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\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 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 perform 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) \cite{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.

```plaintext
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 e1) 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 e1, starts its own transactions (nested inside the surrounding ones), e.g., if e1 = [ : [ : [ : ] ; ] ; ], then these three commits are no joining commits.

□
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). Abstraction 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 as $e_1; e_2$ 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:

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

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 1 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
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 calljc-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 \( p(e) \) 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 \( C \{ f : T; K; M \} \) consists of a name \( C \), a list of fields \( f \) with corresponding type declarations \( T \) (assuming that all \( f \)’s are different), a constructor \( K \), and a list \( M \) of method definitions. A constructor \( C(f : T) \{ \text{this} : f \} \) of the corresponding class \( C \) initializes the fields of instances of that class, these fields are mentioned as the formal parameters of the constructor.

\[
\begin{align*}
P &::= 0 | P \parallel P | P \mid p(e) & \text{processes/threads} \\
L &::= \text{class } C \{ f : T; K; M \} & \text{class definitions} \\
K &::= C(f : T) \{ \text{this} : f \} & \text{constructors} \\
M &::= m(x : T) \{ e \} & \text{methods} \\
e &::= v | v.f | v.f := v \mid \text{if } v \text{ then } e \text{ else } e \\
&\quad | \text{let } x : T = e \text{ in } e \mid v.m(v) & \text{expressions} \\
&\quad | \text{new } C(v) \mid \text{spawn } e \mid \text{onacid} \mid \text{commit} & \\
v &::= r | x | \text{null} & \text{values}
\end{align*}
\]

Table 1: Abstract syntax
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 \texttt{null}. The expressions \( v.f \) and \( v_1.f := v_2 \) represent field access and field update respectively. Method calls are written \( v.m(\vec{i}) \) and object instantiation is \texttt{new} \( C(\vec{i}) \). The next two expressions deal with the basic, sequential control structures: \texttt{if} \( v \) then \( e_1 \) else \( e_2 \) represents conditions, and the let-construct \( \texttt{let} \ x : T = e_1 \text{ in } e_2 \) 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 \( \texttt{let} \ x : T = e_1 \text{ in } e_2 \) where the variable \( x \) does not occur free in \( e_2 \). The let-construct, as usual, binds \( x \) in \( e_2 \). We write \( f v(e) \) for the free variables of \( e \), defined in the standard way. The language is multi-threaded: \texttt{spawn} \( e \) starts a new thread of activity which evaluates \( e \) in parallel with the spawning thread. Specific for TFJ are the two constructs \texttt{onacid} and \texttt{commit}, two dual operations dealing with transactions. The expression \texttt{onacid} starts a new transaction and executing \texttt{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 II 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 \( \texttt{let} \ x_1 = e_1 \text{ in } (\texttt{let} \ x_2 = e_2 \text{ in } x_1.f := x_2) \), making the evaluation order explicit. The transformation from the general syntax to the one of Table II 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

$$E \vdash e \rightarrow E' \vdash e'$$

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 $l_1;p_1,\ldots,l_i;p_i$, i.e., of pairs of transaction labels $l_i$ and corresponding log $p_i$. 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 $p_i$ can be thought of as “local copies” of the heap. The log $p_i$ 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 $\text{reflect}(p, E, \Gamma) = \Gamma'$ and $E = \pi_1E_1,\ldots,\pi_kE_k$, then $\Gamma' = \pi_1E_1',\ldots,\pi_kE_k'$ with $|E_1| = |E_1'|$ (for all $i$).
2. The function spawn satisfies the following condition: Assume $\Gamma = t : E, \Gamma''$ and $p' \notin \Gamma$ and $\text{spawn}(p, p', \Gamma) = \Gamma''$, then $\Gamma'' = \piE''$ s.t. $|E| = |E'|$.
3. The function start satisfies the following condition: if $\text{start}(l, p_1, \Gamma) = \Gamma'$ for a $\Gamma = \pi_1E_1,\ldots,\pi_kE_k$ and for a fresh $l$, then $\Gamma' = \pi_1E_1,\ldots,\pi_kE_k'$, with $|E'| = |E| + 1$.
