Operational Versus Definitional: A Perspective on Programming Paradigms

Allen L. Ambler, University of Kansas
Margaret M. Burnett, Michigan Technological University
Betsy A. Zimmerman, General Dynamics

Programming paradigms guide problem solving and provide frameworks for expressing solutions. This article categorizes paradigms according to their emphasis on stating procedures versus constraining the solution set.

A programming paradigm is a collection of conceptual patterns that together mold the design process and ultimately determine a program's structure. Such conceptual patterns structure thought in that they determine the form of valid programs. They control how we think about and formulate solutions, and even whether we arrive at solutions at all.

Once we can visualize a solution via a paradigm's conceptual patterns, we must express it within a programming language. For this process to be effective, the language's features must adequately reflect the paradigm's conceptual patterns. A language that reflects a particular paradigm well is said to support that paradigm. In practice, a language that supports a paradigm well is often hard to distinguish from the paradigm itself.

A language rarely supports just one paradigm. Typically, it will borrow liberally from many paradigms for its features and support more than one. In this article, we discuss various programming paradigms independent of supporting languages.

In his 1978 Turing Award Lecture, R.W. Floyd[1] stated his belief that

> ... the current state of the art of computer programming reflects inadequacies in our stock of paradigms, in our knowledge of existing paradigms, in the way we teach programming paradigms, and in the way our programming languages support, or fail to support, the paradigms of their user communities.

Floyd described three categories of paradigms: those that support low-level programming techniques (for example, copying versus sharing data structures), those that support methods of algorithm design (divide and conquer, dynamic programming, etc.), and those that support high-level approaches to programming (such as functional and rule-based paradigms). Floyd showed how different programming languages support the paradigms in each of these categories. We focus here on paradigms that support high-level programming.

0018-9162/92/0900-0028$03.00 © 1992 IEEE

Further, we group them according to their approach to problem solving. The *operational* approach describes step-by-step how to construct a solution. The *demonstrational* approach is a variation on it that illustrates the solution operationally for specific examples and lets the system generalize these example solutions for other cases. While the demonstrational approach is definitely operational, it produces some very different results and is reviewed here as a separate category. The *definitional* approach is different. It states properties about the solution to constrain it without describing how to compute it.

These three approaches can be viewed on a continuum from operational to definitional. We start our discussion with the operational approaches, describing each paradigm in a separate section. A sidebar for each paradigm contains a "pure" language solution to the problem of sorting a list into some linear order. This problem allows many different solutions and lets us illustrate different conceptual patterns associated with each paradigm.

In choosing an algorithm for illustrating a particular paradigm, our objective is to best illustrate the natural style of the paradigm, not to find the most efficient solution. In most cases, proponents of these paradigms and the languages supporting them can significantly improve the efficiency of our algorithms.

We have written the solutions in hypothetical languages. In this way we avoid communicating solutions that depend on a particular language. Some of these "pure" languages may resemble real languages. While this resemblance may improve understandability, we do not mean to imply association with any existing languages. We do, however, briefly describe real languages that typify support for the paradigm.

Imperative paradigm

```
procedure swap (x, y)
   temp := x
   x := y
   y := temp

procedure bubblesort (list)
   for i := 1 to MAX–1 do
      for j := MAX downto i+1 do
         if list[j–1] > list[j] then
            swap (list[j–1], list[j])
```

Procedure bubblesort repeatedly compares each of two neighboring elements and exchanges them if they are out of order. The final state of the list is the sorted list. Note that the original list is destroyed in the process.

Many programming languages support this model; most also include extensions that support other paradigms to varying degrees. Today's imperative languages contain a variety of borrowed mechanisms, such as non-side-effecting functions, recursion, and dynamic allocation via pointers. They nevertheless remain conceptually dominated by the machine model.

Operational paradigms

Step-by-step computational sequences characterize operational paradigms. The most difficult aspect of programming within this approach is determining if the operationally computed value set is, in fact, the solution value set. Debugging and verification techniques concentrate on these programming problems. The finer the operational control, the harder it is to define the computed value set and to verify it as identical with the solution value set. We must often settle for a computed value set that is "sufficiently close" to the solution value set, where we interpret "sufficiently close" to mean that the two value sets are indistinguishable over the expected subclass of actual problems.

Operational paradigms are of two basic types: those that proceed by repeatedly modifying their data representation (side-effecting) and those that proceed by continuously creating new data (non-side-effecting). Side-effecting paradigms use a model in which variables are bound to computer storage locations. As a computation proceeds, these storage locations are repeatedly revised (that is, the variables get multiple assignments). When a computation ends, the final values of specified variables represent the results. There are two kinds of side-effecting paradigms: *imperative* and *object-oriented.*

Non-side-effecting paradigms include those that were traditionally called *functional* paradigms. Today, it is important to distinguish functional approaches that are operational from those that are definitional. This is a fine line. Most attempts at general definitional approaches to programming are eventually tainted by operational necessities that work their way back into the supported paradigm. Nevertheless, certain functional approaches are clearly operational and will be discussed here, while others are less operational and will be discussed with the definitional approaches.

Operational paradigms define sequencing explicitly. Operational sequencing is either serial or parallel. If parallel, it can be defined by cooperating parallel processes (asynchronous parallel) or single processes applied simultaneously to many objects (synchronous parallel). Each operational paradigm includes corresponding asynchronous and synchronous parallel variants. We discuss some of these at the end of this section.

