Teaching Multiple Paradigms

Timothy Budd
Department of Computer Science
Oregon State University
Corvallis, Oregon
97331
USA
budd@cs.orst.edu
currently
Laboratoire I3S
Universite de Nice – Sophia Antipolis
250 rue Albert Einstein
06560 Valbonne FRANCE
(+33) 92 94 26 22
FAX: (+33) 92 94 28 98

March 7, 1994

Abstract

The advocates for logic programming, functional programming, and object-oriented programming have each in the past several years made convincing arguments as to the benefits of their style of program development. The basic idea of multiparadigm programming is to provide a framework in which these benefits can each be realized, and in which each of the different paradigms benefits from features provided by the others. In this paper I will introduce the multiparadigm programming language Leda that my students and I have been developing over the past several years. I will illustrate how programming features from each of the different paradigms I have named can be naturally expressed in this language.

1 Introduction

In the past decade object-oriented programming languages and software techniques have received an unprecedented level of publicity. And, indeed, the application of object-oriented techniques in both research and commercial projects has resulted in significant changes in the manner in which software is developed. This has led, naturally, to the desire to integrate object-oriented concepts into the educational curriculum in computer science.

But, while object-oriented concepts have received the lions share of publicity, equally impressive changes have been more quietly occurring in other areas of software development. In particular, considerable research has been conducted in the utility of programming in the functional style, and in the logic programming community.

The importance of this work should not be under-emphasized. Research results from the psychology of programming (such as [Petre 89]) indicate that expertise in programming is far more strongly related to the number of different programming styles understood by an individual than it is to the number of years experience in programming.

The primary thesis of this paper can be summarized in the following points:

- In addition to conventional imperative programming, a modern introduction to software development must expose the student to, not only object-oriented programming techniques, but functional and logic programming methods as well.
- Object-oriented, functional and logic programming methodologies need not be viewed as mutually exclusive, but should instead be considered to be complimentary software development techniques.
- A single programming language can be created which will permit the programmer to exercise the salient characteristics of each of these paradigms. Through the use of this language the important features of each school can be presented, and contrasting styles and design-tradeoffs can be better understood.

The remainder of this paper will be structured as follows. The next section will present a few comments on programming language design in general. The main part of this paper will be found in Section 3, which will give a single application, insertion sort in a linked list, as it could be presented in four different paradigms, yet all in the same programming language. Finally, the paper will end with some concluding remarks.

2 Programming Languages

Because no single existing well-known programming language permits the illustration of software development techniques in the style of each of the paradigms we have named (imperative, object-oriented, functional and logic), many educational institutions have elected to teach a number of programming languages. Common examples include Pascal or C for the imperative paradigm, Eiffel or C++ for object-oriented concepts, ML or Scheme for functional programming, and Prolog for logic programming.

The danger with this approach is that students will over-compartmentalize their knowledge. That is, techniques learned when programming in ML, for example, will be deemed to be applicable only to functional languages, and will not be considered as relevant when programming in, for example, C++.

A single language which permits software to be developed in all four paradigms can help to overcome this problem. Within a single framework students can more easily contrast and compare differing techniques, and form an understanding of the relative merits and uses for each style. Furthermore, a single language permits the combination of techniques even within a single program.

In the past several years my students and I at Oregon State University have been engaged in the development of such a language. Since the primary intent was pedagogical, and not commercial, the objective was to create a language that was relatively small and simple, about the same complexity as, for example, the Pascal language. Nevertheless, despite being simple, the language should permit the natural expression of algorithms in any of the four targeted paradigms, without unnecessarily favoring one style over the other.

The language which has resulted from these efforts is named Leda. The next section will illustrate how each of the four paradigms can be expressed in Leda. A more complete description, including a language reference manual and numerous example programs that integrate differing techniques, will be presented in a forth-coming book [Budd 95].

