General-Purpose Multiparadigm Programming Languages: An Enabling Technology for Constructing Complex Systems

Timothy A. Budd Timothy P. Justice Rajeev K. Pandey

Technical Report 95–60–04

Department of Computer Science
Oregon State University
Corvallis, Oregon 97331–3202
{budd, justict, rpandey}@cs.orst.edu

May 1, 1995

Abstract

Multiparadigm programming languages have been envisioned as a vehicle for constructing large and complex heterogeneous systems, such as a stock market exchange or a telecommunications network. General-purpose multiparadigm languages, as opposed to hybrid multiparadigm languages, embody several prevalent programming paradigms without being motivated by a single problem. One such language is Leda, which embodies the foundational paradigms of imperative, functional, logic, and object-oriented programming. We explore aspects of solving complex problems using Leda, in order to illustrate the benefits of using a multiparadigm language in expressing solutions to complex systems. We claim that general-purpose multiparadigm programming languages like Leda greatly expedite solutions to a variety of complex problems.

1 Introduction

Complex heterogeneous systems are being constructed with increasing frequency. Software is a vital component in all complex systems. Choosing the appropriate programming language is a crucial decision in software development. Each programming language is associated with a programming paradigm–a problem view which is influenced by the constructs and concepts embodied in the programming language. While paradigms abound, computer scientists view four paradigms as foundational [BG94, GJ87, Kam94, Set89]. Traditionally, languages reflect one of these foundational paradigms:

- imperative programming (e.g., FORTRAN, C)
- object-oriented programming (e.g., Smalltalk, Eiffel)

- functional programming (e.g., LISP, ML)
- logic programming (e.g., Prolog)

Recently, a panel of experts at a workshop on future directions in programming languages [Kam94] identified the role of programming languages as that of an *enabling technology*, the machine tools of the computer age. Unfortunately, as concluded by this panel, languages have not always lived up to their role:

> Our society pays an enormous cost for the widespread use of old-fashioned "machine tools" for software development – that is, poorly designed and documented, frustratingly limited, unfriendly, and inefficient languages and language processors.

One approach to resolving the deficiencies of single paradigm languages is to combine the best known problem solving styles into a single language, thus allowing the programmer to choose whichever style most naturally fits a given problem. The resulting languages are referred to as *multiparadigm programming languages*. Creation of such languages is motivated by the observation that many complex problems contain subproblems whose solutions lend themselves to different programming paradigms. If a language were to possess the appropriate paradigms, the problem solution would be expedited by the fact that subproblems could avail of the paradigm that best expresses a solution.

Multiparadigm programming languages have been an active research area for over a decade. Much of the research effort has been devoted to developing implementation techniques for languages spanning several paradigms, as well as experimenting with varying the blended paradigms. Recently, attention has turned to the applicability of these languages to real-world problems. Some problems that have been touted as suitable for multiparadigm solution include a programming language compiler [Bud91], a stock market exchange [JGM86], and a telephone network simulation [Zav89].

Several approaches have been pursued in the creation of multiparadigm languages:

- *augmented languages*
  Augmented languages add additional paradigms to an existing language to permit users to utilize a new programming style without learning a completely new language. The additional paradigm usually represents a natural progression or evolution of the language, based on experience. For example, C++[Str91] extends C [KR88] with support for object-oriented programming.
- *hybrid languages*
  Hybrid languages typically extend an existing functional or logic programming language by embedding other paradigms. The primary motivation for these languages is to provide a wide range of standard programming and knowledge representation paradigms for solving complex problems in artificial intelligence [MNC+91]. Loops [SBK86] is an example of this type of language.
- *general-purpose languages*
  General-purpose multiparadigm languages seek to find the "ideal" blending of several major paradigms in order to provide a more expressive programming vehicle for general problem solving. Leda [Bud95] exemplifies this approach.

Most existing multiparadigm programming languages are not general-purpose, but belong to the augmented or hybrid variety, and are a result of the lack of very specific language features with respect to a specific problem. Rather than attack the problem with an inelegant solution, the language is extended in one of a

variety of ways. Here there is a close affinity between hybrid multiparadigm programming languages and a particular problem, since the language is essentially designed with the specific application in mind. While these languages have definite advantages in the context of a specific problem, they may not easily generalize to solving different problems. Combining the foundational paradigms to create languages like Leda has been identified as the next step in general-purpose language evolution [Ghe93]:

> All the major programming language styles - procedural, functional and logical - have application domains where they are particularly effective. This suggests that general-purpose programming languages must embrace a number of these different approaches. Consequently, it seems likely that the future of the major general-purpose programming languages will be as multi-paradigm languages.

