from fields in that they have a non-trivial effect on the memory consumption; currently their effect is ignored. Refining the rules in that way should largely be orthogonal, except that in particular the effect of commit will then not just decrease the resource consumption by 1 as now, but needs to calculate that all the memory for the committed transaction is deallocated. Due to the nested nature of the transactions, that requires a stack-structured memory estimation as pre- and post-conditions instead of single numbers as now. Furthermore, a challenging step is to automatically infer interface information concerning the resource consumption for method declarations. 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 based on resource consumption.

**References**


Index

\[ \Gamma \vdash P \implies \Gamma' \vdash P' \text{ (global transition), 11} \]
\[ S = \{ (p_0, c_0), \ldots, (p_k, c_k) \} \text{ (parallel weight), 14} \]
\[ E \vdash e \to E' \vdash e' \text{ (local transition), 10} \]
\[ \Gamma \text{ (global environment), 11} \]
\[ \Gamma \vdash P \text{ (global configuration), 11} \]
\[ \Gamma \vdash e : T \text{ (typing judgment), 13} \]
\[ \text{par}(S, l), 15 \]
\[ S \downarrow l, 14 \]
\[ \text{seq}(S, l), 15 \]
\[ |S|, 14 \]
\[ |S| \text{ (resource consumption of } S\text{), 14} \]
\[ \oplus_p, 15 \]
\[ \otimes_s, 15 \]
\[ \text{lft} \text{ (lifting), 15} \]
\[ |t|_t \text{ (initial weight of } t\text{), 15} \]
\[ |t| \text{ (weight of } t\text{), 18} \]
\[ p(e) \text{ (thread), 8} \]
\[ \rho \text{ (log), 10} \]
\[ n_1 \vdash e :: n_2, h, l, i, S \text{ (effect judgment), 13} \]
\[ t_1 \otimes t_2 \text{ (parallel composition), 18} \]
\[ \text{balance}, 13 \]
\[ \text{class definition}, 8 \]
\[ \text{compositionality}, 5 \]
\[ \text{configuration} \]
\[ - \text{ global}, 11 \]
\[ - \text{ constructor}, 8 \]
\[ E \text{ (local environment), 10} \]
\[ \text{environment} \]
\[ - \text{ global}, 11 \]
\[ - \text{ local}, 10 \]
\[ \text{extend}, 10 \]
\[ \text{field}, 8 \]
\[ \text{fv}(e) \text{ (free variables), 9} \]
\[ \text{global transition}, 11 \]
\[ \text{implicit synchronization}, 7 \]
\[ \text{intranse}, 12 \]
\[ \text{jc-sequence}, 14 \]
\[ - \text{ composition}, 15 \]
\[ \text{jc-tree} \]
\[ - \text{ parallel composition}, 18 \]
\[ \text{join}, 16 \]
\[ \text{joining commit}, 5 \]
\[ \text{joining commit sequence}, 14 \]
\[ \text{joining commit sequence}, 14 \]
\[ \text{judgment}, 13 \]
\[ l \text{ (transaction label), 10} \]
\[ \text{lifting}, 18 \]
\[ \text{local environment}, 10 \]
\[ \text{local transition}, 10 \]
\[ \text{method definition}, 8 \]
\[ \text{mtype}, 18 \]
\[ \text{nested}, 13 \]
\[ \text{onacid}, 12 \]
\[ \text{overloading}, 9 \]
\[ \text{parallel weight}, 14 \]
\[ - \text{ associativity}, 19 \]
\[ \text{read}, 10 \]
\[ \text{resource consumption}, 19 \]
\[ \text{sequential composition}, 7 \]
\[ \text{sequential composition and let-construct}, 9 \]
\[ \text{subject reduction}, 22, 23 \]
\[ \text{TFJ}, 8 \]
\[ \text{transaction} \]
\[ - \text{ multi-threaded}, 5 \]
\[ - \text{ nested}, 5 \]
\[ \text{transaction label}, 10 \]
\[ \text{transaction labels}, 13 \]
\[ \text{Transactional Featherweight Java}, 8 \]
\[ \text{weight}, 18 \]
\[ \text{write}, 10 \]