The Joelle Programming Language [*]

Evolving Java Programs Along Two Axes of Parallel Eval

Johan Östlund
Uppsala University
johan.ostlund@it.uu.se

Stephan Brandauer
Uppsala University
stbr3377@it.uu.se

Tobias Wrigstad
Uppsala University
tobias.wrigstad@it.uu.se

Abstract

This short position paper reports on our efforts to create an object-oriented language for concurrent and parallel programs based on the active object pattern. The resulting Joelle language is explicitly designed to enable smooth reuse of existing libraries, and intends to provide an evolutionary path for incrementally transitioning entire legacy programs into the multicore age.

1. Introduction

Object-oriented programming rests on aliasing, mutable state and stable object identities. Taken alone, each member of this troika is innocuous, but in combination, and especially in a parallel setting, they can easily spell disaster. Simply by following a reference in a field, a thread of computation may end up in a completely different part of a program with different invariants that must be upheld to maintain consistency under mutation. Consequently, pointers can be argued the equivalents of the `GOTO` statements in the days of old.

Joelle builds on techniques for understanding where pointers point, reifying patterns of sharing, non-sharing, ownership transfer, etc. into compile-time checkable constructs which we believe are keys to enabling safe parallelism.

In this position paper, we describe the current state of the Joelle language (following previous work on Joëlle [4]), a parallel object-oriented programming language building on Java. A primary goal of Joelle is to provide an evolutionary path for existing Java programs and allow a program to gradually be transitioned into the multicore era by wrapping parts of a system into isolated active objects with minimal effort. With Joelle we hope to support parallel programming which is *simple*, *reliable* and *efficient*—simple enough for non-experts to construct parallel components which scale reliably with additional cores, yet are easily composable, and do not require knowledge of complex, platform-specific memory models.

2. Joelle

Active objects in Joelle build on the Creol semantics [7, 9]. They execute in parallel and communicate asynchronously by bidirectional messages returning future values. Unlike Creol objects (or most active objects or actors for that matter) active objects in Joelle may encapsulate *many* threads of control and process messages in parallel, in a deterministic way transparent to the programmer. The active object serves as a single entry point into an aggregate and all messages to objects inside an active object must go through it. This is different from Creol where an active object's innards may be exported and any interaction with exported objects automatically becomes asynchronous.

Joelle relies on deep ownership types [5] to guarantee isolation of active objects, and builds on previous work on the Joe_1 [3], Joe_3 [11] and Joline [12] languages. To enable efficient transfer of message arguments across active objects, Joelle supports externally unique references to complex object aggregates, immutable data with staged construction, and allow sharing immutable strands of otherwise mutable aggregates. Such arguments can be transmitted in constant time, and do not give rise to races, non-determinism or observational exposure [2]. For a longer treatise on these features of Joelle, see [4].

Currently, our work is focused on exploiting parallelism internal to an active object by partitioning its internal state into multiple disjoint regions. This is similar to OOFX [8] or Ownership Domains [1], but with deep ownership. Methods may be decorated with computational effects summaries over regions and the Joelle scheduler can subsequently use this information at run-time to execute messages in parallel or (we hope) improve cache utilisation.

2.1 Gradual Parallelism

Joelle is intended as a superset of Java with the goal that most valid Java code should be valid Joelle code[1]. Parallelism and opportunities for scaling with the addition of cores can be added gradually by encapsulating subsystems inside active objects, and refactoring their existing interfaces into messages on the active objects.

Joelle supports piecemeal parallelisation of a program along two orthogonal axes: by mapping smaller parts of the system into active objects, and by detailing the design of existing active objects to support parallel message processing where possible.

Naturally, naively breaking a sequential system down into a number of active objects is unlikely to automatically utilise parallel hardware to a high degree—this requires careful redesign. For now, we expect active objects be used mainly to achieve coarse grain parallelism, on the level of components or modules.

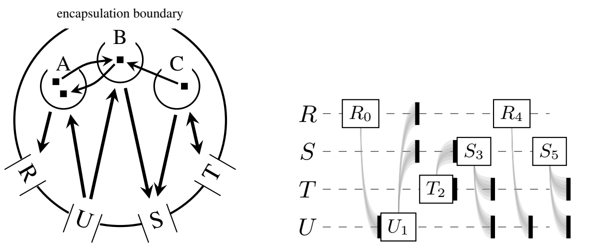
2.2 Regions and Effects

Clarke and Drossopoulou's initial work on Joe_1 [3] introduced the notion of ownership-based effects. Joe_1 cleverly extended type-based alias analysis by utilising ownership information to trivially answer aliasing questions about variables located in different parts of the heap. Put simply, objects belonging to different ownership contexts cannot be aliases. This means that the may-alias question shifts from aliasing of objects in variables to aliasing of ownership contexts in owner parameters (which is not trivial, see [10]).

Inspired by Boyland and Retert's work on OOFX [8], we extend Joelle with support for dividing objects into disjoint regions using deep ownership to avoid "leaking regions".

[*] Supported in part by the Swedish Research Council within the UPMARC Linnaeus centre of Excellence.

[1] Currently, we do not support threads, or global data—such programs must first be refactored.



(a) Active object (b) Message queues with barriers

Figure 1. (a) Active object with three regions, four methods & effects (thick). Objects inside hold cross-references (thin). (b) Example scheduling of messages R_0, U_1, T_2, S_3, R_4, S_5. Barriers in black and dependencies as phased lines.