General-purpose multiparadigm language research at Oregon State University has yielded the language G [Pla91], as well as Leda, our current language of interest. Leda combines all four foundational paradigms enumerated above. Retaining conciseness while combining the four paradigms was a design goal of Leda. While Leda adds two more paradigms than C++, the resultant language is actually smaller than C++[1]. By emphasizing the essential features–the *exemplars*–of the constituent paradigms, Leda facilitates blendings at various levels of granularity, from individual statements to complete modules. Figure 1 illustrates our view of general-purpose multiparadigm languages. This view permits the programmer to apply any of fifteen possible paradigm blendings to a single problem. The figure further illustrates Leda'a articulation of this view.

Leda is the product of six years of research in paradigm blending and language implementation. Leda has evolved through several implementations, is the subject of a textbook [Bud95], and has been taught internationally. Current research is focused on the development of a programming environment and production quality compiler, as well as studies of large application development in Leda.

2 A Multiparadigm Approach to Problem Solving

Various applications have been identified as being well-suited to a particular foundational paradigm. Figure 2 illustrates some of these applications. As problems grow in complexity, a number of the prototypical applications identified in the figure become interrelated subproblems of the much larger problem. This section examines three such problems.

2.1 A Telephone Network Simulation

Zave [Zav89] describes a multiparadigm approach to the construction of a telephone network simulator. The network consists of clusters of switches connecting local clusters of telephones via a collection of trunks. The switches all access a global database to translate numbers, and send billing data to a centralized billing repository.

Three specific computational requirements are identified in the problem description: simulation, numerical computing, and database processing. Figure 2 shows the corresponding paradigms best suited for this application to be object-oriented, imperative, and logic. Zave's implementation employs several independent programming languages in a compositional manner.

[1] For example, Leda has 54% fewer keywords and 67% fewer operators than C++.

