Safe Commits for Transactional Featherweight Java*

10. Sept. 2009

Martin Steffen and Thi Mai Thuong Tran

PMA group, Deptartment of Informatics — University of Oslo

1 Motivation

Transactions, a well-known and successful concept originating from database systems [4,1], have recently attracted interest to be incorporated directly into programming languages. They are advocated as a high-level, declarative alternative to more low-level mechanisms such as locks, monitors, etc.

A recent proposal for integrating transactional features into programming languages is *Transactional Featherweight Java* (TFJ) [2] with syntactic support for nested and multi-threaded transactions. The transactional model of TFJ is quite general, supporting *nested* transactions (a transaction can contain one or more child transactions) and *multi-threaded* transactions (one transaction can contain internal concurrency). Furthermore, the calculus allows to freely start and commit transactions (the respective constructs are called onacid and commit).

The flexibility comes at a cost: not all usages of starting and committing transactions "make sense". In particular, it is an error to perform a commit without being inside a transaction. In this paper, we introduce a static type system to prevent these errors by keeping track of starting and committing transactions, formulated as an effect system [3]. We concentrate on the effect part, as the part dealing with the ordinary types works in a standard manner and is straightforward.

2 A Type and Effect System for TFJ

The purpose of our formal system is to determine correct usage of starting and committing transactions, in particular to avoid committing when one is not inside a transaction. We call such erroneous situations *commit errors*. To prevent them, we basically keep track per thread of the number of onacids minus the number of commits in the code (which we call balance).

The general form of a judgment for a single expression (i.e., inside one thread) is of the form:

* The work has been partly supported by the EU-projects IST-33826 *Credo* (Modeling and analysis of evolutionary structures for distributed services) and FP7-231620 HATS (Highly Adaptable and Trustworthy Software using Formal Methods).

$$n_1 \vdash e :: n_2, S \tag{1}$$

where, as said, we leave aside the expression's "standard" type and concentrate on the effect. The judgment is read as "starting with a balance of n_1, evaluating e will lead to a balance of n_2." The multi-set S of integers mentioned in the post-condition takes care of the balance of new threads spawned by e.

The situation is slightly more involved, as TFJ supports nested and *multi-threaded* transactions. For instance, to commit a transaction, all threads inside must *join* to commit at the same time. To adequately take care of that multi-threading inside a transaction, the multi-set S of equation (1) is needed, which calculates the balance for potentially all threads concerned, i.e, all threads (potentially) spawned during that execution.

The following sketches 4 typical effect rules for expressions, concentrating on the aspects of transaction handling and multi-threading.

$$\frac{}{n \vdash \mathsf{onacid} :: n+1, \emptyset}\ \text{T-Onacid} \qquad \frac{}{n \vdash \mathsf{commit} :: n-1, \emptyset}\ \text{T-Commit}$$

$$\frac{n_0 \vdash e_1 :: n_1, S_1 \qquad n_1 \vdash e_2 :: n_2, S_2}{n_0 \vdash e_1; e_2 :: n_2, S_1 \cup S_2}\ \text{T-Seq} \qquad \frac{n \vdash e :: n', S}{n \vdash \mathsf{spawn}\, e :: n, S \cup \{n'\}}\ \text{T-Spawn}$$

The first basic two rules (cf. rule T-Onacid and T-Commit) are to start and commit a transaction. The dual two commands of onacid and commit simply increase, resp. decrease the balance by 1. In a sequential composition (cf. rule T-Seq), the effects are accumulated. Creating a new thread by executing spawn e does not change the balance of the executing thread (cf. rule T-Spawn). The spawned expression e in the new thread is analyzed starting with the same balance n in its pre-state.

The type and effect system is not only concerned with checking expressions, the declarations of methods are generalized, as well. We do not require that method bodies are balanced: a method may perfectly well be used to implement code for committing a transaction. To ensure, however, that this flexibility does not lead to commit errors, the declaration of a method does not only contains the expected balance of the method body, but also a requirement on where that method can be used as a form of precondition. So the *specification* of a method, as far as its effects are concerned, is of the form $m(\vec{x} : \vec{T})\{e\} : n_1 \to n_2$, and the corresponding rule looks as follows:

$$\frac{n_1 \vdash e :: n_2, \{0, \ldots, 0\}}{\vdash m(\vec{x} : \vec{T})\{e\} : n_1 \to n_2}\ \text{T-Meth}$$

The formal system must make sure that for all client code of the form $o.m(\vec{v})$, where m is the mentioned method, the call is issued only at a location, where the balance is *at least* n_1 if the method does not spawn new threads, or *exactly* n_1 if it does (corresponding to T-Meth).

3 Results

We present a type and effect system TFJ [2] concentrating on the commit-errors. Our contributions are:

Type and effect system: We present a formal derivation system following the ideas sketched above, to avoid improper use of transaction operations.

Soundness Based on the operational semantics of TFJ [2], we prove the soundness of our formal system. The proof takes the form of a standard subject reduction proof.

References

1. J. Gray and A. Reuter. *Transaction Processing. Concepts and Techniques*. Morgan Kaufmann, 1993.
2. S. Jagannathan, J. Vitek, A. Welc, and A. Hosking. A transactional object calculus. *Science of Computer Programming*, 57(2):164–186, August 2005.
3. F. Nielson, H.-R. Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.
4. G. Vossen and G. Weikum. *Fundamentals of Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2001.