Imperative. The imperative paradigm is characterized by an abstract model of a computer that consists of a large store. The computer stores an encoded representation of a computation and executes a sequence of commands that modify the store. This paradigm is best represented by von Neumann-style machine architectures. Although von Neumann machines underlie the implementation of almost all the paradigms discussed in this article, the imperative paradigm uses this machine model for conceptualizing solutions. The other

paradigms, by contrast, use conceptual models removed from this implementation model.

Programming in the imperative paradigm is dominated by determining what data values will be required for the computation, representing these data values by associating them with storage locations, and deriving a step-by-step sequence of transformations to the store so that the final state represents the correct result values.

Object-oriented paradigm

```
Class Sequence
|
quickSort
| pivot lowerPart middlePart upperPart |
if ((self size) >= 2) then
   pivot := self selectPivot.
   lowerPart :=
      ((self class) new) addAll: (self select: [elt where elt < pivot])
   middlePart :=
      ((self class) new) addAll: (self select: [elt where elt = pivot])
   upperPart :=
      ((self class) new) addAll: (self select: [elt where elt > pivot])

   lowerPart quickSort
   upperPart quickSort

   self updateFrom: lowerPart and: middlePart and: upperPart
|
selectPivot
   return (self first)
|
```

All computations are accomplished by sending messages to objects. The objects respond by following methods of the same names. For example, the method definition for quickSort says that when a sequence receives the message quickSort, it creates three new local objects, lowerPart, middlePart, and upperPart. The (built-in) method select: is used to construct a new smaller sequence containing each element for which the predicate (contained within the block surrounded by []) evaluates to true. The reference to self refers to the object processing the message, and the construct new is used to dynamically create a new object of the specified class. Thus, a new sequence is created and all selected elements are added to it. Once the three subsequences have been created, lowerPart and upperPart are sent messages to sort themselves. After these sorts are complete, the three sequences are used via updateFrom:and:and: to update the original sequence in place.

The quickSort is highly polymorphic. It is defined here for the class Sequence. All subclasses of this class (for example, lists, arrays, and files) will inherit the quickSort method. Through methods of their own or through inheritance, the subclasses can interpret the messages size, first, addAll:, updateFrom:and:and:, and select:. The contents of these sequences can be any type of object as long as the object understands the messages <, =, and >. Thus, one definition suffices for sorting files of numbers, arrays of names, lists of personnel records, etc. This combination of polymorphic methods and inheritance provides much of the power of the object-oriented paradigm.

Smalltalk[1] is the preeminent object-oriented language, both historically and in the extent of its compatibility with the paradigm. It fully supports encapsulation, inheritance, and message-passing. Everything in the language is modeled as an object. In addition to the basic language, the Smalltalk environment includes a large library of predefined objects for basic data structures like lists, arrays, and collections.

Although most object-oriented languages support the class-based model presented here, an alternative approach supports a bottom-up view of objects. In this view, an object's characteristics and capabilities are determined not by the class of which it is a member, but rather by the object it most closely resembles (termed the *prototype*). The Self language is an example of this prototype-based approach.[2]

References

1. A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, Mass., 1983.
2. D. Ungar and R.B. Smith, "Self: The Power of Simplicity," *Proc. OOPSLA'87*, ACM Press, New York, 1987, pp. 227-242.

In its pure form, the imperative paradigm supports only simple commands that modify the store and carry out conditional and unconditional branching. Even when a simple form of procedural abstraction is added, this model remains basically unchanged. Parameters are aliases for a portion of the store, no values are returned, and a procedure alters the store through its parameters and/or direct global references.

Object-oriented. With the imperative paradigm, the conceptual model is a single store into which abstract data values are represented and on which one or more procedures are applied. Each procedure deals directly with the stored representation. The object-oriented paradigm retains much of this model, but procedures operate on abstract values, called objects, rather than on stored representations. As a result, this paradigm requires the capabilities of defining new objects composed of existing objects and of manipulating them by defined procedures (called methods). Object-oriented programming first defines suitable objects for the problem at hand, then uses these objects to describe step-by-step operational sequences.

Manipulation of abstract values rather than concrete representations requires respect for the encapsulated state of objects (that is, the right of objects to define their concrete representations and to perform all manipulations upon such representations). This is accomplished by sending messages that describe the desired manipulations and leaving it to the objects to perform them. Objects, which are implemented via other subobjects, use operational sequences to alter their internal representations. Such sequences include sending messages to their subobjects.

This process recurses until at some level the objects and the methods defined on them are primitive. Thus, sorting involves sending an object a message to sort itself. The message sender does not care how the object sorts itself, only that it gets sorted.

While the distinction between directly manipulating concrete representations and applying methods to objects may seem subtle, the impact on programming is not. Object orientation encourages thinking about individual concepts rather than a single global concept. Each object is conceived and implemented as self-contained.

Object orientation encourages thinking about individual concepts rather than a single global concept.

We have thus far described only encapsulation — the mechanism for enforcing data abstraction. Inheritance is a second characteristic associated with object-oriented paradigms. It is based on the concept of object classes. A class is the definition of an object from which instances of the definition are created. Inheritance allows rapid definition of a new object class from the concrete representation and methods of an existing class. The new class includes all of the methods defined on the inherited representation, as well as any new concrete representations and new or revised methods added to it.

The sidebar example defines sorting for all sequences in such a way that specific subclasses, such as lists, inherit the sorting method. Under inheritance, subsequent modifications to the original class (called the superclass) are reflected in the new subclasses.

