A Language-oriented Approach to Teaching Concurrency

Tom Van Cutsem*
Vrije Universiteit Brussel
Brussels, Belgium
tvcutsem@vub.ac.be

Stefan Marr†
Vrije Universiteit Brussel
Brussels, Belgium
smarr@vub.ac.be

Wolfgang De Meuter
Vrije Universiteit Brussel
Brussels, Belgium
wdmeuter@vub.ac.be

ABSTRACT

This paper argues in favour of a language-oriented approach to teach the principles of concurrency to graduate students. Over the past years, the popularity of programming languages that promote a functional programming style has steadily grown. We want to promote the use of such languages as the appropriate basic tools to deal with the "multicore revolution".

We describe some of these programming languages and highlight two of them: Erlang and Clojure. We use these languages in a new graduate-level course that we will teach starting next academic year. Our goal is not to convince the reader that Erlang and Clojure are the best possible choices among this pool of candidate languages. Rather, our goal is to promote a functional programming style to tackle concurrency issues, and to teach this style in a programming language that makes it easy, straightforward and convenient to use that style.

We do not want to get bogged down in a discussion on the usefulness or importance of learning new programming languages. For a good summary of the diverse advantages of studying new programming languages, we refer to a recent white paper by the ACM SIGPLAN education board [6].

1. BACKGROUND

Starting next academic year, the first author will teach a new graduate-level course on concurrency at the Flemish Free University of Brussels in Belgium. As in many other institutions worldwide, our computer science curriculum is slowly beginning to adapt to the reality that virtually every new computer is a parallel machine.

To date, our computer science curriculum featured just one course that touches upon concurrent programming. However, this course focuses on the traditional topics of using concurrency for parallelism and high-performance computing (HPC). As a consequence, the course mostly focuses on low-level concurrency primitives, employing libraries such as Pthreads, OpenMP and MPI for C/C++.

Our new course is an opportunity to teach students the principles of concurrent programming without necessarily tying this to the domain of HPC. Moreover, our university has a good track record of using a language-oriented approach to teach many fundamental computer science principles (first-year students are taught the classic Structure and Interpretation of Computer Programs course [1], and are well-versed in Scheme). Our experience suggests that the difficult problems of concurrent programming would be better taught using a language-oriented approach, rather than by using libraries bolted onto existing languages that seem inherently unfit for the job.

2. EMBRACING IMMUTABILITY

The message that we want to bring across to students is to mitigate the issues of concurrent programming by embracing a functional programming style. Functional programming is a broad concept, often associated with referential transparency, immutability, determinism, higher-order abstractions and expressive parametric type systems. Our focus here is mostly on immutability. We want to teach students how to program with immutable data structures.

Why this focus on immutability? Joe Armstrong put it succinctly by stating that "Side effects prevent concurrency" and "the absence of side effects is the key to increasing concurrency" [2]. And why via a new programming language? Well-designed languages make the right approach to solving a problem the "path of least resistance". Although one can adopt a functional programming style in virtually any language, that does not necessarily make it the natural way. Also, adopting a functional programming style in a language students already know may make it harder for them to unlearn their existing habits in using the language. Finally, it is hard to maintain a functional programming style throughout a larger program if the language's core libraries do not embrace this style as well.

3. CHOOSING A LANGUAGE

We used the following criteria to create a list of candidate languages for the course. First, the language should be sufficiently practical for students to experiment with (i.e. it needs a reliable implementation and sufficient documentation). Second, the language has to promulgate a functional programming style. Third, the language has to have some-

*Postdoctoral Fellow of the Research Foundation, Flanders (FWO).

†Funded by a doctoral scholarship of IWT-Vlaanderen.

CCP 2010 Reno, Nevada USA

thing out of the ordinary – to paraphrase Alan Perlis: the language should change the way students think about (concurrent) programming, or else it is not worth knowing.

Below we list potential candidate languages that adhere to the above criteria. We split them up according to their typing scheme – statically versus dynamically typed, for reasons that we discuss later.

Statically typed languages.

Haskell is a practical yet pure functional language. Simon Peyton-Jones's paper on Software-transactional Memory [10] is a particularly good starting point for concurrent programming in Haskell. Another candidate is **F#**, which is one of the main supported languages on Microsoft's `.NET` platform. It has built-in abstractions to do asynchronous programming and to easily compose asynchronous (parallel) tasks. **Scala** has a good actor library that enables Erlang-style message-passing concurrency. Although Scala promotes a functional programming style, it makes it just as easy to write traditional imperative code. Also, its actors are not truly isolated from one another (i.e. they can still share state). **JoCaml** is a dialect of OCaml that uses the Join-calculus [5] to coordinate concurrent activities. **Go**, while not being a functional language, features an interesting concurrency model that derives from Hoare's CSP and Milner's π-calculus. Go features concurrent *goroutines* that communicate with each other synchronously via channels. Unfortunately, Go is designed primarily as a systems programming language, so it still embraces imperative programming.