4. The function intranse satisfies the following condition: Assume $\Gamma = \Gamma''$, $p:E$ s.t. $E = E', l;p$ and $\text{intranse}(l, \Gamma') = \tilde{p}$, then
   (a) $p \in \tilde{p}$ and
   (b) for all $p_i \in \tilde{p}$ we have $\Gamma = \ldots,p_i : (E_i,l;p_i),\ldots$
   (c) for all threads $p'$ with $p' \notin \tilde{p}$ and where $\Gamma'' = \ldots,t'(E',l';p'),\ldots$, we have $l' \neq l$.
5. The function commit satisfies the following condition: if $\text{commit}(\tilde{p}, E, \Gamma) = \Gamma'$ for a $\Gamma = \Gamma''$, $p:E, l:p$ and for a $\tilde{p} = \text{intranse}(l, \Gamma)$ then $\Gamma' = \ldots,p_iE_i',\ldots,p_jE_j$, where $p_i \in \tilde{p}, p_j \notin \tilde{p}, p_jE_j \in \Gamma$, with $|E_j'| = |E_j|$ and $|E_i'| = |E_i| - 1$. 

9
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 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 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 fields\((C)\) looks up fields of class \(C\) and 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 \(C(\nu\vec{1})\), which has all fields initialized to \(\nu\vec{1}\). 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:

\[ \Gamma \vdash P, \]

(2)

where \(P\) is a program and \(\Gamma\) 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 \( p(e) \), we need one \( E \) for each
thread \( p \). This means, \( \Gamma \) 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 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, those functions are left abstract (cf. Defini-
tion [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 \( \text{intrans}e(l, \Gamma) \) 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.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Semantics (local)</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-RED</td>
<td>( E \vdash \text{let } x:T = v \in e \implies E \vdash e[v/x] )</td>
</tr>
<tr>
<td>R-LET</td>
<td>( E \vdash \text{let } x_1:T_1 = (\text{let } x_1:T_1 = e_1 \in e) \implies E \vdash \text{let } x_1:T_1 = e_1 \in e )</td>
</tr>
<tr>
<td>R-COND</td>
<td>( E \vdash \text{let } x:T = (\text{if } \text{true then } e_1 \text{ else } e_2) \in e \implies E \vdash \text{let } x:T = e_1 \in e )</td>
</tr>
<tr>
<td>R-LOOKUP</td>
<td>( E \vdash \text{let } x:T = \text{read},(r,E) \implies E \vdash \text{let } x:T = \text{read}(r,E) )</td>
</tr>
<tr>
<td>R-UPD</td>
<td>( E \vdash \text{let } x:T = r.f_1 \in e \implies E' \vdash \text{let } x:T = u_i \in e )</td>
</tr>
<tr>
<td>R-CALL</td>
<td>( E \vdash \text{let } x:T = r.m,(\overline{f}) \in e' \implies E' \vdash \text{let } x:T = e' )</td>
</tr>
<tr>
<td>R-NEW</td>
<td>( E \vdash \text{let } x:T = \text{new } C \in e \implies E' \vdash \text{let } x = r \in e )</td>
</tr>
</tbody>
</table>

Table 2: Semantics (local)
\[ E \vdash e \leadsto E' \vdash e' \quad \Gamma \vdash p : E \quad \text{reflect}(p, E', \Gamma) = \Gamma' \]

\[ \Gamma \vdash \mathcal{P} \parallel p(e) \implies \Gamma' \vdash \mathcal{P} \parallel p(e') \]

\[ p' \text{ fresh} \quad \text{spawn}(p, p', \Gamma) = \Gamma' \]

\[ \Gamma \vdash \mathcal{P} \parallel p \langle \text{let } x : T = \text{onacid in } \mathcal{E} \rangle \quad \Gamma' \vdash \mathcal{P} \parallel p \langle \text{let } x : T = \text{null in } \mathcal{E} \rangle \quad \Gamma = \Gamma' = \Gamma'' \]

\[ E = E' \vdash \mathcal{P} \parallel p : E \quad \text{intranse}(l, \mathcal{E}) = \mathcal{E} \vdash \mathcal{P} \parallel p : E \quad \text{commit}(\mathcal{E}, \mathcal{P}) \]

\[ \Gamma \vdash \mathcal{P} \parallel \ldots \parallel p_i \langle \text{let } x : T_i = \text{commit in } \mathcal{E}_i \rangle \ldots \implies \Gamma' \vdash \mathcal{P} \parallel \ldots \parallel p_i \langle \text{let } x : T_i = \text{null in } \mathcal{E}_i \rangle \ldots \]

\[ \Gamma = \Gamma'' \quad \mathcal{P} \parallel E = \emptyset \]

\[ \Gamma \vdash \mathcal{P} \parallel p \langle \text{let } x : T = \text{commit in } \mathcal{E} \rangle \implies \text{error} \]

Table 3: Semantics (global)

**Definition 4.** Let TrName be the type of transaction labels. Given a local environment \( E \), the function \( l : (LEnv \rightarrow \text{List of TrName}) \) is defined inductively as follows:

\[ l(\varepsilon) = \varepsilon \] \quad \text{and} \quad \( l(p :: l(E)) = \mathcal{E} \)

Overloading the definition, we lift the function straightforwardly to global environments (with type \( l : \text{TName} \times \mathcal{GEnv} \rightarrow \text{List of TrName} \)), s.t.

