Reusable Concurrent Data Types

Vincent Gramoli
EPFL

Rachid Guerraoui
EPFL

Abstract data types (ADTs) have shown to be instrumental in making sequential programs reusable. ADTs promote *extensibility* as one ADT can be specialized through inheritance by overloading or adding new methods, and *composition* as two ADTs can be combined into another ADT whose methods invoke the original ones. Unfortunately, most ADTs that export concurrent methods, often called Concurrent Data Types (CDTs), are not reusable: the programmer can hardly build upon them.

In fact, CDTs are typically optimized for exporting a fixed set of methods that are guaranteed to be "atomic", letting the programmer reason simply in terms of sequential accesses. Atomicity is, however, no longer preserved upon introduction of a new method that is susceptible of executing concurrently with existing ones. Often, the resulting semantics is far more complex than the sequential specification, making it difficult to write correct concurrent programs.

Two years ago, we reported a bug in the Java ConcurrentLinkedQueue CDT to the JSR166 group, namely the non-documented atomicity violation of its size method. In short, one cannot extend the Michael and Scott's queue [7] with a size method without modifying the original methods, a problem related to the inheritance anomaly [6]. Another bug, reported in [9, 1], outlines the difficulty of ensuring atomicity of a composite CDT: the Vector(Collection) constructor of the JDK 1.4.2 raises an ArrayIndexOutOfBoundsException.

In theory, there are solutions to address these limitations. Universal constructions [4], CASN [2], transactional memory [5, 8] could play the role of method wrappers that preserve the atomicity of series of read/write accesses under composition [3]. In addition, a programmer invoking every method of an ADT through one of these wrappers guarantees the atomicity of these methods in any possible concurrent execution. Unfortunately, any of these wrappers must accomodate the method with the strongest requirement being thus overly conservative for all other methods and limiting their concurrency. For example, the wrapper should read all elements that are present in a sorted linked list at a common point of the execution when executing a size method, yet this requirement is unnecessary when executing a contains method.

We present the Polymorphic Transaction (PT) methodology, a methodology to construct reusable CDTs. In short, PT exploits a pre-existing polymorphic set of transactional wrappers (compatible with each other) that preserve the atomicity and concurrency of methods to simplify concurrent programming.

References

[1] Cormac Flanagan, Stephen N. Freund, Marina Lifshin, and Shaz Qadeer. Types for atomicity: Static checking and inference for Java. *ACM Trans. Program. Lang. Syst.*, 30(4), 2008.

[2] Tim Harris, Keir Fraser, and Ian A. Pratt. A practical multi-word compare-and-swap operation. In *DISC*, 2002.

[3] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP*, pages 48–60, 2005.

[4] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. Program. Lang. Syst.*, 15(5), 1993.

[5] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2), 1993.

[6] Satoshi Matsuoka and Akinori Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In *Research directions in concurrent object-oriented programming*, pages 107–150. MIT Press, Cambridge, MA, USA, 1993.

[7] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, 1996.

[8] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC*, 1995.

[9] Liqiang Wang and Scott D. Stoller. Runtime analysis of atomicity for multithreaded programs. *IEEE Trans. Softw. Eng.*, 32(2), 2006.