A third characteristic of object-oriented paradigms is message-passing. We come to think of objects as active entities that send messages to one another. This view encourages us to think of decomposing problems into players who accomplish a task cooperatively. The specification of any one player should be relatively uncomplicated, with players organized into teams for more complex tasks. Several languages have explicitly modeled this player concept.[2]

Some languages that support the object-oriented paradigm are extensions of languages that tend to be primarily imperative. These extensions do not support objects as active entities that receive messages. Rather, they continue to invoke procedures or functions that pass objects. To support this model, procedures must be polymorphic — that is, multiply defined with particular invocations determined by examining the types of the parameter objects. While the message-passing mechanism of the object-oriented paradigm is computationally equivalent to the procedure-call approach of these extended languages, it leads to a very different way of looking at problem solutions. There is a conceptual difference between searching through procedures — all of the same name — for a match of the particular parameter types and sending a message to a particular object that knows only one such method. In the former case, the main program controls all the work, while in the latter, each object has full responsibility for correctly handling requests made directly to it.

Functional (operational). The functional paradigm is based on the mathematical model of functional composition. In this model, the result of one computation is input to the next, and so on until some composition yields the desired result. There is no concept of a location that is assigned or modified. Rather, there are only intermediate values, which are the results of prior computations and the inputs to subsequent computations. For convenience, these intermediate values can be given names. There is no form of command, and all functions are referentially transparent.

Functional programming includes the concept of functions as first-class objects. This means that functions can be treated as data (that is, they can be passed as parameters, constructed and returned as values, and composed with other forms of data).

Application developers conceive the solution as compositions of functions. For instance, to sort a list, we might conceive the solution as a concatenation of some smaller lists, each of which is already sorted. This reduces the problem to selecting the smaller lists.

The way functions are specified can vary. In particular, we can specify them operationally, or mathematically without control sequencing. Here we address only the operational case. The mathematical case will be discussed under "Definitional paradigms."

The operational approach explicitly controls the order in which computations occur. For instance, in sorting a list, we might first determine if the list is empty and proceed only if it is not. This

explicit ordering of computations causes an overspecification that characterizes operational approaches and leads to the discussion of parallel versus sequential control in the next subsection.

In practice, languages supporting this operational form of the functional paradigm often include imperative constructs such as multiple assignment, which destroy the non-side-effecting nature of the paradigm and force further sequencing considerations into the construction of programs. Such compromises are in part historical. They date back to a time when creating efficient solutions required modeling a machine's store. Also, most imperative and object-oriented languages have adopted some form of function, but their reliance on side effects and their lack of functions as first-class objects prevent them from fully supporting functional programming.

Sequential versus parallel control flow. Constructing parallel operational programs requires extending the conceptual models presented thus far. This extension is required to handle the overspecification forced by explicit control sequencing. For example, in each of the two quicksort examples we have described, the two sublists are sorted sequentially and in a specified order, when they might be sorted simultaneously, or at least in any order. The problem is that once we define a rigorous ordering, the system must follow it. When programmers realize that they are overstating control, they may desire a means of telling the system when it is safe to violate this ordering. This leads to the extended models discussed here.

While parallel considerations extend the paradigms we have discussed, the efficient use of these extensions often leads to algorithms that are more than relaxed sequential algorithms. In particular, the neighborhood_sort algorithm developed in the sidebar on asynchronous paradigms assumes parallel evaluation from its very conception. Without these extensions to our paradigm mode, we would not likely derive the solution simply by eliminating sequential control from an algorithm that was previously stated sequentially. Thus, we consider these parallel-extended paradigms to be paradigms themselves.

Parallel programming languages historically have followed one of three approaches:

(1) automatically detect parallelism in an otherwise sequential language,

(2) add mechanisms that directly mimic the parallel operations of a particular machine, or

(3) add general mechanisms for expressing the parallelism in the problem.[3]

With the first option, the compiler determines what parts of the application can be executed in parallel. If parallelism was not a paradigm, this might end up the most reasonable approach to parallel programming. With the second option, the language provides data and program structures that directly reflect the architecture of the machine. The resulting software is finely tuned for a particular machine and may not be suitable for any other architecture. With the third option, the language provides data and program structures that let the software developer express the parallelism inherent in a problem without reference to the hardware. For example, the language will provide the most basic operations — ones that allow for fine-grained parallelism. Such language constructs encourage thinking of the solution in terms of the parallelism inherent in the problem.

Parallel processes are either *interfering* or *noninterfering*. Interfering processes have at least some potential for

Functional paradigm (operational)

```
define function quicksort (lst)
  if lst is null then
     nil
  otherwise
     append (quicksort (choose_members (function (<), pivot (lst), lst)),
             choose_members (function (=), pivot (lst), lst),
             quicksort (choose_members (function (>), pivot (lst), lst)))

define function pivot (lst)
  first (lst)

define function choose_members (function, val, lst)
  if lst is null then
     nil
  otherwise
     if function-call (function, first (lst), val) then
        append (list (first (lst)),
                choose_members (function, val, rest (lst)))
     otherwise
        choose_members (function, val, rest (lst))
```

In this operational-style functional rendition of quicksort, the sequence of the conditions given is important. For example, if the test for the null list in choose_members were not first, the program would either fail to terminate or end in error, depending upon the implementation of the functions <, =, and >.

"Pure" Lisp[1] represents this paradigm best. However, the numerous enhanced versions of Lisp in use today, including Common Lisp, might equally well support imperative programming. These languages include side-effecting destructive modification, fixed storage data types (such as arrays and objects that are manipulated primarily by side-effect), and control-flow constructs (for example, iteration and exception handling).