3 Paradigm Examples – Sorting

As we noted earlier, our purpose in this section is to illustrate how the Leda programming language supports each of four different programming paradigms; namely the imperative paradigm, the functional paradigm, the logic programming paradigm, and the object-oriented paradigm. We will do this by examining one specific problem, an insertion sort performed using linked-list structures, and provide solutions to this problem written in Leda that exemplify the salient characteristics of each of the four paradigms.

3.1 Linked Lists

A linked list is commonly defined as series of link values, which are dynamically constructed as instances of a structure. Although Leda does not have structures in the conventional sense (as found in, for example, C or Pascal), an adequate substitute is provided by a class declaration that defines only data fields, and does not define any of its own behavior. Such a declaration is shown in Figure 1. Each instance of the class maintains a single integer value, and a value which represents the remainder of the list.[1] A list of integers is simply a collection of such values, each element of the list referring to the next, with the final link holding a "next" field which is undefined.

Note that Leda does not have the concept of an explicit "pointer". Instead, the Leda language uses the more abstract notion that a value may or may not be defined. Each element in the list structure holds, in addition to an integer value, a field which is, in fact, the remainder of the list. In the final element this value is undefined, a state which can be tested using a system-provided predicate named **defined**. A realistic picture of a linked list is therefore a series of nested boxes:

[1] This list type is can only maintain integer values. The language Leda includes both type-parameterized classes and functions, and these can be used to create data types that can hold arbitrary values; our purpose here is to simply provide a gentle introduction to both Leda and each of the different programming styles.

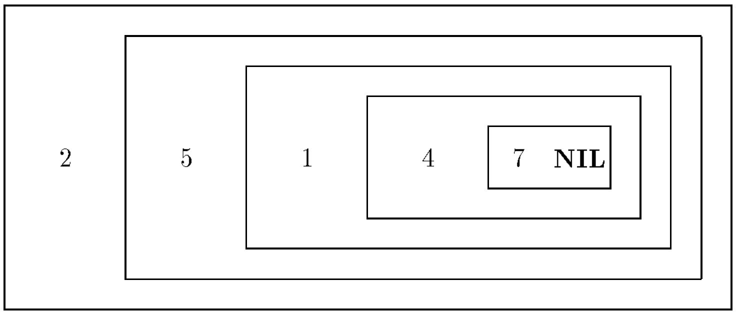
```
class intList;
var
    value : integer;
    next : intList;
end;

function head (lst : intList)->integer;
begin       { return first element in list }
    return lst.value;
end;

function tail (lst : intList)->intList;
begin       { return list with first element removed }
    return lst.next;
end;

function empty (lst : intList)->boolean;
begin       { undefined list is empty, all other lists are non-empty }
    return ~ defined(lst);
end;
```

Figure 1: Declaration of a Link for Linked Lists



We will use four functions in the manipulation of our simple linked lists. Every class declaration automatically causes the generation of an associated *constructor* function, having the same name as the class type. The constructor function takes a series of arguments matching in type and order the data fields in the class declaration. The constructor provides for both creation and initialization of a new instance of the class. The linked list of five elements described above might therefore be generated in the following fashion:

```
var
    aList : intList;
begin
```

```
    aList := intList(2, intList(5, intList(1, intList(4, intList(7, NIL)))));
    ...
end;
```

The value `NIL` a polymorphic value representing the state of being "undefined".

While the constructor is used to put a list together, the functions `head` and `tail` (Figure 1) are used to break a list apart. The function `head` returns the value assigned with the first element in a list, while the function `tail` returns the list formed with the first element removed. Both functions will generate a run-time error if they are applied to an empty list. Finally, the boolean function `empty` returns true if the argument is an empty list, and false otherwise.

3.2 An Imperative Sorting Procedure

The defining characteristic of the imperative approach to problem solving is the direct manipulation and modification of structures which maintain the "state" of the calculation. This is exemplified by the sorting procedure shown in Figure 2. The procedure takes as a single argument a list, which is transmitted using pass-by-reference (indicated by the `byRef` designation) and modified in situ.

