Advanced Programming Language Design in Enterprise Software

A lambda-calculus theorist wanders into a datacenter

Chet Murthy
IBM Research
chet@watson.ibm.com

Abstract

Enterprise software systems automate the business processes of most nontrivial organizations in the world economy. These systems are immensely complex, and their function is critical to our living standards and everyday lives. Their design, implementation, and maintenance occupies many thousands of programmers and engineers, who work in what are aptly called the "COBOL dungeons"[1] of the IT sector. These systems have persisted, growing by accretion – some for decades; there are enterprise systems in existence today whose original and even subsequent authors are retired or deceased. Such extraordinarily old, multi-layered systems might appear to be the last place to apply *avante-garde* techniques, but in fact, they are extremely promising candidates, and for reasons directly connected to their history and structure.

In this talk we take a tour of several deployed enterprise software systems, and demonstrate that the appropriate application of methods from functional programming can and does in fact yield dramatic performance improvements and thus commercial advantage in the design and implementation of enterprise software. This concrete application is an instance of a general plan for the application of advanced programming language design and analysis methods, to the problem of improving enterprise software. It is the thesis of this talk that to a great extent, it is in enterprise software that advanced PL techniques can find their most advantageous application. This talk literally breaks no new ground in PL research: every technique discussed is nearly two decades old, and our goal is to introduce PL researchers to what we feel is an ideal target for their work.

Categories and Subject Descriptors D.3.2 [*PROGRAMMING LANGUAGES*]: Language Classifications; F.4.1 [*MATHEMATICAL LOGIC AND FORMAL LANGUAGES*]: Lambda calculus and related systems; H.2.4 [*DATABASE MANAGEMENT*]: Systems

General Terms Languages

Keywords Transaction processing, enterprise software, functional languages

[1] The modern version of COBOL is Java/C#.

POPL'07 January 17–19, 2007, Nice, France.
ACM 1-59593-575-4/07/0001.

1. Enterprise Systems in a Nutshell

Enterprise software systems usually consist of a collection of individual applications, receiving and processing requests and returning replies. The dominant communication paradigm in use for several decades has been remote procedure call, and these servers normally process many such requests concurrently. It is very common that in a nontrivial application, a "front-end" server will, in the course of processing a request, dispatch subsidiary requests to "back-end" servers, and this layering of servers can proceed to many "tiers" of servers. The processing of a request is loosely referred to as a "transaction", and while the term has a precise meaning in the context of databases and concurrency-control, in actual systems the term is much more liberally applied to almost any sort of request/response processing.

The multi-tier aspect of these systems comes about for three reasons: (a) they are assembled out of off-the-shelf software which often cannot be linked together into a single application; (b) they are assembled over time, and "front-end" servers or entire collections of servers, are often added long after the original system was deployed, in order to provide new function or new access, and (c) the programmers who maintain and extend these systems often do not understand and do not have the skills to dig into the internals of some system they must extend, preferring to add new function "in front of" the pre-existing system.

A key aspect of servers that process transactions is that, aside from side-effects that are applied to some (often external) persistent store, all side-effecting operations during the transaction leave behind no remnant effects. Practically, one can view the actual code of the transaction as a functional program that happens to read from its persistent store, write back to that store, and produce some functional data-structure.

Internally, the code of a single server can itself be decomposed into "layers". In the business, these many layers of software are referred to as "frameworks", and often there can be many of them in a single program. Empirically we find that the execution of code in these layers *also* tends not to leave behind side-effects, except, again, by their effect on some store.

Taken together, these two empirical observations mean that to a great extent, one can both replace "layers" in a server with equivalent implementations, and entire server applications with equivalent implementations, to a greater extent than in other sorts of software.

2. ML in the Enterprise: reducing the overhead of framework layering

As a concrete example of such replacement, the Xylem team took a deployed transactional application at a major e-business company, and replaced one of its "layers" with a completely rewritten version, using ML as an intermediate language. We were able to do

so without disturbing the rest of the application, and achieved significant performance improvements. This application went live on the Internet several years ago. What is more, we then proceeded to replace neighboring layers in the same code, and by application of naive versions of various program transformation techniques well-known to the POPL community, we were able to get further speed-up.

In short, we found that not only did applying *avante-garde* technology from the PL community allow us to improve a particular component, but by applying it to other components in an incremental manner, we were able to gain *further* speed-up.