Reference

1. J. McCarthy, "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part 1," *Comm. ACM*, Vol. 3, No. 4, Apr. 1960, pp. 184-195.

affecting the computation of each other. The primary problem associated with interfering processes is restricting their interference in controlled ways so that they compute predictably. Interfering processes tend to be coarse-grained (that is, larger) processes and are often heterogeneous due to the complexities of coordinating interference.

Noninterfering processes are simpler to specify, but usually much more fine grained. For instance, adding two $n \times m$ matrices requires $n * m$ additions. Each of these additions is noninterfering, but perhaps just this one addition operation is to be executed in parallel and the execution is then to be sequentialized again. Because of this fine-grained aspect, noninterfering processes tend to resequentialize often, behaving in a rather synchronous manner.

We will refer to these two approaches as asynchronous and synchronous parallelism.

Asynchronous parallelism. A simple algorithm for sorting in parallel divides the list into N sublists, sorts each in parallel, then merges the results. The individual sorts can either use a sequential algorithm or recursively subdivide again into N sublists, sort and merge, etc. To handle the merge, we might have each process repeatedly attempt to access a common merged list, where access is granted to only one process at a time. Once the process gains access, it merges its sorted list into the already merged list (which is initially empty), generating a modified merged list. The process then releases control of the merged list and terminates.

This sort has the properties of an asynchronous approach: Each process operates asynchronously on some possibly sizeable, probably heterogeneous task during which it explicitly coordinates its interactions with other processes to prevent interference. This algorithm is conceptually distinct from any of the previous algorithms; it does not result from a simple substitution of parallel for serial control sequencing in those algorithms. Carriero and Gelernter[4] discuss asynchronous parallel programming in detail.

Synchronous parallel. A straightforward approach to a synchronous parallel sort would be to use the quicksort algorithms and simply extend the model to allow the three sublists to be processed in parallel. However, freed of the constraints of sequentiality, we might imagine other approaches, such as comparing simultaneously all adjacent pairs and reversing all pairs that are out of order. By alternating between odd and even pairs, the list will eventually sort. At each step where adjacent pairs are processed in parallel, such processing is noninterfering. The regularity of the data enables the same operations to be applied in parallel with all processes acting in unison.

Asynchronous paradigm

```
sort (list)
  if ( length (list) <= 1 )
     list
  otherwise
     sublist_size := length (list) / NUM_PROCESSES
     forall i <= NUM_PROCESSES
        fork (sort_process (THIS_PROCESS_ID,
                              sublist (i, sublist_size, list)))
     list := []
     repeat
        receive (sorted_list)
        merge (list, sorted_list)
     send (parent_process_id, list)

sort_process (parent_process_id, list)
  sort (list)
  send (parent_process_id, list)
```

In this example, sort divides the sorting task by creating child processes and splitting the initial list among them. Each child process operates in parallel with the parent as a separate entity, ultimately returning the sorted sublist to the parent via a message. The actual sort used to accomplish this task could be any sort algorithm, or each child process might further delegate and make use of other asynchronous processes to accomplish this task.

Send sends a message to a process specified by the sender of the message. The message is sent without blocking subsequent processing on the sender's part. Receive receives a message. If no message is waiting, the receiver process is suspended until a message arrives. A process can only service one message at a time. The system queues up any additional messages that may arrive. Thus, as the individual sort_processes accomplish their individual tasks, only one list at a time will be received and merged into the final list.

Many approaches have been developed to facilitate cooperation, synchronization, and/or communication between asynchronous processes. These mechanisms are designed with course-grained parallelism in mind. Linda[1] is an example of a language that adds such facilities to base languages like C and Fortran.

Reference

1. N. Carriero and D. Gelernter, "Linda in Context," *Comm. ACM*, Vol. 32, No. 4, Apr. 1989, pp. 444-458.

Definitional paradigms

In definitional paradigms, a program is constructed by stating facts, rules, constraints, equations, transformations, or other properties about the solution value set. From this information, the system must derive a scheme including an evaluation ordering for computing a solution. There is no step-by-step description of how to reach the solution. These paradigms allow variables, but

not as repositories of state information — rather, as convenient names for intermediate values. They usually include variations of single assignment.

Since definitional paradigms attempt to specify the solution value set without necessarily specifying how to compute a solution, in principle this approach eliminates the need to prove that the computed value set is the solution value set. In practice, the situation is often more complex. While many of the paradigms have no control sequencing and no side effects that require a notion of state, solutions are still often stated as constructions rather than specifications. This raises the question of whether or not these constructions produce the desired solutions. In addition, some definitional paradigms have difficulties that are resolved by working operational techniques back into these paradigms.

Synchronous paradigm

```
neighborhood_sort (list)
index
   ODD = { the set of odd array indices }
   EVEN = { the set of even array indices }
var
   temp : parallel indexed array [1..MAX]
   n : integer

   for n := 1 to (MAX div 2) do
      if list[ODD] > list[ODD + 1] then
         temp[#] := list[#]
         list[#] := list[# + 1]
         list[# + 1] := temp[#]

      if list[EVEN] > list[EVEN + 1] then
         temp[#] := list[#]
         list[#] := list[# + 1]
         list[# + 1] := temp[#]
```

This synchronous sort algorithm[1] has several characteristics that are typical of synchronous languages. The list parameter passed into the routine is an array that can be accessed in parallel. The index references, EVEN and ODD, describe a subset of the array range.

