Programming Models and Languages for Distributed Computation

Christopher S. Meiklejohn
Université catholique de Louvain
Louvain-la-Neuve, Belgium
christopher.meiklejohn@gmail.com

March 3, 2016

Contents

1 Promises

1.1 Relevant Reading

- *Promises: linguistic support for efficient asynchronous procedure calls in distributed systems*, Liskov and Shrira, PLDI 1988 [7].
- *Multilisp: A language for concurrent symbolic computation*, Halstead, TOPLAS 1985 [4].

1.2 Commentary

Outside of early mentions from Friedman and Wise on a *cons* cell with placeholder values [3] and Baker and Hewitt's work on incremental garbage collection [1], *futures* originally appeared as one of the two principal constructs for parallel operations in MultiLisp. MultiLisp attempted to solve a main challenge of designing a language for parallel computation: how can parallel computation be introduced into a language in a way that fits with the existing programming paradigm. This problem is motivated by the fact that computer programmers will need to introduce concurrency into applications because automated analysis may not be able to identify all of the points for parallelism. Halstead decides there is quite a natural fit with a Lisp/Scheme: expression evaluation can be done in parallel. MultiLisp introduces two main concepts: *pcall*, to evaluate the expressions being passed to a function in parallel and introduce concurrency into evaluation of arguments to a function, and *futures*, to introduce concurrency between the computation of a value and the use of that value. Halstead also notes that futures closely resemble the "eventual values" in Hibbard's Algol 68, however were typed distinctly from the values they produced and later represented. [4]

In 1988, Liskov and Shrira introduce the concept of a *promise*: an efficient way to perform asynchronous remote procedure calls in a type-safe way [7]. Simply put, a promise is a placeholder for a value that will be available in the future. When the initial call is made, a promise is created and the asynchronous call to compute the value of the promise runs in parallel with the rest of the program. When the call completes, the value can be "claimed" by the caller.

An excerpt motivation from *Promises: linguistic support for efficient asynchronous procedure calls in distributed systems (Liskov and Shrira, PLDI 1988)*:

> "Remote procedure calls have come to be the preferred method of communication in a distributed system because programs that use procedures are easier to understand and reason about than those that explicitly send and receive messages. However, remote calls require the caller to wait for a reply before continuing, and

therefore can lead to lower performance than explicit message exchange."

The general motivation behind the work by Liskov and Shrira can be thought as the following critiques of two models of distributed programming.

- The Remote Procedure Call (RPC) paradigm is preferable by programmers because it is a familiar programming model. However, because of the synchronous nature of RPC, this model does not scale in terms of performance.

- The message passing paradigm is harder for programmers to reason about, but provides the benefit of decoupling of request and response, allowing for asynchronous programming and the subsequent performance benefits.

Promises attempts to bridge this gap by combining the remote procedure call style of building applications, with the asynchronous execution model seen in systems that primarily use message passing.

The first challenge in combining these two programming paradigms for distributed programming is that of order. Synchronous RPC imposes a total order across all of the calls in an application: one call will fully complete, from request to response, before moving to the next call, given a single thread of execution. If we move to an asynchronous model of RPC, we must have a way to block for a given value, or result, of an asynchronous RPC if required for further processing.

Promises does this by imagining the concept of a *call-stream*. A *call-stream* is nothing more than a stream of placeholder values for each asynchronous RPC issued by a client. Once a RPC is issued, the *promise* is considered *blocked* asynchronous execution is performed, and once the value has been computed, the *promise* is considered *ready* and the value can be *claimed* by the caller. If an attempt to *claim* the value is issued before the value is computed, execution blocks until the value is available. The stream of placeholder values serves as an implicit ordering of the requests that are issued; in the Argus system that served as the implementation platform for this work, multiple streams were used and related operations sequenced together in the same stream[1].

1.3 Impact and Implementations

While promises originated as a technique for decoupling values from the computations that produced them, promises, as proposed by Liskov and

[1]Promises also provide a way for stream composition, where processes read values from one or more streams once they are *ready*, fulfilling placeholder *blocked* promises in other streams. One classic implementation of stream composition using *promises* is the Sieve of Eratosthenes.

Shrira mainly focused on reducing latency and improving performance of distributed computations. The majority of programming languages in use today by practitioners contain some notion of *futures* or *promises*. Below, we highlight a few examples.

The Oz [6] language, designed for the education of programmers in several different programming paradigms, provides a functional programming model with single assignment variables, streams, and promises. Given every variable in Oz is a dataflow, and therefore every single value in the system is a promise. Both Distributed Oz [5] and Derflow (an implementation of Oz in the Erlang programming language) [2] provide distributed versions of the Oz programming model. The Akka library for Scala also provides Oz-style dataflow concurrency with Futures.

More recently, promises have been repurposed by the JavaScript community to allow for asynchronous programs to be written in direct style instead of continuation-passing style. ECMAScript 6 contains a native Promise object, that can be used to perform asynchronous computation and register callback functions that will fire once the computation either succeeds or fails [8].

References

[1] H. C. Baker, Jr. and C. Hewitt. The incremental garbage collection of processes. *SIGPLAN Not.*, 12(8):55–59, Aug. 1977.

[2] M. Bravo, Z. Li, P. Van Roy, and C. Meiklejohn. Derflow: Distributed deterministic dataflow programming for erlang. In *Proceedings of the Thirteenth ACM SIGPLAN Workshop on Erlang*, Erlang '14, pages 51–60, New York, NY, USA, 2014. ACM.

[3] D. P. Friedman and D. S. Wise. Aspects of applicative programming for parallel processing. *IEEE Transactions on Computers*, C-27(4):289–296, April 1978.

[4] R. H. Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(4):501–538, 1985.

[5] S. Haridi, P. Van Roy, and G. Smolka. An overview of the design of distributed oz. In *Proceedings of the second international symposium on Parallel symbolic computation*, pages 176–187. ACM, 1997.

[6] M. Henz, G. Smolka, and J. Würtz. Oz-a programming language for multi-agent systems. In *IJCAI*, pages 404–409, 1993.

[7] B. Liskov and L. Shrira. *Promises: linguistic support for efficient asynchronous procedure calls in distributed systems*, volume 23. ACM, 1988.

[8] Wikipedia. Futures and promises — wikipedia, the free encyclopedia, 2016. [Online; accessed 4-March-2016].