\[ l(p :: l(E), \mathcal{G}) = l(E) \]

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 \( \text{nested} : \text{TrName} \times \mathcal{GEnv} \rightarrow \text{List of TName} \) 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 \( \text{Void} \) for typing side-effect-only expressions. The corresponding underlying type system for judgments of the form \( \Gamma \vdash e : T \) (“under type assumptions \( \Gamma \), 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:

$$n_1 \vdash e :: n_2, h, l, \vec{f}, S$$

(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 $\vec{f}$ and $S$ take their contribution into account. The $\vec{f}$ 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 $\vec{f}$ a joining-commit sequence, or jc-sequence for short. In Example 3, the right-hand expression $e_r = \text{[spawn } e_2 \text{]} e_3 e_4$ has one joining commit and the jc-sequence $\vec{f} = 10, 7$.

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 $\vec{f}$ 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 $\vec{f}$ 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:

$$par(S, l) = \{(p, c) \in S | p \leq l\}$$

$$seq(S, l) = \{(p, c) \in S | p > l\}$$

$$[S]_l = \{(l, 0), (l, 0), \ldots\}$$

$$S |_l = par(S, l) \cup [seq(S, l)]_l$$

(5)
Furthermore, the overall parallel weight of $S$ is defined as $|S| = \sum (p_i + c_i)$.

**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:

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

(6)

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

$$S \ominus_n \vec{t} = t'_0, t'_1, \ldots, t'_m$$

(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 $\text{par}(S_1, l_2)$ of $S_1$ not affected by a commit in $e_2$ and thus able to run in parallel with $e_2$.
2. The part $\text{seq}(S_1, l_2)$ 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 parallel weight $\text{par}(S_1, l_2)$ 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 $e_2$. 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 $[\text{seq}(S_1, l_2)]_{l_2}$ 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 $\ominus_{\downarrow}$ operation of Definition 7. 2) Secondly, joining commits of $e_2$ may no longer be joining commits of the composed expression $\text{let } x = e_1 \text{ in } e_2$. For instance, in Example 3 the (only) joining commit of $e_3$ (the one separating $e_3$ from $e_4$) is no longer a joining commit of $e_1; e_2$, as it cannot synchronize with anything outside the composed expression. The calculation of the composed jc-sequence from the constituent ones as $\vec{s} \oplus_{n_2 \rightarrow l_1} \vec{t}$ “merges” an appropriate number of elements from $\vec{t}$ (using $\lor$) depending on how many joining commits disappear in the
mal balance is given as least upper bound and dually the minimal balance as greatest lower bound, which makes the compositional treatment of the sequential composition complicated.

This number \( p \) is given by \( n_2 - l_1 \). So in rule T-Let, the overall \( \bar{u} \) is given as \( s \oplus_p (S_1 \otimes_{n_2} \bar{f}) \).