The if-then can operate in parallel on every element in the array or on some subset of them. The first comparison in this algorithm is the simultaneous comparison of the ODD array elements with the corresponding ODD + 1 elements. The # symbol within the body of the if-then statement represents the set of indices for which the comparison is true. The operations within the body of the if-then statement are then performed in parallel for the set of indices that passed the comparison.

The end of the body of the if-then statement becomes a synchronization point. A similar process then takes place for the EVEN array indices.

Most languages designed to support synchronous parallelism are extensions to an existing language base. Many of these extended languages are designed around a specific machine architecture. Others provide data and program structures that allow the programmer to express the parallelism inherent in the problem without exploiting a particular architecture. For example, Actus[1] and Paralation Lisp[2] provide architecture-independent constructs.

References

1. R.H. Perrott, *Parallel Programming*, Addison-Wesley, Reading, Mass., 1987.
2. G. Sabot, *The Paralation Model*, MIT Press, Cambridge, Mass., 1988.

The resulting paradigms and supporting languages are not truly definitional. We refer to these tainted approaches as pseudodefinitional. The pseudodefinitional approaches discussed in this section are *functional*, *transformational*, and *logic*.

Some paradigms do not have the difficulties that require reintroducing control sequencing. These are *form-based*, *dataflow*, and *constraint-programming*. The form-based and dataflow paradigms avoid these difficulties primarily through restrictions on the form of equations allowed and the dependency-driven evaluation models employed. The constraint programming paradigm avoids the difficulties by making no restrictions; however, its generality also limits the use of this paradigm.

In principle, definitional paradigms are not inherently serial or parallel because they do not address control sequencing and thus do not alter the natural parallelism of algorithms. However, the pseudodefinitional paradigms require at least a limited degree of sequencing and thus have parallel and serial versions. The techniques associated with these parallel approaches and their impact on the corresponding paradigms are similar to those already discussed for operational paradigms.

Functional (definitional). The functional paradigm attempts to match the mathematical model by expressing functions as mathematicians do. For instance, given the mathematical definition

$$f(x) = \begin{cases} \text{expr}_1(x), \ \text{cond}_1(x) \\ \text{expr}_2(x), \ \text{cond}_2(x) \\ \quad \cdots \\ \text{expr}_n(x), \ \text{cond}_n(x) \end{cases}$$

a function f is interpreted as $\text{expr}_i(x)$, if exactly one guard condition $\text{cond}_i(x)$ holds; otherwise, it is not well defined. As long as exactly one of the conditions is valid for any particular value of the input domain, we need not specify the order of evaluation of the guards.

The distinction between the functional paradigm as discussed in the operational section and here hinges on whether or not sequencing is explicitly specified. With the operational style, the software developer is responsible for the complete sequencing of instructions, including proper termination. By contrast, the definitional style provides for termination without requiring explicit control

sequencing. The differences, while subtle, are significant in framing our conceptual model of programming. In the operational style, we approach the problem as a construction in which we describe a sequence of steps that will use functional composition to compute the desired result. In the definitional style, we approach the problem as a collection of disjoint transformations that, taken collectively, define a computational function.

Often this mathematical definitional model utilizes lazy evaluation (also called nonstrict evaluation). Simply stated, in lazy evaluation the arguments to a function are not evaluated until and unless they are individually needed. This is in contrast to conventional evaluation, called eager evaluation, which evaluates all arguments prior to invoking the function call. Besides making it possible to evaluate arguments selectively, a feat otherwise accomplished only via special primitives, lazy evaluation also provides a natural means of dealing with infinite structures. Since evaluation is performed only when there is a need for a particular value, and then only to obtain that one value, definitions of functions that generate infinite sequences can be used in place of the actual sequence as long as only a finite number of the sequence's values are ever actually required.

If we in any way relax the requirement that exactly one guard holds, then the order of evaluation is again important. It is this last point that leads to the designation "pseudodefinitional." It is statically undecidable whether or not any particular set of guards is disjoint; thus, most functional paradigms define the evaluation order and leave it to the programmer to ensure either that the guards are disjoint or that the conditions are properly ordered.

Hudak[5] presents an excellent in-depth discussion of functional languages.

Transformational. Transformational paradigms employ pattern-matching and term-rewriting techniques. Evaluation proceeds by repeatedly applying transformation rules to derive from an initial input token sequence a series of transformed token sequences leading to the solution token sequence. For instance, beginning with the word *Jelly* and transformation rules that replace *J* by *H* and *y* by *o*, the system might construct either the derivation Jelly → Helly → Hello or the derivation Jelly → Jello → Hello. The programming task specifies the necessary transformation rules, leaving the system to choose and apply them.

A transformation rule consists of a guard and an action. If the guard is true (applicable) given a particular token sequence, then the action can be applied to yield a new token sequence. At each step in the derivation, a single transformation rule is selected from those whose guards are true. A derivation continues until no rule is applicable.

To preserve correctness, the programmer must ensure at each step either that only a single rule is applicable or that if more than one rule is applicable, the final solution does not depend on which rule is chosen. For instance, in the derivation from Jelly, both rules can be applied to the initial token sequence. Regardless of which rule is chosen first, the other will be applied later and the resulting token sequence will be the same.

An alternative to forcing the programmer to guarantee order independence is to concede an order for purposes of determining the applicability of rules. For instance, allowing that the rules will be searched for applicability in a known order and that the first applicable rule will be used lets the programmer order the rules such that when more than one rule might apply, the desired rule will be encountered first. This rule simplifies the programming problem, but if order is given significance in this way, the paradigm is no longer completely definitional.

