Designing and Implementing an Integrated Identity Strategy (Mar 4th Webinar) - Save Your Seat

Joe Duffy on the Future of Concurrency and Parallelism

Joe Duffy is a lead architect on an OS incubation project at Microsoft, and was the architect for Parallel Extensions to .NET. He is also the author of Concurrent Programming on Windows.

The reason we wanted to talk to you is that research into concurrency and parallelism appears to have taken a turn. For much of the last half-decade or so there has been a lot of interest in functional programming, alone or in combination with software transactional memory. The focus seemed to be on trying to achieve provable thread safety via eliminating or at least containing side effects.

But more recently there has been a lot of talk about languages and libraries that seek to solve problems in other ways, more or less ignoring the issue of side effects. The ones that come to mind include:

- D 2.0: This language defaults to containing objects within a single thread, even going to far as to store globals in thread local storage. If you want something to pass from one thread to another you have to make it immutable or otherwise indicate that it is actually thread-safe.
- .NET 5: VB and C# are both altering the language to support asynchronous programming almost as an alternative to multi-threading.
- Grand Central Dispatch, TPL Dataflow, and several other libraries are heavily invested in message passing as a means for cross-thread communication.

InfoQ:
Are we really seeing a major shift in how multi-threaded applications are going to be written or are these just refinements of what we have been doing all along?

Joe Duffy:
I actually have two answers.

My first answer is that, according to the past several decades of research, no, this isn't a major shift to how parallel programs are architected.

The Actor model, introduced nearly 40 years ago, was truly revolutionary, and made 1st class the notion that agents would concurrently orchestrate some larger interesting task through message passing. This general idea has surfaced again and again over the years, accumulating useful extensions and variants over time. Hoare CSP's, Liskov's Argus, occam, Ada, MPI, Erlang, and more recently, languages like Scala, Clojure, and Go. Specialized architectures have even been built, such as The Connection Machine, that use message passing to accomplish highly scalable fine-grained parallelism.

In my book, for example, I recommend that everybody architect their concurrent programs as an asynchronous sea of mostly-isolated islands of code that communicate through message passing, and internally make use of shared-state through task and data parallelism.

My second answer is different. I would say that, yes, this is a major shift for most of the development community, because mainstream programming environments are now adopting the ideas that we've seen in research and experimental languages over the years.

I fully expect that C#, C++, and Java, for example, will have Actor-based offerings within the next 5 years. In the meantime, developers on such platforms will experiment with new languages, and continue to increasingly build architectures in this style without first class support in their primary environment. Developers have done this for years, through a combination of threads -- now tasks -- and, going back even further, using COM, EJBs, and web services.

In fact, a very specialized class of programmers on this planet -- technical computing developers in HPC, science, and on Wall Street -- are already accustomed to doing this with specialized toolkits, like MPI.

The major shift we face will be that mainstream languages will start to incorporate more concurrency-safety -- immutability and isolation -- and the platform libraries and architectures will better support this style of software decomposition. OOP developers are accustomed to partitioning their program into classes and objects; now they will need to become accustomed to partitioning their program into asynchronous Actors that can run concurrently. Within this sea of asynchrony will lay ordinary imperative code, frequently augmented with fine-grained task and data parallelism.

InfoQ:
And do you think these are steps along the right path or should we be looking elsewhere?

Joe Duffy:
Yes, absolutely, this is the right path forward, in my opinion.

It's important not to fall into the trap of believing in a single silver bullet, however. Message passing is not a panacea; frequently the best path to scalability is data parallelism. Functional programming and immutability is not a panacea; if you're replacing your C algorithms and need to compete performance-wise, you very well might need to use one of those familiar and efficient mutable data structures from your favorite algorithms book. We won't throw out decades' worth of research, although we will need to evolve the right parts of our programs in the right ways.

Another important thing to realize is that there is shared state in message passing models. This is a dirty little secret. It is emergent and arises through the communication of a bunch of stateful Actors. Just as fine-grained invariants can be broken and witnessed by concurrent interleavings, so can these more emergent coarse-grained invariants be broken and witnessed. Concurrency means indeterminism, no matter how you slice it. There was an interesting paper last year at ICFP on finding "races in the large" in Erlang, a pure message passing programming language. Many of the same synchronization concerns and techniques used in shared memory programs arise at a higher level of abstraction in message passing programs; this tends to make them easier to understand and reason about, however it's just a series of tradeoffs, really.

InfoQ: Do you think languages such C#, C++, and Java can offer an Actor based model just with libraries? Or to do it right would we need to see some sort of syntactical changes to the languages?

Joe Duffy:
It's certainly possible that these platforms can dip their toes in the lake by first trying a library-only approach. This is a great way to experiment.