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 \( \text{let } x = e_1 \text{ in } e_2 \), 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.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>T-Var</td>
<td>( n \vdash x :: n, n, n, [n], \emptyset )</td>
</tr>
<tr>
<td>T-Null</td>
<td>( n \vdash \text{null} :: n, n, n, [n], \emptyset )</td>
</tr>
<tr>
<td>T-Lookup</td>
<td>( n \vdash e f :: n, n, n, [n], \emptyset )</td>
</tr>
<tr>
<td>T-Upd</td>
<td>( n \vdash v_1 := v_2 :: n, n, n, [n], \emptyset )</td>
</tr>
<tr>
<td>T-New</td>
<td>( n \vdash \text{new } C :: n, n, n, [n], \emptyset )</td>
</tr>
<tr>
<td>T-Onacid</td>
<td>( n \vdash \text{onacid} :: n + 1, n + 1, n, [n + 1], \emptyset )</td>
</tr>
<tr>
<td>T-Commit</td>
<td>( n \vdash \text{commit} :: n - 1, n, n - 1, [n; n - 1], \emptyset )</td>
</tr>
<tr>
<td>T-Let</td>
<td>( n \vdash \text{let } x \mapsto e :: n_1, n, l, l, \bar{u}, S )</td>
</tr>
<tr>
<td>T-Cond</td>
<td>( n \vdash \text{if } v \text{ then } e_1 \text{ else } e_2 :: n', n_1 \lor h_1, l_1, \bar{f} \lor \bar{f}, S )</td>
</tr>
<tr>
<td>T-Spawn</td>
<td>( n \vdash \text{spawn } e :: n_1, n_2, n_3, [n_1 + n_3], S \cup {(n_1, h - n_1)} )</td>
</tr>
<tr>
<td>T-Call</td>
<td>( n \vdash v \mapsto v(i) :: n_2, l, \bar{f}, S )</td>
</tr>
</tbody>
</table>

Table 4: Effect system
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 \( \vec{s} \cup \vec{t} \), 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 \( v.m(\vec{t}) \) (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):

\[
0 \vdash [ [ : \text{spawn} (e_1 ; ) ] ] : [?],\{(2,3)\} \\
2 \vdash [ [ : \text{spawn} (e_2 ; ) ] ] : [?]; e_1 : e_4 :: [0;8],[\{(1,0)\}]
\]

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

**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:

\[
\Gamma \vdash P :: t \quad (8)
\]

**Definition 8 (Jc-tree).** Joining commit trees (or \( jc \)-trees for short) are defined as tree of type \( JCtree = \text{Node} \times \text{Nat} \times \text{Lab} \times (\text{List of JCtree}) \), with typical element \( t \). We write \( \vec{t} \) for lists of \( jc \)-trees. We write also \([]\) for the empty list, and \( \text{Node}(n,l,\vec{t}) \) 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 \( |\text{Node}(n,l,\vec{t})| = n \lor \sum_{i=1}^{\vec{t}} |\{t_i\}| \). The initial weight of a join tree \( t \), written \( |t|_1 \), is the weight of its leaves.
Table 5: Effect system

| E |-n.h.l.s, t |-lift(E,s) | p:E |-p(a) |-t |
|-------------------|-------------------|-----------------|
| p:E |-p(a) |-t |-T-THREAD | F |-P |-t |-T-PAR |
| F |-P |-t |-F |-P |-t |-t |-T-PAR |

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

\[
\begin{align*}
\otimes_1 \; & \equiv \; [t] \\
\text{Node}(l, n_1, f_1) \otimes_1 \text{Node}(l, n_1, f_2) & \equiv (\text{Node}(l, n_1 + n_2, f_1 \otimes_2 f_2)) \circ f \\
\text{Node}(l_1, n_1, f_1) \otimes_1 \text{Node}(l_2, n_2, f_2) & \equiv (\text{Node}(l_1, n_1, f_1)) \circ (\text{Node}(l_2, n_2, f_2)) \\
\text{Node}(l, n, f_1) \otimes_1 \text{Node}(l, n, f_2) & \equiv (\text{Node}(l, n, f_1) \otimes_1 f) \circ f_2 \\
\otimes_2 \; & \equiv \; [t] \\
\begin{align*}
\text{Node}(l, n, f) & \equiv \text{Node}(\bot, n, [f]) \\
\text{Node}(l, \rho :: E, \bar{s} :: n) & \equiv \text{Node}(l, n, \text{lift}(E, \bar{s}))
\end{align*}
\]

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 11 (Lifting).** The function $\text{lift}$ of type LEnv $\times$ Nat $\rightarrow$ JCTree is given inductively as follows.

\[
\begin{align*}
\text{lift}([], n) & \equiv \text{Node}(\bot, n, []) \\
\text{lift}(l; \rho :: E, \bar{s} :: n) & \equiv \text{Node}(l, n, \text{lift}(E, \bar{s}))
\end{align*}
\]

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

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 environment $E$, written $|E|$ is defined as its length, i.e., the number of its $l; \rho$-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, which are later needed in the inductive proof of subject reduction.