Functional paradigm (definitional)

```
qsort ([X|Xs])
   =  qsort (sublist (<, X, Xs))
      || sublist (=, X, [X|Xs])
      || qsort (sublist (>, X, Xs))
qsort ([])= []

sublist (f, val, [X|Xs])
   =  [X|sublist (f, val, Xs)] if f (X, val)
   =  sublist (f, val, Xs)      otherwise
sublist (f, val, []) = []
```

This definitional-style version of quicksort shows two ways of expressing alternative interpretations of a function. First, where simple matching of parameters is possible, as in the treatment of the null list, separate formulations of the function can be used. Second, where such parameter matching fails, guards are employed to distinguish alternative expressions.

The first statement says that the result of qsort, when given a list of element X and sublist Xs, is the result of appending (||) the sorted sublist of elements less than X to the sublist of elements equal to X and to the sorted sublist of elements greater than X.

The example demonstrates the equational and pattern-matching aspects of definitional functional languages. These aspects allow the function definitions in the algorithm to be stated in any order. However, there is an evaluation ordering assumption associated with the use of the otherwise statement in sublist. This assumption could be removed by restating the previous guard condition in the negative.

A good example of a modern functional language is Haskell,[1] which represents an effort by the functional programming language community to reach a consensus and thereby encourage wider use of functional languages.

Reference

1. P. Hudak, S. Peyton Jones, and P. Wadler, "Report on the Programming Language Haskell, a Nonstrict Purely Functional Language Version 1.2," *ACM SIGPlan Notices*, Vol. 27, No. 5, May 1992.

Transformational paradigm

```
qsort ([]) { [] }
qsort ([P|Xs]) { qsort (small (Xs, P)) || [P| qsort (big (Xs, P))] }

small ([], P) { [] }
small ([X|Xs], P) { X < P; small_aux (X, Xs, P) }
true; small_aux (X, Xs, P) { [X| small (Xs, P)] }
false; small_aux (X, Xs, P) {small (Xs, P) }

big ([], P) { [] }
big ([X|Xs], P) { X >= P; big_aux (X, Xs, P) }
true; big_aux (X, Xs, P) { [X| big (Xs, P)] }
false; big_aux (X, Xs, P) { big (Xs, P) }
```

A transformational system consists of a set of transformation rules. Each rule is made up of a head followed by a body written in braces {}. The system finds a subexpression that matches the head of a rule. That subexpression is then rewritten by substituting the body of the rule in place of the subexpression's head.

Heads may contain variables that are capable of binding portions of the matched subexpression. Such bound variables can then be used in expanding the body of the rule. Expanding the body of a rule consists of either replacing variables with their bound values or replacing them with the result of some computation on their bound values, such as the sum of two bound variables. The process is repeated until the subject expression contains no reducible subexpressions.

Beginning with an original expression, qsort ([5, 6, 4, 1, 3, 9]), the system will search for a rule whose head matches. Initially, only the second rule will match, resulting in a transformed expression qsort (small ([6, 4, 1, 3, 9], 5)) || [5 | qsort (big ([6, 4, 1, 3, 9], 5))]. Using this expression, the system will then look for yet another applicable transformation rule; for instance, it might apply the eighth rule, yielding qsort (small ([6, 4, 1, 3, 9], 5)) || [5 | qsort (6 >= 5; big_aux (6, [4, 1, 3, 9], 5))]. This expression reduces to qsort (small ([6, 4, 1, 3, 9], 5)) || [5 | qsort (true; big_aux (6, [4, 1, 3, 9], 5))]. Then we can apply the ninth rule to get qsort (small ([6, 4, 1, 3, 9], 5)) || [5 | qsort ([6 | big ([4, 1, 3, 9], 5)])] and so on until only [1, 3, 4, 5, 6, 9] remains and no rule applies.

Bertrand[1] is a transformational programming language based on augmented term rewriting. Augmented term rewriting is an extension to term rewriting; it supports bindings, objects, and types. Because much of Leler's work involves the use of these transformational techniques to build constraint-satisfaction systems, Bertrand is often associated with constraint programming as well.

Reference

1. W. Leler, *Constraint Programming Languages, Their Specification and Generation,* Addison-Wesley, Reading, Mass., 1988.

Logic. The logic paradigm assumes that we begin with a set of known facts, such as "Tom is a father," and a set of rules that allow deduction of other facts. For example, "For all X, if X is a father, then X is male" allows us to deduce that Tom is male. Thus, logic programming from the programmer's perspective is a matter of correctly stating all necessary facts and rules.

Kowalski[6] pioneered the logic paradigm. To date, most logic programming languages have been based on Horn clauses, a subset of first-order predicate logic. The clausal notation of predicate logic combines variables, constants, and expressions to express conditional propositions such as

$$\text{Grandparent}\ (x, z) \leftarrow \text{Parent}\ (x, y),\ \text{Parent}\ (y, z)$$

which states that x is the grandparent of z if x is the parent of y and y is the parent of z. Horn clauses are a restricted form of predicate logic with exactly one conclusion in any one clause.

Logic programming then is a statement of only the logic component of an algorithm. The system derives the control sequencing component. By separating logic from control, the program becomes merely a formal statement of its specifications. Hence, its correctness should be easily provable. In fact, since logic programming can be viewed as automated theorem proving, all logic programs are "correct" by definition. Of course, even if the programs are correct when compared to their written specifications (facts and rules), the question still remains of whether or not the specifications correctly reflect the true problem.