To make Actors first class for the developer, however, it is likely language designers will eventually explore language support. There are two main challenges that I believe will drive them in this direction:

1. **Syntax**. As we have seen with, say, TPL and C#'s new await keyword, nothing can beat the expressiveness of a language construct. All great Actor languages in the past have made them first class.
2. **Safety**. To have a truly world class Actors model, you need the safety. This means isolation and immutability. Obviously, none of the aforementioned languages have these notions today; it's plausible unsafe Actors could be offered, but this is less ideal.

In a sense, we already see experimentation with library-only approaches. Microsoft's CCR and TPL Dataflow technologies, for example, ended up on the library route for message passing, with neither syntactic support nor safety. But the results are more imperative mechanisms than higher-level programming models. That's fine and works well for them, but they would certainly benefit from language support built on top. There's no doubt that has to be in the platform's future, if this style of architecture and programming is to become ubiquitous.

InfoQ: Back in July of last year you wrote an article with the line "Isolation first; immutability second; synchronization last". Do you see a need for more research into language constructs that would enforce isolation or immutability?

Joe Duffy:
Absolutely! We need to make concurrency something that real human beings can introduce into their programs with a high degree of confidence; eliminating races, threads, and locks, and getting compile-time safety in on the game is the only path forward.

There are some analogies with type-safety. We write C# and Java code without needing to worry about type-system holes. An entire class of program errors is eliminated by-construction; it is beautiful. The same needs to happen for concurrency-safety. But it sure won't be easy retrofitting these concepts into existing languages like C#, C++, and Java.

There is a great deal of research already showing what might be possible in this area, more in the Java community than in C# or C++. But the ideas transfer quite readily, and of course imperative languages will continue to learn important lessons from functional ones too.

InfoQ: Of the stuff coming out of Java, which ones in particular do you think are worth watching?

Joe Duffy:
In general, Java has been a step or two ahead of C# and C++ when it comes to concurrency over the past decade. This includes the “old era” of concurrency, which includes Doug Lea’s work back in Java 1.4, and the whole Java Memory Model JSR133 effort. More recently, there has been a great deal of good work in immutability and controlling side-effects. The work I am most excited about is Deterministic Parallel Java (DPJ) coming out of the University of Illinois. If you look at the past dozen major language conferences, e.g. PLDI and POPL, in addition to the associated workshops, you will see many other cool and innovative ideas building upon one another. It’s unclear how much of this research will make it into Java proper, however as an incubator of great ideas it has been great.

It’s also great to see that, as .NET has seen with the CLR over the years, the JVM is now a truly multi-language platform. Both Clojure and Scala began by targeting the JVM, and then promptly added support for also running on the CLR. As the runtimes become increasingly parallel, the languages above them are finding more incentive to expose those capabilities in safe ways. Heck, even JavaScript has been given the await keyword Google’s new Traceur extensions to JavaScript. Now all we need are browsers that ship with first class internal concurrency and we’ll have yet another multi-language parallel runtime.

InfoQ: If I may turn from languages and libraries to tools for a moment, what are your thoughts on currency bug detecting tools such as CHESS and Cuzz?

Joe Duffy:
On one hand, I love them. Any pragmatic tool that helps to find obscure and costly concurrency errors is a wonderful thing. These are some of the most painful bugs to test: before these tools became broadly usable, teams at Microsoft regularly spent months upon months doing long-haul stress on exotic hardware, and even then never found them all. And they are definitely some of the most painful and costly bugs to find in the wild too: there are some pretty famous examples of costly concurrency errors missed during testing, like the race condition that caused the Northeast Blackout of 2003.

On the other hand, that these tools must exist is a symptom of the larger problem: our languages are unsafe and do nothing to help us write correct concurrency programs by-construction. For every bug uncovered by a tool like this, there are fifty other highly subtle multi-threaded interactions that a programmer had to spend hours getting right. (And probably a few that the tool missed, given the explosive search space required to permute the interleavings.) Just consider for a moment all that wasted productivity!

About the Book Author

Joe Duffy is lead architect on an OS incubation project at Microsoft, where he is responsible for its developer platform, programming language, and concurrency programming model. Prior to this role, he founded the Parallel Extensions to .NET effort, and served as its architect. Joe is an author and frequent speaker, most recently writing Concurrent Programming on Windows (Addison-Wesley), and currently working on Notation and Thought, a fun historical journey through the world of programming languages. When not working or writing, he can be found traveling with his wife, Kim, writing music, guzzling wine, or reading about the history of mathematics. You can read more of his thoughts on his blog.

1

Please see https://www.infoq.com for the latest version of this information.