Lemma 1. \((S_1 \cup S_2) \otimes_n \vec{t} = S_1 \otimes_n (S_2 \otimes_n \vec{t})\).

Proof. Straightforward. \(\Box\)

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. \(\Box\)

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 \(f(S_1,1,S_2) = S_1 \downarrow_1 \cup S_2\). 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. \(\Box\)

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

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

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

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

\[
\begin{align*}
\vec{s} \oplus_{p_1} (S_1 \otimes_{n_2} (\vec{t} \oplus_{p_2} (S_2 \otimes_{n_3} \vec{u}))) &= \\
\vec{s} \oplus_{p_1} (S_1 \otimes_{n_2} (\vec{t} \oplus_{p_2} (u_0 + |S_2|,u_1 + |S_2|,\ldots,u_q + |S_2|,\ldots,\downarrow_{n_3-q}))) &= \\
\vec{s} \oplus_{p_1} (S_1 \otimes_{n_2} (\vec{t} \oplus_{p_2} \vec{u})) &= \\
\vec{s} \oplus_{p_1} (S_1 \otimes_{n_2} (t_0,t_1,\ldots,t_m \lor u_0' \lor u_1' \lor \ldots \lor u_{p_2}'+u_{p_2+1}'+\ldots,u_q')) &= \\
\vec{s} \oplus_{p_1} (S_1 \otimes_{n_2} (t_0,t_1,\ldots,t_m,u_{p_2+1}',\ldots,u_q')) &= \\
\vec{s} \oplus_{p_1} (t_0 + |S_1|,t_1 + |S_1|,\ldots,t_m + |S_1|,\downarrow_{n_2-m},u_{p_2+1}'+S_1 \downarrow_{n_2-(m+1)} \ldots,u_q + S_1 \downarrow_{n_2-(m+q-p_2)}) &= \\
\vec{s} \oplus_{p_1} (t_0',t_1',\ldots,t_m',u_{p_2+1}',\ldots,u_q') &= \\
\vec{s} \oplus_{p_1} (t_0',t_1',\ldots,t_m',u_{p_2+1}',\ldots,u_q') &= \\
\vec{s} \oplus_{p_1} (t_0',t_1',\ldots,t_m',u_{p_2+1}',\ldots,u_q') &= \\
s_0,\ldots,s_k \lor t_0' \lor t_1' \lor \ldots \lor t_{m'}' \lor u_{p_2+1}' \lor \ldots \lor u_q' &=
\end{align*}
\]
and

\[ (\tilde{S} \oplus p_1 (S_1 \uplus n_2, \tilde{t})) \oplus p_2 (S_2 \uplus S_1 \downarrow t) \cap \oplus n_3, \tilde{u}) = \]
\[ (\tilde{S} \oplus p_1 (t_0 + [S_1], t_1 + [S_1 \downarrow n_2 - 1], \ldots, t_m + [S_1 \downarrow n_2 - m])) \oplus p_2 (S_2 \uplus S_1 \downarrow t \cap \oplus n_3, \tilde{u}) = \]
\[ (\tilde{S} \oplus p_1 (t'_0, t'_1, \ldots, t''_m)) \oplus p_2 (S_2 \uplus S_1 \downarrow t \cap \oplus n_3, \tilde{u}) = \]
\[ (s_0, \ldots, s_{k-1}k \uplus s_0 \uplus t_0' \uplus t'_1 \uplus \ldots \uplus t''_m) \oplus p_2 (S_2 \uplus S_1 \downarrow t \cap \oplus n_3, \tilde{u}) = \]
\[ (s_0, \ldots, s_{k-1}k \uplus s_0 \uplus t_0' \uplus t'_1 \uplus \ldots \uplus t''_m) \oplus p_2 (S_1 \downarrow t \cap \oplus n_3, \tilde{u}) = \]
\[ (s_0, \ldots, s_{k-1}k \uplus s_0 \uplus t_0' \uplus t'_1 \uplus \ldots \uplus t''_m) \oplus p_2 (S_1 \downarrow t \cap \oplus n_3, \tilde{u}) = \]
\[ \ldots \]
\[ s_0, \ldots, s_{k-1}k \uplus t_0' \uplus \ldots \uplus t''_m \uplus u_0'' \uplus \ldots \uplus u_p'' \uplus u_{p+1}'' \uplus \ldots, u_q'' = \]
\[ s_0, \ldots, s_{k-1}k \uplus t_0' \uplus \ldots \uplus t''_m \uplus u_0'' \uplus \ldots \uplus u_p'' \uplus u_{p+1}'' \uplus \ldots, u_q'' = \]
\[ (11) \]