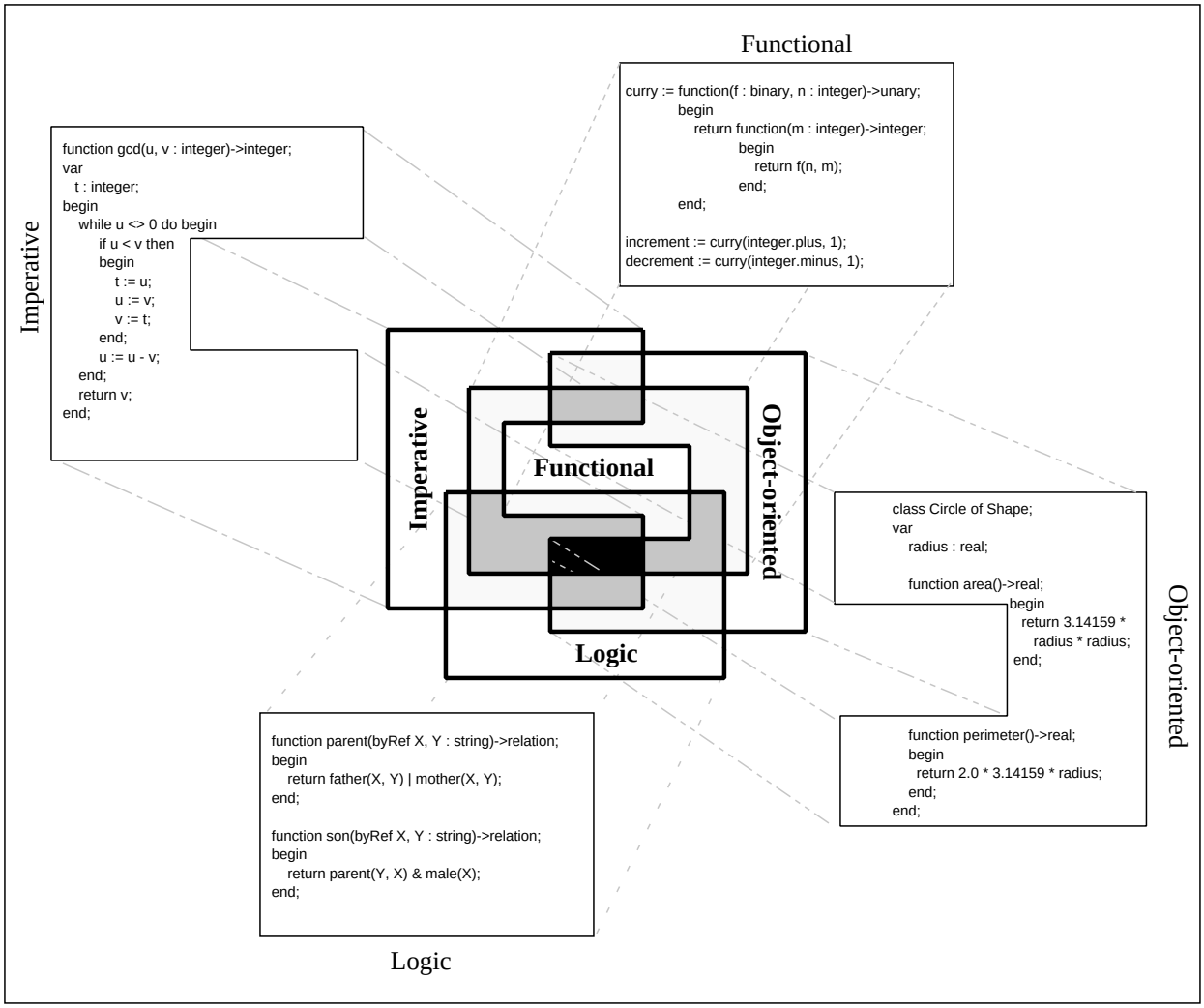

Figure 1: The general-purpose multiparadigm language Leda.

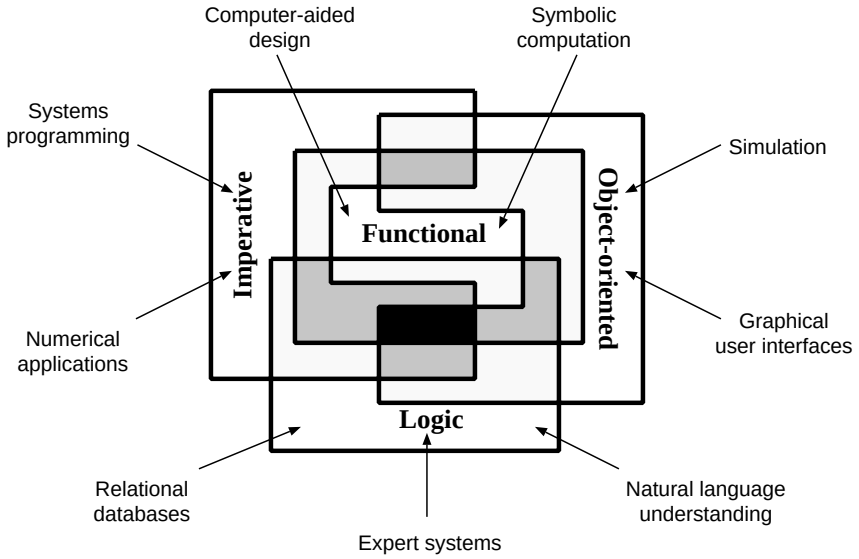

Figure 2: Strengths of foundational paradigms.

2.2 A Stock Market Application

A stock trading application is considered by Jenkins, Glasgow, and McCrosky [JGM86] as a potential application for a general-purpose multiparadigm language:

> As an example, consider the problem of building a system that supports manipulation of data on the trading of stocks on the Tokyo stock exchange. It would require access to a large database component to hold the raw data associated with trading...The system would require strong numerical capabilities to do data aggregation and statistical analysis. In addition, an inferencing capability applied to knowledge bases of various kinds would be needed to support intelligent interfaces for the many players involved...

From this description we can identify paradigm strengths that will need to be present in the solution language. The logic paradigm will need to be present to provide the database component. Numerical processing requirements suggest need for the imperative paradigm, while inferencing requires the logic paradigm. For any such application, graphical user interfaces will also be necessary, which are easily expressed in an object-oriented style. The interfaces may also require a degree of symbolic computation, motivating the presence of the functional paradigm as well.