Dynamically typed languages.

Erlang started out as an experimental programming language developed at the Computer Science Lab of telecom giant Ericsson. Syntactically, it derives from Prolog. Its concurrency model derives from Hewitt and Agha's Actor model [7]. Erlang features lightweight concurrent *processes* that communicate via asynchronous messages. Processes are isolated (they do not share memory). The process isolation is the basis for Erlang's failure handling model, which allows the creation of highly reliable concurrent systems.

Clojure is a dialect of LISP that uses the JVM as its runtime platform. It embraces the use of immutable data structures. Even though Clojure is a LISP dialect, it shuns LISP's imperative language features. By default, variables are immutable. It features Software Transactional Memory to coordinate updates to *Refs* (mutable references). It also features *agents* which support message-passing concurrency. Unlike Erlang, Clojure's agents are not fully isolated and require a shared-memory architecture to communicate.

Oz is a multi-paradigm language, influenced by concurrent logic programming languages. It features lightweight threads that coordinate through shared (logic) variables, but also has a *port* abstraction that can be used to build Erlang-style processes.

4. WHY ERLANG AND CLOJURE?

Why did we settle on Erlang and Clojure as the drivers for our course? First, our cultural background is in dynamically typed programming languages. Most of our research is done in Scheme, Lisp, Smalltalk and Javascript, so we were prejudiced to pick a dynamically typed language. Other institutes have different cultures and would probably end up making different choices, which is perfectly fine.

Second, Erlang and Clojure are complementary: both epitomize a programming model for a different kind of hardware model. Erlang embraces message passing, which maps well onto distributed-memory hardware. Clojure's prominent software-transactional memory system is used to coordinate updates of multiple threads on shared references. This model is best supported by shared-memory hardware models. Oz, while able to encode both message passing and shared-memory concurrency, embraces neither model. That makes it more flexible, but also less of an exemplar.

5. PROGRAMMING PATTERNS

Choosing the right concurrent language to get started is important, but it is not sufficient. Programming languages only offer primitives out of which more high-level abstractions can be built. Many of these high-level abstractions are described in a language-neutral manner as patterns.

For concurrent and parallel programming, a good pattern language already exists [9]. We will use this pattern language to introduce students to the right high-level abstractions, but we will also ground these abstract patterns in concrete implementations in our chosen languages.

The effectiveness and popularity of two patterns in particular will deserve further attention in our course. The first is fork/join parallelism through divide-and-conquer algorithms as pioneered by Cilk [3] and made mainstream via Doug Lea's Fork/Join framework for Java [8]. The second is Google's MapReduce [4] abstraction. It is interesting to observe that both of these abstractions have a strong affinity with functional programming style: fork/join algorithms are expressed naturally using recursive functions, and MapReduce is a prime example of a higher-order function.

6. CONCLUSION

This paper takes the position that the difficulties of concurrent programming are best mitigated by engaging students in a functional programming style, in particular by exposing them to programming with (mostly) immutable data structures. We also feel that the best way to immerse students in a functional programming style is by selecting a programming language that embraces this style.

Multicore programming is hard enough as it is. We might as well make sure that our students learn to use the best tools available for the job.

7. REFERENCES

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, 1985.

[2] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[3] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, 1995.

[4] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[5] C. Fournet and G. Gonthier. The reflexive cham and the join-calculus. In *POPL '96: Proceedings of the*

23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 372–385, New York, NY, USA, 1996. ACM.

[6] S. N. Freund, K. Bruce, K. Fisler, D. Grossman, M. Hertz, D. Lea, G. T. Leavens, A. Myers, and L. Snyder. Why undergraduates should learn the principles of programming languages, 2010. ACM SIGPLAN Education Board White Paper.

[7] C. Hewitt. Viewing control structures as patterns of passing messages. *Artif. Intell.*, 8(3):323–364, 1977.

[8] D. Lea. A java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, pages 36–43, New York, NY, USA, 2000. ACM.

[9] T. Mattson, B. Sanders, and B. Massingill. *Patterns for parallel programming*. Addison-Wesley, 2004.

[10] S. Peyton-Jones. Beautiful concurrency. In A. Oram and G. Wilson, editors, *Beautiful Code*. O'Reilly, 2007.