In the calculation, we used the following abbreviations:

\[ \tilde{u}' = S_2 \uplus n_3, \tilde{u} = (u_0 + |S_2|, u_1 + |S_2 \uplus n_3 - 1|, \ldots, u_q + |S_2 \uplus n_3 - q|) \]
\[ \tilde{t}_m = t_m \uplus u_0' \uplus u_1' \uplus \ldots \uplus u_p' \]
\[ t'_0, t'_1, \ldots, t'_{m-1} = (t_0 + |S_1|, t_1 + |S_1 \uplus n_2 - 1|, \ldots, t_m + |S_1 \uplus n_2 - (m-1)|) \]
\[ t''_m = t_m + |S_1 \uplus n_2 - m| \]
\[ u''_{p+1}, \ldots, u''_q = u'_{p+1} + S_1 \uplus n_2 - (m+1), \ldots, u'_{q} + S_1 \uplus n_2 - (m+q-p) \]

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

\[ t''_m = t_m + |S_1 \uplus n_2 - m| = \]
\[ (t_m \uplus u'_0 \uplus u'_1 \uplus \ldots \uplus u'_p) + |S_1 \uplus n_2 - m| \]
\[ = ((t_m \uplus |S_1 \downarrow n_2 - m|) \uplus |u'_0 \uplus |S_1 \downarrow n_2 - m|) \uplus \ldots \uplus (u'_p \uplus |S_1 \downarrow n_2 - m|) \]
\[ = (t_m + |S_1 \downarrow l_2|) \uplus |u'_0 \uplus |S_1 \downarrow l_2|\) \uplus (u'_1 + |S_1 \downarrow l_2|) \]
\[ \ldots \uplus (u'_p \uplus |S_1 \downarrow l_2|) \]
\[ = (t_m + |S_1 \downarrow l_2|) \uplus (u'_0 + |S_1 \downarrow l_2|) \uplus (u'_1 + |S_1 \downarrow l_2|) \]
\[ \ldots \uplus (u'_p + |S_1 \downarrow l_2|) \]
\[ = (t_m + |S_1 \downarrow l_2| \uplus u'' \uplus \ldots \uplus u_p' \]
\[ = t''_m \]

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 \( u''_{p+1}, \ldots, u''_q = u'_{p+1}, \ldots, u'_q \), we argue as follows:
We define the binary relation
\[ \leq \]
on trees inductively as follows: Node\((n, l, \vec{s})\) \(\leq\) 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.

**Definition 13 (Order on trees).** We define the binary relation \(\leq\) on jc trees inductively as follows: Node\((n, l, \vec{s})\) \(\leq\) 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 \(s_i \leq t_j\).

**Lemma 5 (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 6 (Lifting and commit).** \(\text{lift}(E, l; |n, \vec{u}|) \geq \text{lift}(E, \vec{u})\).

**Proof.** Straightforward. \(\square\)

**Lemma 7 (Monotonicity).** If \(t_1 \leq t_2\) and \(t_2 \leq t_3\), then \((t_1 \otimes t_2) \leq (t_1 \otimes t_3)\).

**Proof.** By straightforward calculation. \(\square\)

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:** \(E \vdash \text{let } x : T = v \in e \rightarrow E \vdash e[v/x]\)

The assumption of well-typedness gives

\[
\begin{align*}
n_1 \vdash v :: n_1, n_1, n_1, [n_1], \emptyset & \quad n_1 \vdash t :: n_2, h_2, l_2, \vec{s}, S \\
n_1 \vdash \text{let } x = v \in t :: n_2, h_2, l_2, \vec{s}, \vec{S} & \quad \text{T-LET}
\end{align*}
\]

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

\(20\)
\[\Gamma \vdash P :: t\] and \[\Gamma \vdash P \implies \Gamma' \vdash P'\] implies \[\Gamma' \vdash P' \text{ where } t' \leq t.\]

**Proof.** By induction on the derivation/derivation tree of the reduction step \(\Gamma \vdash P \implies \Gamma' \vdash P'\) by the rules of the semantics.

