A Functional Approach to Numerical Simulations

Anthony West

12 Nov 2011

Numerical simulation is a tool for research in a wide variety of subjects, including physics, chemistry, materials science, meteorology, fluid mechanics, molecular biology, and finance. Despite their superficial differences, these domains of application usually share a common underlying mathematical formulation of their problems. The numerical solutions they require may be sets of different states of the modelled system, or they may be sequences representing an evolution of these states. The solutions are sometimes so large as to require a terabyte of storage, and querying them requires that their retrieval be efficient.

The entire process is usually implemented *ad hoc*. The simulators are written in a fast, low-level language, and a scripting language is used to deploy them, often in parallel on a cluster of machines. Their output is stored on disk in the form of many large flat files, and low- or high-level languages are then used to patch these files together and to query their contents. Workers are usually familiar only with imperative languages, so they tend to write in Fortran, C, Python, *etc*. The result is that writing, extending, testing, and debugging the code is tedious and time-consuming labor, often delegated to subordinate workers. Much of it is duplicated across research groups.

Observe, however, that every interesting part of the simulation process entails the computing of a function. The simulator computes sets and sequences that are functions of the initial state. In so doing, it computes interactions (*e.g.*, forces) that are functions of the instantaneous state. Subsequent queries (*e.g.*, averages) on the computed solutions are also naturally expressed as functions. In fact, the entire enterprise is a composition of functions, whose polymorphic type is parameterized by the type of the modelled system, *i.e.*, by the domain of application.

For all the above reasons, the software should be written in a programming language that is functional rather than imperative. Functional languages are conducive to abstraction, composition, concision, and correctness, qualities that facilitate the building of simulation software and the analyzing of its output. More recent functional languages, such as Racket, provide powerful macro systems for syntax extension, which help the programmer create domain-specific languages. Others, such as Haskell, evaluate lazily, thereby enabling a stream-oriented approach to data, which suits the sequences computed by simulations. And by promoting referential transparency and immutable data structures, a functional language would more clearly reflect the mathematical definition of all parts and stages of a simulation.

The objection will be raised that if a functional language were used to write all parts of the software, used even to implement the interactions between elements of the modelled system, then execution would become infeasibly slow. But note that the computation of interactions is the only bottleneck, for its time complexity is typically no smaller than quadratic in the size of the modelled system, whereas the remaining computations are asymptotically faster. The language should therefore provide a foreign function interface to a low-level language (*e.g.*, C or Fortran) in which to implement those bottlenecks. This optimization should accelerate the simulator to competitive speeds.

As for the generated solutions, they are invariably relations and should therefore be stored in a relational database. DBMSs relieve the user of the burden of managing flat files, and some can be extended to provide abstractions for querying sequential data. The *de facto* relational query language is SQL, and its most used commands in fact form a functional subset of it. But it is verbose and Turing-incomplete, and its syntax is designed for commerce. In place of SQL should be a concise, functional, Turing-complete relational query language whose syntax is well suited to scientific mathematics.

Indeed, the entire simulation process should be conducted in one monolithic software system, written entirely in one functional language. In addition to having the virtues mentioned so far, and in addition to providing libraries of algorithms and data structures that permit the modelling of the domains listed above, this language should provide abstractions that allow the three software components—simulation, storage, and query—to be integrated seamlessly, so that, for example, disk-resident data can be fetched as easily by a running simulation as by a query.

Implementation responsibilities should reflect the simulator's polymorphic type. That is, the domain-specific code should factor out like the type parameter that it represents, and it should be written only by user-scientists, who are more likely to have intuitions about optimizing domain-specific bottlenecks. The domain-agnostic remainder, which corresponds to the parameterized type, should be designed and implemented by computer scientists.

Such a disciplined and unified system would give computer scientists a platform for research in databases and programming languages, if not also in numerical algorithms. To physical scientists it would give immense gains in productivity.