Evaluation starts with a goal and attempts to prove it by either matching it with a fact or deducing it from some rule. A goal is deduced from a rule if bindings can be found for all free variables such that, once substituted, all antecedents can be proved. These antecedents become new subgoals that must be matched with facts or proved via other rules. The process terminates successfully when all subgoals have been proved. The final solution is determined by applying the bindings developed along the way to any free variables in the initial goal.

Evaluation as just described is purely definitional. It assumes that whenever a rule is selected, either there is only one possibility or the one needed to derive the solution is somehow selected. A solution is found if a suitable set of rules and substitutions exists such that applying the substitutions to the rules produces a set of grounded rules (that is, a set with no free variables) sufficient to deduce the goal from known facts.

A process known as unification deterministically develops substitutions for free variables; however, there is no similarly deterministic algorithm for selecting rules. This leads to difficulties. Implementations must approach evaluation with a breadth-first search, where each

rule selection point initiates separate independent computations, one for each possible applicable rule. If a solution exists, the evaluation algorithm will eventually terminate, but this approach can be very inefficient.

This inefficiency has led to the development of many variants of logic programming, each with different evaluation algorithms. Popular among these variants is to impose an order on the selection of rules and to employ a depth-first search with a backtracking algorithm. When a rule must be selected, this algorithm selects the "first" one. If it eventually leads to a dead end, the "next" one is selected, and so on until all possibilities have been tried. At each failure, backtracking is performed for the most recent decision. If this fails because there are no more possibilities to try, then further backtracking occurs. When all possibilities fail, no solution exists.

By careful ordering of facts and rules, a programmer can greatly impact performance, as well as termination. Of course, this changes the paradigm to pseudodefinitional. Logic programming is often used to refer to the pure form discussed, while the pseudodefinitional version is referred to by the name of the language, for example, Prolog programming.

There are other nondefinitional features found in most implementations of logic-programming variants, most notably cuts. The cut is an extra-logical device that creates a barrier inhibiting the backtracking mechanism of the theorem prover. Cuts were introduced to improve efficiency. They prevent re-evaluation of rules during backtracking. In addition, they are often used to specify logic.

The logic paradigm is intended for general-purpose programming. Indeed, pseudodefinitional logic is used for production programming in industry, where it is very well-suited to certain types of problems. These include backtracking search problems that may require multiple solutions; problems that are naturally expressed in terms of production rules, such as natural-language translation; and executable specifications for rapid prototyping.

Form-based. The form-based paradigm, as well as the dataflow paradigm in the next subsection, defines a computation via a collection of equations. These equations avoid any expression of control sequencing, but a restriction placed on the form of these equations guarantees that the system can easily derive an evaluation order sufficient to compute a solution. For the form-based paradigm, this restriction is that all equations are of form $X = f(Y_1, Y_2, \ldots, Y_n)$ where all of the Y_i's must be computable independently from X.

In the form-based paradigm, the programmer designs a form including formulas that ultimately compute values. Each formula corresponds to the right-hand side of an equation, that is, $f(Y_1, Y_2, \ldots, Y_n)$. Typically, we associate a description of the visual display of the formula's value with each formula. A cell is a formula and its associated display description. In constructing formulas, cell references to other cell values can be formed by pointing at the corresponding cells.

The use of forms in this manner is analogous to the way we use familiar forms, such as tax forms. Designing a form implies a two-dimensional syntax. In practice, the form-based paradigm is

Logic paradigm

```
sort (Xs,Ys) ← permutation (Xs,Ys), ordered (Ys)

permutation ([], [])
permutation (Xs, [Z|Zs]) ← select (Z,Xs,Ys), permutation (Ys,Zs)

ordered ([X])
ordered ([X, Y|Ys]) ← X ≤ Y, ordered ([Y|Ys])
```

The permutation sort[1] states two facts (lines 2 and 4) and three rules. These facts and rules characterize a fully sorted list, thus illustrating the declarative nature of the logic paradigm. Collectively, they state:

- The list Ys is a sorted version of the list Xs if Ys is a permutation of Xs and Y is ordered (line 1);
- An empty list is a permutation of itself (line 2);
- Given a list Xs and a list consisting of an element Z followed by a sublist Zs, if Z can be selected from Xs leaving a remainder Ys that is a permutation of Zs, then Xs is a permutation of [Z|Zs] (line 3);
- A list with only one element is ordered (line 4); and
- Given a list that starts with the elements X and Y, followed by the list Ys, if X ≤ Y and [Y|Ys] is ordered, then the original list is ordered (line 5).

The supporting code for select requires only one additional fact and one additional rule:

```
select (X, [X|Xs], Xs)
select (X, [Y|Ys], [Y|Zs]) ← select (X, Ys, Zs)
```

Although the Prolog language[1,2] has become synonymous with logic programming, it actually includes a specific problem-solving strategy and a number of features not true to the logic paradigm. While its pure logic features provide the advantages of the logic paradigm, Prolog can be used in an entirely different way that resembles an operational paradigm.

References

1. L. Sterling and E. Shapiro, *The Art of Prolog*, MIT Press, Cambridge, Mass., 1987.
2. W.F. Clocksin and C.S. Mellish, *Programming in Prolog, Third Revised and Extended Edition*, Springer-Verlag, Berlin, 1987.