The procedure operates by sorting the initial prefix of the list, enlarging the ordered section element by element until finally the entire list is sorted. The variable `current` is used to loop over the linked structure, preserving the invariant that all elements prior to the current are properly ordered. This invariant is trivial to initially establish; when `current` points to the first element, the list of previous values is empty, and thus automatically ordered.

To reiterate the point we made at the beginning of this section, the fundamental property of this solution that makes it characteristic of imperative programming is the idea of computing by making small changes to a data structure, until by a long series of such small changes the desired outcome is reached.

3.3 A Logic Programming Sorting Procedure

If imperative program is concerned with the development of a series of detailed step-by-step instructions which when executed combine together to produce a desired outcome, then the antithesis of the approach is perhaps epitomized by the techniques used in programming in the logic programming style. The hallmark of logic programming is that it is one of the purest examples of *declarative programming*. Rather than being concerned with *how* a solution will be attained, a logic programmer concentrates on defining precisely the *characteristics* of the desired solution, leaving it to the computer to discover exactly how this solution will be found.

A logic programming solution to the insertion sorting problem is shown in Figure 3. Logic programs are built around a stereotypical type of function, which returns a data type called a *relation*. A relation can be thought of as a form of boolean, although the reality (namely, the implementation) is slightly more complicated. The intent is that a function returning a relation is in fact returning true if the arguments satisfy a certain property. More importantly, the difference between a relation and a boolean is that a relation can be used to generate values in such a manner that the property is satisfied.

```
function sort (byRef aList : intList);
var
    current, p : intList;
    x, temp : integer;
begin
    current := aList;
    while defined(current) do begin

        { place current element into proper location }

        x := current.value;
        p := aList;

        while p ~= current do begin
            if x < p.value then begin
                temp := p.value;
                p.value := x;
                x := temp;
            end;
            p := p.next;
        end;

        { advance to next position }
        current.value := x;
        current := current.next;
    end;
end;
```

Figure 2: The Imperative Sorting Procedure

```
function insertion (startList : intList,
    x : integer, byRef newList : intList)->relation;
var
    temp : intList;
begin
    return empty(startList) & newList <- intList(x, NIL)
        | ~ empty(startList) &
            (x < head(startList) & newList <- intList(x, startList)
            | x >= head(startList)
                & insertion(tail(startList), x, temp)
                & newList <- intList(head(startList),temp));
end;

function sort (startList:intList, byRef endList:intList)->relation;
var
    temp : intList;
begin
    return empty(startList) & endList <- startList
        | ~ empty(startList) & sort(tail(startList), temp)
            & insertion(temp, head(startList), endList);
end;
```

Figure 3: A Logic Programming Program for Insertion Sort

The first relation shown in Figure 3 is given by the function named `insertion`. This relation can be thought of as characterizing what it means for a new element (the variable `x`) to be inserted into a sorted list (the variable `startList`) resulting in a new list (the variable `newList`, which is transmitted using pass-by-reference, and thus can be used to return a value to the caller.)

The body of a relation is typically a single expression constructed using logical operators. The logical expression divides naturally into several cases. If the starting list is empty, then a new list is simply a list containing a single element. A special form of assignment is used, written using the `<-` arrow. This form of assignment performs the binding of value to name, and in addition returns a boolean "true" value. (A third characteristic of the assignment operator is the ability to "undo" the binding of name to value in certain situations. However, this property of the arrow operator is not used in the current problem.)

If, on the other hand, the starting list is not empty, then the insertion of a value which is smaller than the first element in the list is simply a new list constructed with the new element at the beginning. Finally, if the new value is larger than or equal to the start of the list then the insertion is characterized by the list that has the head of the starting list for a first element, and the insertion of the new element into the remainder of the list. (The latter requires two steps and the introduction of a temporary variable).

A sorted list can similarly be recursively characterized by a defining predicate. A list is sorted if it is empty, or if not empty if the first element is an inserted value into the sorted version of the remainder of the list.