**Case:** G-PLAIN
A consequence of subject reduction for the local level (Lemma\[\text{[3]}\]), the compatibility of the orders for the sequences on the local level and the trees on the global level (Lemma\[\text{[4]}\]) and fact that the reflect-function does not change the length of the local environments (cf. Definition\[\text{[2]}\]).
Case: G-Spawn
We are given $\Gamma \vdash p(\text{let } x : T = \text{spawn } e_1 \in e_2) \implies \Gamma' \vdash p(\text{let } x : T = \text{null in } e_2) \parallel p'(e_1)$. Well-typedness of the configuration before the steps gives:


g_1 \vdash e_2 :: 0, h_2, 0, \bar{u}, S_2

\begin{align*}
& n_1 \vdash \text{spawn } e_2 :: n_1, n_1, n_1 + [n_1 + u_0], S_2 \cup \{(n_1, h_2 - n_1)\} \\
& n_1 \vdash e_1 :: n_2, h_1, 0, \bar{v}, S_1 & S = S_1 \cup S_2 |_{0} \\
& n_1 \vdash \text{let } x = \text{spawn } e_2 \text{ in } e_1 :: 0, h_1, 0, \bar{x}, S
\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-NUL:

\begin{align*}
& p_1 : E \vdash p_1(\text{let } x = \text{null in } e_1) :: \text{lift}(E, \bar{v}) \\
& p_2 : E \vdash p_2(e_2) :: \text{lift}(E, \bar{u}) \\
& n_1 \vdash \text{let } x = \text{null in } e_1 :: 0, h_1, 0, \bar{v}, S_1 \\
& n_1 \vdash \text{let } x = \text{null in } e_1 :: 0, h_2, 0, \bar{u}, S_2 \\
& 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 $S_2' = S_2 \cup \{(n_1, h_2 - n_1)\}$. We need to prove that $\text{lift}(E, \bar{x}) = \text{lift}(E, \bar{v}) \otimes \text{lift}(E, \bar{u})$. 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 $\text{lift}$-function).

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

\begin{align*}
& n_1 \vdash \text{onacid} :: n_1 + 1, n_1 + 1, n_1, [n_1 + 1], \emptyset \\
& n_1 \vdash e :: 0, h_1, 0, \bar{x}, S
\end{align*}

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

\begin{align*}
& n_1 + 1 \vdash \text{null} :: n_1 + 1, n_1 + 1, n_1 + 1, [n_1 + 1], \emptyset \\
& n_1 + 1 \vdash e :: 0, h_1, 0, \bar{x}, S
\end{align*}

- T-LET

\begin{align*}
& n_1 + 1 + \text{let } x \text{ null in } e :: 0, h_1, 0, (s_0 \lor s_1, s_2, \ldots), S
\end{align*}

Case: G-Comm
We are given $\Gamma \vdash \ldots p_i(\text{let } x = \text{commit in } e_i) \implies \ldots \Gamma' \vdash \ldots p_i(\text{let } x = \text{null in } e_i) \implies \ldots$. 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, n_i - 1, [n_i, n_i - 1], \emptyset \\
& n_i \vdash e_i :: 0, h_i, 0, \bar{u}_i, S
\end{align*}

\begin{align*}
& n_i \vdash \text{let } x = \text{commit in } e_i :: 0, h_i, 0, [n_i, \bar{u}_i], S \\
& p_i : E_i \vdash p_i(\text{let } x = \text{commit in } e_i) :: \text{lift}(E, [n_i, \bar{u}_i])
\end{align*}
Note that \((n_i - 1 \lor u_{0_i}) = u_{0_i}\) since \(u_{0_i} \geq n_i - 1\) and \(|\bar{u}_i| = n_i\) because all the on acids are committed at the end (cf. T-THREAD). By T-THREAD, T-LET, and T-NUL we can derive

\[
\begin{align*}
\text{Lemma 6,} & \quad (E) \quad \text{where } E \text{ are committed at the end (cf. T-T THREAD). By T-THREAD, T-LET, and T-NUL we can derive} \\
\text{Lemma 10.} & \quad |t_1 \otimes t_2| = |t_1| + |t_2| \\
\end{align*}
\]

\[
\begin{align*}
n_i - 1 & \vdash \text{null} \ldots & n_i - 1, n_i - 1 & \vdash \text{null} \ldots & n_i - 1 & \vdash e_i :: 0, h_i, 0, \bar{u}_i, S \\
\hline
n_i - 1 & \vdash \text{let } x = \text{null in } e_i :: 0, h_i, 0, \bar{u}_i, S \\
p_i; E_i' \vdash p_i(\text{let } x = \text{null in } e_i) :: \text{lift}(E_i', \bar{u}_i)
\end{align*}
\]

