# The Joelle Programming Language

## Evolving Java Programs Along Two Axes of Parallel Eval

<table>
<thead>
<tr>
<th>Johan Ostlund</th>
<th>Stephan Brandauer</th>
<th>Tobias Wrigstad</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uppsala University</td>
<td>Uppsala University</td>
<td>Uppsala University</td>
</tr>
<tr>
<td><a href="mailto:johan.ostlund@it.uu.se">johan.ostlund@it.uu.se</a></td>
<td><a href="mailto:stbr3377@it.uu.se">stbr3377@it.uu.se</a></td>
<td><a href="mailto:tobias.wrigstad@it.uu.se">tobias.wrigstad@it.uu.se</a></td>
</tr>
</tbody>
</table>

### 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 **Joe** [1]), 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 inners 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** [3], Joe’s [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. 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 [3] introduced the notion of ownership-based effects. Joe1 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”.

---

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

---

* Supported in part by the Swedish Research Council within the UPMARC Linnaeus centre of Excellence.
Figure 1. (a) Active object with three regions, four methods & effects (thick). Objects inside hold cross-references (thin). (b) Example scheduling of messages $R_0$, $U_1$, $T_2$, $S_5$, $R_4$, $S_5$. Barriers in black and dependencies as phased lines.

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 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


http://shootout.alioth.debian.org/