Notice there are no imperative actions (other than assignment) anywhere in these two functions. There are only boolean expressions which define the properties that a solution must possess. Furthermore, these properties closely match a logical definition of what it means for an insertion to be performed on a list, and what it means for a list to be sorted. It is this close relationship between description or definition and program that makes logic programs particularly appealing in many problem domains.

3.4 A Functional Solution

The remaining two paradigms to be examined are the functional programming paradigm and the object-oriented paradigm. The two techniques are similar in that a broad goal for both approaches is to permit and facilitate the creation of general, reusable tools which can be employed to raise the level of abstraction in programming. That is, rather than starting each new program on a foundation composed solely of the programming language alone, as if no other program had ever been written, both the functional and object oriented programming paradigms encourage the development of tools which can be created within the language itself, but which can be carried from one project to the next.

On a superficial level, where the two techniques differ is in the mechanism used to express these tools. In functional programming such tools are expressed as functions, while in the object-oriented approach tools are generally expressed as classes. On a deeper level, these two philosophies differ in how they view the process of computation. The object-oriented approach builds on the conventional view that computation proceeds as a series of steps which each make small changes to the state of the machine. The object-oriented technique takes this to a higher level of abstraction, providing a mechanism for generating "computers in the small". Each miniaturized-computer is responsible

```
function insertion (x : integer, aList : intList)->intList;
begin
    if empty(aList) then
        return intList(x, NIL)
    else
        if x < head(aList) then
            return intList(x, aList)
        else
            return intList(head(aList), insertion(x, tail(aList)));
end;

function reduce (aList : intList, ident : intList,
    binFun: function(integer, intList)->intList )->intList;
begin
    if empty(aList) then
        return ident
    else
        return binFun(head(aList), reduce(tail(aList), ident, binFun));
end;

function sort (aList:intList)->intList;
begin
    return reduce(aList, NIL, insertion);
end;
```

Figure 4: Functional Program for Insertion Sort

for a certain amount of memory and a certain set of behaviors. Rather than directly manipulating memory, computation proceeds by asking different computing units in turn to manipulate their own memory.

Functional programming, on the other hand, rejects altogether the notion that computation consists of the modification of memory cells. Rather than changes to state, the functional programming view is that computation is a sequence of transformations. Instead of changing existing values, new values are constructed which possess certain desired attributes.

These characteristics are exhibited by the functions in Figure 4, which illustrate a functional approach to the insertion sort algorithm. The emphasis on transformation is illustrated by the first function, which takes as argument a single value and an existing list (presumably already sorted) and returns a new list in which the value has been correctly placed so as to preserve ordering. Compare this function with the similarly named function in Figure 3. Besides the use of explicit control-flow constructs, the most obvious difference is that the earlier function returned the result through a pass-by-reference parameter, while the latter version returns the new list as the result of executing the function. Because of this, the two actions required in the general case (inserting the element into the tail of the list, and inserting the head of the list into the resulting value) can be performed in one expression without the need to introduce a temporary variable.

The tool-building characteristic of functional programming is exemplified by the second function shown in Figure 4. This function is, for historical reasons, named **reduce**. The **reduce** function has nothing to do with sorting or ordering values, and thus can be used in a wide variety of situations. The **reduce** function takes as arguments a list, a value named the "identity", and a binary function. If the list is empty the identity value is returned. Otherwise, the head of the list is stripped off, and the binary function applied to this value and the result of recursively applying **reduce** to the remainder of the list.