Our ML implementation, Xylem, is a simple polymorphically typed ML with algebraic datatypes and weak module system. We compile Xylem to Java, first applying naive versions of a collection of standard techniques from the PL literature:

- partial evaluation and simple inlining
- aggressive memo-ization
- wide-ranging common subexpression elimination
- deforestation
- data-type specialization
- view types

ML's fit with enterprise software is (paradoxically) extremely good *because* such software tends to be written without intricate side-effects that are visible across layers; further, the accretive nature of the software development process in this area results in systems with many, many layers, and the application of techniques such as partial evaluation is thus extremely effective to reduce the overhead of framework layering. View types were found to be extremely effective in combination with abstract datatypes, as a way of eliminating framework layering while leaving source code unaffected.

Further, the wide-ranging optimizations we applied were significantly more effective than any low-level compiler optimizations, and this is a recurring pattern: the application of the high-level methods developed in the PL community over many years can have profound effects on the performance of these systems.

Paradoxically, we can sell ML into the enterprise, not based on its *beauty*, but on its *speed*.

3. Future Directions: Fixing the rest

Improving other parts of the application will require different techniques, and it is interesting to contemplate how many of them are at this point old news in the PL community:

lazy languages An important class of problems arises in the ETL (extract/transform/load) business, and these problems can be characterized as stream-processing problems: creating, transducing, joining, and consuming streams of data. We believe such processing is a perfect fit for lazy languages, especially since the optimization methods from that community can be applied to improve the performance of what are today computationally intensive problems.

logic programming The application we described had a *significant* data-access framework to provide a convenient interface to data stored in a relational database. This is very common, and these frameworks tend to be complex and inefficient. In addition, the view of data that the application would prefer to use, often does not correspond to the view that the database provides. We believe that by applying techniques from logic programming we can "push down" queries written against the application's data abstractions, down to the database, where they can be executed as SQL. Such query pushdown can yield transformative performance improvements.

I/O automata, reactive systems modeling Presentation frameworks are responsible for producing human-readable results, and processing input into a form internal to the application. The typical web-application has lots of complex user-interaction function, and this function is often implemented using the model/view/controller (MVC) paradigm. We believe that methods from the concurrent automata modeling community, used to model reactive and embedded systems, can be profitably applied to reduce the complexity and cost of building presentation layer code.

First-order and Higher-order Attribute Grammars Modern web-based user-interfaces are characterized by significant amounts of code-generation and event-based processing. It appears that attribute grammars, and especially higher-order attribute grammars, might be a good fit for this sort of functionality.

3.1 The Distributed System

Heretofore we've focused on single server nodes and the code that runs on them, with the exception of database access. As more and more of these systems is written in functional languages, the opportunity will arise to both coalesce code (from multiple servers to a single server) where desirable, and also to select the "cut points" (where a large program is distributed across multiple nodes) in an automatd manner, in order to optimize, say, round-trips between browser and server, amount of browser-side state, disconnected operation, or minimizing the number of places where persistent state must be managed.

4. Conclusion

It is an article of faith in the functional language and wider programming language research community, that functional languages, partial evaluation, and a host of other tools and methods, result in simpler, easier-to-understand, and easier-to-develop and maintain programs. Such a value proposition, since it is difficult to measure, is difficult to sell to the users of enterprise software. We have found empirically that a different value proposition, based on *pure performance* is much easier to explain, and we have successfully deployed an ML system into a commercial transaction-processing system in the most important application in that system on that basis. We see further opportunities for such application, and we believe that enterprise software is almost uniquely suited for the application of such methods.

5. About the author

Chet Murthy has been a Research Staff Member at IBM Thomas J. Watson Research Center since 1995. Previously he spent four years doing postdoctoral research in theorem-proving and formal methods at Cornell University and INRIA-Rocquencourt. He devotes a significant part of his time to debugging critical situations at large e-business and transaction-processing customers, and the rest of his time transforming that experience into research and development initiatives. He received a Ph.D. in computer science in 1990 from Cornell University, and a BSEE in 1986 from Rice University.

Acknowledgments

The author would like to acknowledge the Xylem team, and specifically Dennis Quan and Joseph Latone, without whom these ideas would have remained merely a promising theory.