Figure 1(a) shows an active object with three regions A, B and C and methods R, U, S, T that read from ($\downarrow$), write to ($\uparrow$) or both read and write ($\updownarrow$) these regions. In Figure 1(a), the method R reads the region A, U writes, but does not read, both A and B, S reads but does not write both B and C, and T both reads and writes C. Notably, objects in different regions may freely cross-reference. However, if evolving T introduces a use of the reference from C into B, the compiler forces this to be captured in T's signature (introducing an arrow from B to T in the figure).

2.3 Scheduling of Messages

Messages are partially ordered by a happens-before relation: ordering exists between messages that have a read-write or write-write conflict on a region. The happens-before relation guarantees that messages sent from the same thread are always processed in sending-order if they conflict. Unordered messages can run out of order or in parallel. In terms of Figure 1(a), the messages R and U are conflicting on the region A. Similarly, T messages self-conflict because T involves a read/write-access.

If the active object in Figure 1(a) received the messages R_0, U_1, T_2, S_3, R_4, S_5, in that order, the messages and dependencies would assume the shape of Figure 1(b). When a message is submitted, a barrier is added to the end of the queue of each conflicting message-class (there is one queue per asynchronous method). After a message was executed, it and its barriers are removed. Messages are safe to run iff there is no barrier in front of them. Here, R_0 and T_2 are free to run after which U_1 is scheduled. Note: T_2 conflicts with neither R_0 nor U_1.

The data structure in Figure 1(b) groups messages by their class, allowing the scheduler to execute similar messages (e.g. S_3, S_5) in succession. The possibility of improving cache hit rates by-design arises. In a parallel project [6, 13], we are exploring the use of thread-locality information to simplify cache coherence protocols. We hope to extend this work to also include Joelle.

3. Active Objects as Single Parallel Abstraction

Active objects have been claimed a natural fit to marry object-oriented programming with concurrency [14]. Threads of control destroy the conceptual view of objects collaborating by sending messages to each other by forcing a programmer to consider thread-safety and multiple interactions with a single object taking place simultaneously. Putting the object in control over when it will process a message, greatly simplifies programming, reasoning and verification [7, 9].

Message queues are a natural place to carry out simple on-line analysis of disjointness in a way transparent to a programmer. For example, two messages with effects that only concern their argument can be trivially processed in parallel if the arguments are not aliases. This allows a straightforward encoding of fork/join style tasks inside active objects. Furthermore, operations on unique references are trivially disjoint from all other state and do not even need the simple alias-analysis at run-time.

The shared-nothing model of computation performs poorly but scales well (in theory). For example, it excludes parallel in-place sorting and copying easily introduces memory bandwidth bottlenecks. By permitting mutable structures inside an active object and allowing internal parallelism where it can be determined safe, we hope to strike a good balance between share-all and share-nothing while keeping programming simple.

4. A Very Preliminary Note on Performance

Currently, we are only using a small number of well-known micro benchmarks and mockups that simulate real systems to evaluate Joelle's performance, mostly as a sanity check for our own implementation. We are continuously comparing ourselves with the likes of Erlang, Jetlang, Scala and Habanero Java. As an anecdotal data point, a port of the Erlang solution[2] of the Thread Rings benchmark to Joelle runs about 15% faster than the original program.

5. Concluding Remarks

In this short paper we have presented the current state of Joelle, which is still in its infancy, both in terms of language design and implementation. Joelle aims to smooth the transitioning of legacy Java applications into the multicore age. Our immediate future work includes supporting multiple scheduling strategies, investigating issues and scalability of real application, as well as investigating how far we can stretch the active object as a single parallel abstraction.

References

[1] J. Aldrich and C. Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In *ECOOP*, 2004.

[2] J. Boyland. Why we should not add readonly to Java (yet). *JOT*, 2006. Special issue: ECOOP 2005 Workshop FTfJP.

[3] D. Clarke and S. Drossopoulou. Ownership, Encapsulation and the Disjointness of Type and Effect. In *OOPSLA*, 2002.

[4] D. Clarke, T. Wrigstad, J. Östlund, and E. Broch-Johnsen. Minimal Ownership for Active Objects. In *APLAS*, 2008.

[5] D. G. Clarke, J. Potter, and J. Noble. Ownership Types for Flexible Alias Protection. In *OOPSLA*, 1998.

[6] B. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. Duato. Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks. In *ISCA*, 2011.

[7] F. S. de Boer, D. Clarke, and E. B. Johnsen. A Complete Guide to the Future. In *ESOP*, 2007.

[8] A. Greenhouse and J. Boyland. An Object-Oriented Effects System. In *ECOOP*, 1999.

[9] E. B. Johnsen and O. Owe. An Asynchronous Communication Model for Distributed Concurrent Objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.

[10] J. Östlund and T. Wrigstad. Regions as Owners. In *IWACO*, 2011.

[11] J. Östlund, T. Wrigstad, D. Clarke, and B. Åkerblom. Ownership, Uniqueness, and Immutability. volume 11 of *TOOLS*, 2008.

[12] T. Wrigstad. *Ownership-Based Alias Management*. PhD thesis, Royal Institute of Technology, Kista, Stockholm, May 2006.

[13] T. Wrigstad, F. Pizlo, F. Meawad, L. Zhao, and J. Vitek. Loci: Simple thread-locality for java. In *ECOOP*, 2009.

[14] A. Yonezawa, J.-P. Briot, and E. Shibayama. Object-oriented concurrent programming abcl/1. *SIGPLAN Not.*, 21(11):258–268, June 1986.

[2] http://shootout.alioth.debian.org/