Lazy *vs.* Strict*

Philip Wadler
University of Glasgow
Glasgow G12 8QQ, Scotland
`wadler@dcs.glasgow.ac.uk`
`http://www.dcs.glasgow.ac.uk`

Lazy, or call-by-need, languages schedule work dynamically by building closures, and shun side effects; strict, or call-by-value languages avoid the overhead of closures and may exploit side effects. Each style has complementary advantages and complimentary adherents.

The gap between the lazy and strict camps has two dimensions, which we shall name *style* and *models*. Recent developments suggest that along both dimensions the gap is shrinking. We list some commercial applications of each kind of language, and examine each dimension of difference in turn.

Applications

Here are two commercial applications that demonstrate the utility of laziness, and one that demonstrates the utility of strictness.

ICAD is a market leader in the area of CAD systems. Such systems represent a design by a program that constructs a tree (or graph) of parts, which may then be displayed. An overview requires that the tree be constructed from its root down to a limited depth, while a close-up requires that a portion of the tree be evaluated to a greater depth. Lazy evaluation is essential for good performance, because it minimises the portion of the tree constructed. Indeed, a few years ago I could not have written this, because the use of lazy evaluation by ICAD was a trade secret.

Natural Expert is an expert system shell marketed by Software AG, which interfaces to a database system called Natural [3]. Users write shell programs in a language similar to Haskell. Lazy evaluation is essential to manage interaction with the database, which returns potentially enormous data structures. Over twenty commercial applications have been developed in Natural Expert, including IVANHOE, which invoices services for every flight passing through Orly or Roissy airport in Paris.

Erlang is an strict functional language, used by Ericsson to program telecommunications switches [2]. Erlang fuses ideas from functional and concurrent languages. On the functional side it provides ease of allocating, traversing, and deallocating complex structures; on the concurrent side it provides processes,

*To appear in *ACM Computing Surveys*, June 1996.

message passing, and distribution. Synergy arises because the messages passed are complex structures; for instance, functional pattern matching may determine which concurrent message to receive. Strictness is essential to make it easy to predict the order in which concurrent operations occur. Several hundreds of thousands of lines of Erlang code appear in Ericsson products, and it has been found to be superior to C++ on a number of projects.

Narrowing the style gap

The advantages of lazy and strict languages appear fundamentally immiscible. Laziness is an advantage precisely when it is difficult to predict the required order of evaluation in advance, while strictness is an advantage precisely because it allows such prediction.

But within the last few years, lazy functional programmers have hit upon a way to incorporate features that require a specified order of evaluation, including exception handling, assignment, input-output, and concurrency. Moggi discovered that each of these operations, and more, could be modelled denotationally by applying a notion from category theory called a *monad* [5]. Functional languages are not far removed from denotational semantics, and it proved straightforward to adapt Moggi's approach to structure functional programs [7] and to incorporate side effects [8]. These uses of monads have been advanced by a number of researchers, as surveyed elsewhere [8].

That a strict language may be embedded in a lazy one has been known for two decades, thanks to results by Reynolds, Plotkin, and others concerning continuation passing style (CPS). Indeed, the direct use of CPS for this purpose was tried earlier, in Hope+ and in Haskell, with some success; and monads may be regarded as a mild generalisation of CPS [7]. Perhaps there is nothing surprising here, except the degree to which monads have surpassed CPS in popularity as a technique for embedding strict behaviour within a lazy language.

It is also well known that a lazy language can be embedded in a strict one, via a variant of CPS or via closures. These applications, too, fit into the monad framework, and tentative explorations at using monads to structure SML and Erlang programs have met with some success. Again, this is perhaps not surprising, but it is pleasing that both lazy-to-strict and strict-to-lazy embeddings can be treated in the monad framework. The recent popularity of monads within the lazy community brings new hope that the strict and lazy communities may come closer together.

Narrowing the model gap

To a first approximation, lazy or call-by-name languages are modelled by Church's λ calculus, while strict or call-by-value languages are modelled by Plotkin's λ_v calculus.

The (simply typed) λ calculus is complete, in that any equation which is true in all models is provable, while λ_v is not. For example, if we write $\texttt{let}\, x =$

M in N as an abbreviation for $(\lambda x.\, N)\, M$, then the equation

$$\texttt{let}\; y = (\texttt{let}\; x = L\; \texttt{in}\; M)\; \texttt{in}\; N \quad = \quad \texttt{let}\; x = L\; \texttt{in}\; (\texttt{let}\; y = M\; \texttt{in}\; N)$$

holds in both lazy and strict languages. But while provable in λ it is unprovable in λ_v.

On the other hand, the λ_v calculus provides a reasonable model of evaluation, while λ does not. For example, λ_v suggests that the cost of evaluating the term

$$\texttt{let}\; x = M\; \texttt{in}\; x + x$$

is approximately the cost of evaluating M plus the cost of an addition, while λ suggests that the cost is approximately the cost of an addition plus *twice* the cost of evaluating M.

The resolution of these difficulties is to choose less simple but more accurate models. The construction of such models is remarkably recent, and they turn out to be surprisingly similar.

Church's λ calculus may be improved by replacing it with the call-by-need calculus discovered by Ariola and Felleisen and by Maraist, Odersky, and Wadler [1]. This calculus provides a reasonable model of evaluation. It is observationally equivalent to λ calculus, and hence retains its completeness properties.

Plotkin's λ_v calculus may be improved by Moggi's λ_c calculus, as proposed by Sabry and Wadler [6]. This calculus is based on Moggi's work with monads, and is sound and complete over all monad models. It contains λ_v as a subset, and remains a reasonable model of evaluation.

Astonishingly, these improved calculi are nearly identical. Maraist, Odersky, Turner, and Wadler [4] studied variants LET of call-by-value and NEED of call-by-need that differ by exactly one law,

$$\texttt{let}\; x = M\; \texttt{in}\; N \quad \longrightarrow \quad N, \quad \text{if } x \text{ is not free in } N$$

which is absent in the former and present in the latter. The similarity appears to hold for a range of formulations: Sabry and Wadler [6] conjecture that augmenting λ_c by the above law again turns a model of call-by-value into one of call-by-need.

For too long the lazy and strict camps have been untied. With some rearrangement, may they become united?

References

[1] Z. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler, A call-by-need lambda calculus, *22'nd ACM Symposium on Principles of Programming Languages*, San Francisco, California, January 1995.

[2] J. Armstrong, R. Virding, C. Wikström, M. Williams, *Concurrent Programming in Erlang* (2'nd edition), Prentice Hall, 1996.

[3] N. Hutchison, U. Neuhaus, M. Schmidt-Schauss, and C. Hall, Natural Expert: A commercial functional programming environment, *Journal of Functional Programming*, to appear.

[4] J. Maraist, M. Odersky, D. N. Turner, and P. Wadler, Call-by-name, call-by-value, call-by-need, and the linear lambda calculus, *11'th International Conference on the Mathematical Foundations of Programming Semantics*, New Orleans, Louisiana, March 1995.

[5] E. Moggi, Notions of computation and monads, *Information and Computation*, 93:55–92, 1991.

[6] A. Sabry and P. Wadler, A reflection on call-by-value, *1'st ACM International Conference on Functional Programming*, Philadelphia, Pennsylvania, May 1996.

[7] P. Wadler, The essence of functional programming. *19th ACM Symposium on Principles of Programming Languages*, Albuquerque, New Mexico, January 1992.

[8] P. Wadler, How to declare an imperative. *International Logic Programming Symposium*, Portland, Oregon, MIT Press, December 1995.