2.3 A Language Compiler

We have recently constructed a compiler for the C programming language using Leda [JPB94]. The source language grammar is expressed as a set of rules, via the logic paradigm. These rules are encapsulated within an object-oriented framework that provides the services of scanning and parsing a program, as well

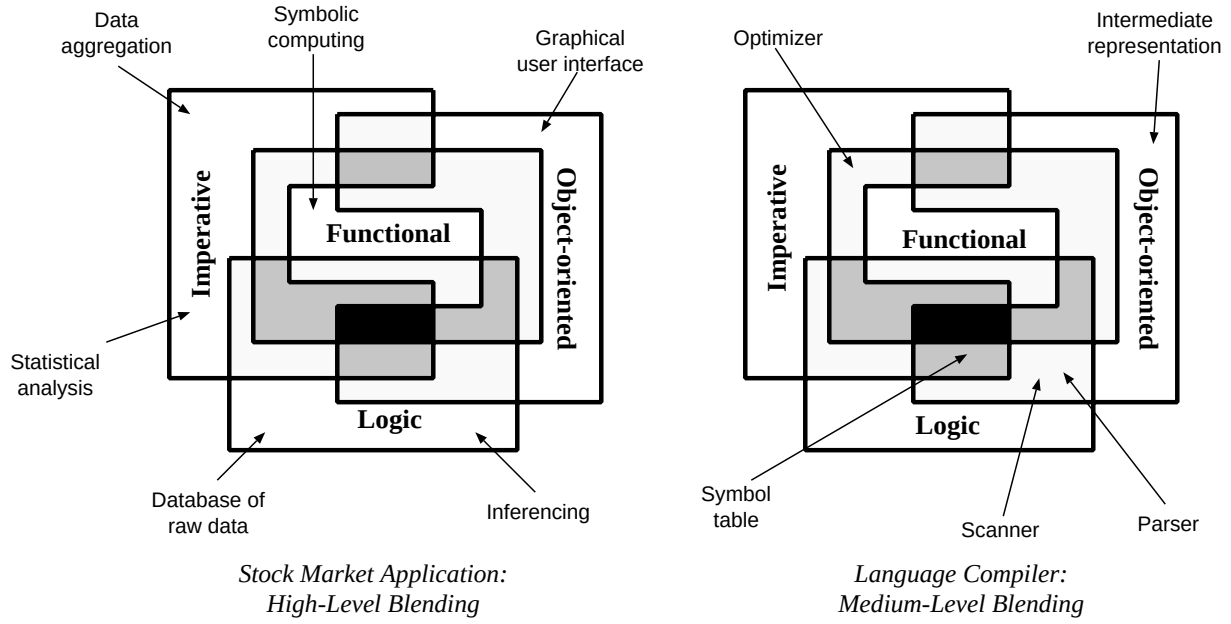

Figure 3: Examples of paradigm blending in Leda.

as populating a symbol table with relevant information. The intermediate representation consists of tuple objects, with functional optimizations performed on this representation. Direction of these activities is performed at an imperative level.

2.4 Evaluation

Zave's solution to the telephone network simulator utilizes several independent languages. In the absence of a general-purpose multiparadigm language, this approach would likely be the method of choice. A key benefit of Leda is that the programmer can utilize any of the foundational paradigms within a single linguistic framework. This reduces the syntactic overhead incurred when working on different parts of the system. Also reduced is the number of tools necessary for constructing the system. Furthermore, as the application evolves and features are added, existing functionality is likely to migrate from one paradigm to another. Performing this migration using independent languages can be difficult and sometimes infeasible. This results in the existing implementation having a strong influence on the choice of paradigms used for system enhancements. Leda eliminates the syntax shift that accompanies a language shift. As a result, implementing enhancements is motivated more by suitability of paradigm to the enhancement than by the existing implementation.

The partitioning of the stock trading application proposed by Jenkins, Glasgow, and McCrosky represents a high level blending of paradigms. The partitioning is illustrated in Figure 3. This approach exploits paradigm blending between modules. Each module is implemented in a single paradigm. Paradigm blending is achieved by combining the modules into the final application. Leda supports this type of blending.

The compiler application exhibits medium level paradigm blending. As illustrated in Figure 3, the major components of the compiler are implemented as blendings of several paradigms. The resultant application is comprised of a group of interacting computing agents in the form of objects, functions, and relations, as opposed to the traditional rigid sequence of phases.