To sort a list, the **reduce** function is simply invoked using the original (unsorted) list, an empty list (represented by the undefined value **NIL**) as an identity, and the previously defined function named **insertion** as the binary function argument. The result sorts a list of values through a sequence of recursive calls on **reduce**, followed by a sequence of calls on **insertion** as the recursive calls on **reduce** are "unwound". We can envision this process working on a five-element list as follows:

```
reduce(2 5 1 4 7)
insert(2, reduce(5 1 4 7))
insert(2, insert(5, reduce(1 4 7)))
insert(2, insert(5, insert(1, reduce(4 7))))
insert(2, insert(5, insert(1, insert(4, reduce(7)))))
insert(2, insert(5, insert(1, insert(4, insert(7, reduce(NIL))))))
insert(2, insert(5, insert(1, insert(4, insert(7, NIL)))))
insert(2, insert(5, insert(1, insert(4, 7))))
insert(2, insert(5, insert(1, 4 7)))
insert(2, insert(5, 1 4 7))
insert(2, 1 4 5 7)
1 2 4 5 7
```

3.5 An Object-Oriented Solution

The last approach we will examine in this paper is the object-oriented paradigm. As we noted earlier, the object-oriented philosophy views computation as the creation of a universe populated by collections of processing elements. Each processing element, like the computer itself, consists of a collection of memory values and a series of functions that can be invoked in order to get the processing element to perform certain actions. The functions defined within the scope of a class definition are sometimes referred to as *methods*, a term we will use interchangeably with "function."

In the object-oriented world it is considered improper to do something "to" a value (meaning, to directly modify the internal state of an object), rather, all changes to the state of an object should be performed by the object itself in response to a request (generated through an invocation of an object method). It is as if an object were a sentient being, and indeed, personification and anthropomorphic descriptions of computation abound in the literature on object-oriented programming.

These ideas are expressed in our object-oriented version of insertion sort, shown in Figure 5. In order to associate behavior with our data structures we introduce a new data type, called an ordered list. In addition to maintaining two data values, an ordered list has two actions that it can perform.

```
class orderedList;
var
    value : integer;
    next : orderedList;

    function append (newLink : orderedList);
    begin
        next := newLink;
    end;

    function check ()->orderedList;
    var
        n : orderedList;
    begin
        if defined(self.next) then
            if next.value < self.value then begin
                n := self.next;
                self.next := n.next;
                n.append(self.check());
                return n;
            end;
        return self;
    end;
end;

function sort (aList : intList)->orderedList;
var
    newList : orderedList;
begin
    while defined(aList) do begin
        newList := orderedList(head(aList), newList);
        newList := newList.check();
        aList := tail(aList);
    end;
    return newList;
end;
```

Figure 5: An Object-Oriented Insertion Sort

The behavior for instances of a class is indicated by the functions defined within the scope of the class description. The first action is named “append”. It merely changes the value of the next-link data field in the structure. (Remember, it is not considered proper to directly modify the data fields in an object; rather, you must request, through a function call, that the object modify itself). The second action is more complex, and is at the heart of the insertion sorting algorithm. This action requests that a value check to see that it is in order, changing the list, if necessary, to ensure it is ordered.

To form a sorted list each element will be added one by one. As each new element is added to the front of a list, the element is asked (using the check function) to place itself in the proper relationship to the other elements.

This particularly sorting algorithm is admitted contrived in order to most completely illustrate the features of the object-oriented philosophy. The most important characteristic to note concerning the object-oriented solution is that no *changes* to the state of objects are ever made except by functions associated with the objects themselves.

4 Concluding Remarks

In this paper we have introduced Leda, a multiparadigm programming language that incorporates features of imperative, functional, object-oriented or logic programming software development. The intent in the development of Leda has been to create a simple language in which algorithms can be naturally expressed in any of these four styles, and which does not favor any one style over the others. We have illustrated Leda by presenting four versions of an insertion sort algorithm, one in each of the four targeted paradigms.

The pedagogical intent behind the design of Leda is the hope that it can provide a framework through which students can be introduced to a wide variety of software development techniques, and can better appreciate the relative merits and advantages of contrasting styles.

References

[Budd 95] Timothy A. Budd, *Multiparadigm Programming in Leda*, to be published by Addison-Wesley, 1995.

[Petre 89] Marian Petre, “Finding a Basis for Matching Programming Languages to Programming Tasks,” PhD Thesis, Department of Computer Science, University College London, 1989.