where \(E_i = E_i' \upharpoonright t \cdot p\), i.e., \(|E_i'| = |E_i| - 1\) and (cf. Definition \text{2} and rule G-Comm). By Lemma \text{6}, \((\text{lift}(E_i', \bar{u}_i)) \leq (\text{lift}(E_i, [n_i, \bar{u}_i]))\), and therefore by monotonicity from Lemma \text{7}, \((\otimes_i (\text{lift}(E_i', \bar{u}_i)) \leq (\otimes_i (\text{lift}(E_i, [n_i, \bar{u}_i]))\), 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 \text{[15]}. \qed

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

**Lemma 10.** \(|t_1 \otimes t_2| = |t_1| + |t_2|\)

**Proof.** Straightforward from the definition. \(\square\)

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 \(|\Gamma|\) 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 \(|\Gamma|\).

**Lemma 11.** If \(\Gamma \vdash P :: t\), then \(|\Gamma| \leq |\Gamma|\).

**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

\[
\begin{align*}
\Gamma_1 & \vdash P_1 :: t_1 \\
\Gamma_2 & \vdash P_2 :: t_2 \\
\Gamma_1, \Gamma_2 & \vdash P_1 \parallel P_2 :: t_1 \otimes t_2
\end{align*}
\]

Using induction on the two sub-goals gives \(|\Gamma_1| \leq |t_1|\) and \(|\Gamma_2| \leq |t_2|\) and the result follows by Lemma \text{10} and the fact that \(|\Gamma_1, \Gamma_2| = |\Gamma_1| + |\Gamma_2|\). \(\square\)

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 \(\Gamma_0 \vdash \pi_0(\bar{e}_0)\) and \(\Gamma_0 \vdash \pi_0(\bar{e}_0) :: t\) (with \(\Gamma_0\) as empty global context). If \(\Gamma_0 \vdash \pi_0(\bar{e}_0) \Rightarrow^* \Gamma \vdash P\), then \(\Gamma| \leq \Gamma|\).

**Proof.** An immediate consequence of subject reduction (Lemma \text{9}) and Lemma \text{11}. \(\square\)
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 been 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 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 post-
conditions. The language does not have explicit de-allocation. In \cite{Albert2009}, 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 \cite{Choi1999} to collect objects that do not escape their scopes. The bytecode language has neither explicit de-allocation nor scope. Later in \cite{Albert2007} 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. \cite{Pham2010} 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 \cite{Choi1999}, 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**


Index

\( \Gamma \vdash P \implies \Gamma' \vdash P' \) (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
\( \Gamma \) (global environment), 11
\( \Gamma' \vdash P \) (global configuration), 11
\( \Gamma' \vdash e : T \) (typing judgment), 13
\( \text{par}(S, l) \), 14
\( S \downarrow l \), 14
\( \text{seq}(S, l) \), 14
\( \| S \| \) (resource consumption of \( S \)), 14
\( \oplus_p \), 14
\( \oplus_t \), 14
\( \text{lift} \) (lifting), 17
\( |r|_1 \) (initial weight of \( r \)), 17
\( |r| \) (weight of \( r \)), 17
\( p(e) \) (thread), 7
\( \rho \) (log), 9
\( n_1 \vdash e :: n_2, h, i, l, S \) (effect judgment), 13
\( t_1 \otimes t_2 \) (parallel composition), 17

balance, 13

class definition, 7
compositionalty, 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, 4
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, 5
  - nested, 4
transaction label, 9
transaction labels, 12

Transactional Featherweight Java, 7

weight, 17
write, 10
# List of Tables

<table>
<thead>
<tr>
<th></th>
<th>Abstract syntax</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Semantics (local)</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>Semantics (global)</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>Effect system</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>Effect system</td>
<td>17</td>
</tr>
</tbody>
</table>