While we have not presented examples of low level paradigm blending, many algorithms are amenable to multiparadigm solution. In addition to high and medium level paradigm blending, Leda supports low level blending, down to the individual statement. An examination of using Leda to implement many fundamental data structures and algorithms can be found in [Bud95].

3 Conclusion

Not only do multiparadigm programming languages afford the programmer access to a variety of problem solving approaches, there are also the synergistic benefits to be derived from the presence of the different paradigms. Multiparadigm programming languages allow for the simultaneous existence of multiple views of the same problem space: some aspect of a problem may be crucial from one paradigm's viewpoint, while being negligible in another. This attribute allows programmers to construct solutions that emphasize the crucial aspects of the problem at the appropriate phase of the solution.

The ideal solutions to complex systems will likely display a migration toward a fuller blending of programming styles. General purpose multiparadigm languages like Leda allow for such a migration to occur. By making all of the foundational paradigms available in a coherent and synergistic fashion, Leda presents the opportunity to expedite solutions to a variety of complex problems. It should be noted that Leda has already been used as a pedagogic vehicle in graduate-level courses worldwide, has freely available implementations, and is the topic of a recently published textbook.

Sha's [Sha94] recent identification of software "grand challenges" for industrial computing includes transportation systems, manufacturing systems, sensor systems, communications, and medical computing. Constructing these complex systems is a difficult and labor-intensive undertaking, whether or not a multiparadigm solution is being attempted. The mere presence of a variety of paradigms does not necessarily expedite a solution. However multiparadigm programming languages attack an essential aspect of complex systems that Brooks [Bro87] identifies: the drastic differences among the various system components is inherent in the problem, and is not an artifact of our current problem solving approaches. Access to drastically different paradigms working in concert via multiparadigm programming languages may be the best complex system solution strategy available today.

References

[BG94] Henri E. Bal and Dick Grune. *Programming Language Essentials.* Addison-Wesley Publishing Company, Wokingham, England, 1994.

[Bro87] Frederick P. Brooks, Jr. No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer*, 20(4):10–19, 1987.

[Bud91] Timothy A. Budd. Blending Imperative and Relational Programming. *IEEE Software*, 8(1):58–65, 1991.

[Bud95] Timothy Budd. *Multiparadigm Programming in Leda*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1995.

[Ghe93] Carlo Ghezzi. Modern non-conventional programming language concepts. In John A. McDermid, editor, *Software Engineer's Reference Book*, pages 44/1–44/16. CRC Press, Inc., Boca Raton, Florida, 1993.

[GJ87] Carlo Ghezzi and Mehdi Jazayeri. *Programming Language Concepts*. John Wiley & Sons, New York, second edition, 1987.

[JGM86] Michael A. Jenkins, Janice I. Glasgow, and Carl D. McCrosky. Programming Styles in Nial. *IEEE Software*, 3(1):46–55, January 1986.

[JPB94] Timothy P. Justice, Rajeev K. Pandey, and Timothy A. Budd. A Multiparadigm Approach to Compiler Construction. *SIGPLAN Notices*, 29(9):29–37, September 1994.

[Kam94] Samuel Kamin et al. Report of a Workshop on Future Directions in Programming Languages and Compilers. (available from Kamin's WWW page, URL http://www.cs.uiuc.edu), May 1994.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, New Jersey, second edition, 1988.

[MNC+91] Gérald Masini, Amedeo Napoli, Dominique Colnet, Daniel Léonard, and Karl Tombre. *Object-Oriented Languages*, volume 34 of *A.P.I.C. Series*. Academic Press Inc., San Diego, California, United States edition, 1991.

[Pla91] John Placer. The Multiparadigm Language G. *Computer Language*, 16(3/4):235–258, 1991.

[SBK86] Mark J. Stefik, Daniel G. Bobrow, and Kenneth M. Kahn. Integrating Access-Oriented Programming into a Multiparadigm Environment. *IEEE Software*, 3(1):10–18, January 1986.

[Set89] Ravi Sethi. *Programming Languages: Concepts and Constructs*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1989.

[Sha94] Lui Sha. Industrial Computing: A Grand Challenge. *IEEE Computer*, 27(1):12–13, 1994.

[Str91] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, Reading, Massachusetts, second edition, 1991.

[Zav89] Pamela Zave. A Compositional Approach to Multiparadigm Programming. *IEEE Software*, 6(5):6–9, September 1989.