Form-based paradigm

To Sort:	40	30	20	10
	= MIN (above, rightAbove)	= MAX (above, leftAbove)	= MIN (above, rightAbove)	= MAX (above, leftAbove)
	= above	= MIN (above, rightAbove)	= MAX (above, leftAbove)	= above
	= MIN (above, rightAbove)	= MAX (above, leftAbove)	= MIN (above, rightAbove)	= MAX (above, leftAbove)
Sorted:	= above	= MIN (above, rightAbove)	= MAX (above, leftAbove)	= above

To Sort:	40	30	20	10
	30	40	10	20
	30	10	40	20
	10	30	20	40
Sorted:	10	20	30	40

The top form displays each cell's formula, and the bottom form displays the values obtained from evaluating these formulas. The value of each cell is defined by its own formula. Only two distinct formulas are needed to fill in the form: the minimum of a pair of elements and the maximum of that pair. The *n* elements are initially selected pairwise and the minimum of each pair is placed before the maximum. Next, the elements are paired with the neighbor on the other side, and the process is repeated. This continues until the sequence is sorted.

In practice, the programmer merely specifies the first pair of formulas and propagates them appropriately. The actual mechanisms to do this vary with individual supporting languages. Some languages provide a replication facility, while others let a formula serve as a general definition for a group of cells.

Although it appears that this solution may only apply to a fixed number of cells, this is not the case. The form-based paradigm does not necessarily restrict the form size to a fixed number of cells. One common approach defines an infinite number of rows, relying on lazy evaluation to ensure that only cells with actual values are evaluated. Examples of form-based languages include Forms/3[1,2] and Plane Lucid.[3]

References

1. M. Burnett and A. Ambler, "Generalizing Event Detection and Response in Visual Programming Languages," Tech. Report, TR 91-02, Computer Science Dept., Michigan Technological Univ., Houghton, Mich., 1991.
2. G. Viehstaedt and A. Ambler, "Visual Representation and Manipulation of Matrices," *J. Visual Language and Computing*, Vol. 3, No. 3, Sept. 1992.
3. W. Du and W. Wadge, "A 3D Spreadsheet Based on Intensional Logic," *IEEE Software*, Vol. 7, No. 3, May 1990, pp. 78-89.

supported only by visual programming languages. Further, the distinction between the user and the programmer tends to blur under this paradigm, because the concept of defining the form as well as filling one in are comfortable tasks for users. Supporting languages tend to be highly interactive, fostering an "experimental" approach to programming.

A natural practice of the form-based paradigm is the collection of partial results or values down and across the form, approaching a solution at the bottom and right of it. The neighborhood sort in the sidebar shows an example of this two-dimensional approach. It builds up the intermediate results on successively lower rows of the form, until eventually the final result is achieved on the bottom row.

Because a form is a collection of formulas that are functional expressions, the form-based paradigm has a lot in common with the functional paradigm. However, the collection of formulas in the form-based paradigm allows for formula dependencies whose graph is any directed graph, not just the hierarchical graphs of pure functions.

Dataflow. In the dataflow paradigm, streams of data flow like fluids through a network of nodes, each of which performs a computation that consumes the data flowing into the node and produces new data that flows out of the node. The programmer specifies only the node equations. The evaluation order is implied by data dependencies in the node equations. The system schedules computations whenever all the input data to a node is available.

Dataflow programming can be demonstrated graphically with dataflow diagrams such as the one in Figure 1. In the figure, data elements, called tokens, enter a node representing addition. The computation consumes these input tokens and produces the sum as a new token.

Computations are composed by using node outputs as inputs to new nodes. Like the form-based paradigm, the dataflow paradigm requires equations to be expressed in the form $X = f(Y_1, Y_2, \ldots, Y_n)$; however, the codependency restric-

tion is loosened. In the dataflow paradigm, evaluation is viewed as a continuous process in which each equation is computed each time a new set of input values is available. Thus, a variable represents a stream of values, one for each time its formula equation is evaluated. Given this model, if X^i represents the ith value in the stream for X, then the dataflow restriction is that X^i can depend on X^j, directly or indirectly, only as long as $i > j$. This prevents circular dependencies of the type that would inhibit computation. (Note that if we reduce all streams to length one, the dataflow restriction is equivalent to the form-based restriction.)

We can view the notion of dataflow streams that flow into and out of nodes as sequences of values along a time dimension. Conceptually, this allows a node's output arc to bend around and serve as an input stream to the same node (that is, it allows feedback in the dataflow graph). This time dimension notion is readily captured in graphical dataflow languages because the relations between input and output sequences are explicitly shown via the arcs between the nodes. Some of the textual dataflow languages deal with these relations implicitly rather than explicitly, requiring special constructs to access specific previous values in a controlled way.

While this restriction allows for feedback, and thus iteration, programmers usually employ recursion. This is completely in keeping with the dataflow concept of routing data between nodes, where each recursive invocation is logically another node in the dataflow graph.

Two execution models are used in dataflow programming. Both are strictly dependency driven. In the data-driven model, a node may execute as soon as all required input is available. This model encourages parallelism, but it can cause overproduction of data. In the demand-driven model, a node does not execute unless all required input is available *and* its output is requested. While overproduction of data cannot occur using demand-driven evaluation, parallelism may be impaired because nodes that could be computing will wait until a demand for their output occurs.

The dataflow paradigm has some similarities to the functional paradigm. Much of the literature discusses it as a variant of functional programming and many textual dataflow languages are based on functional languages. However, from a paradigm perspective, there are significant differences. A functional program is represented by a single equation; a dataflow program is a collection of equations. Dataflow nodes operate on streams of data; thus, they do not obey either eager or lazy evaluation.

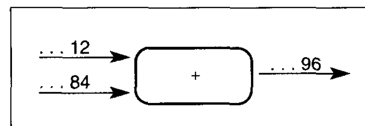
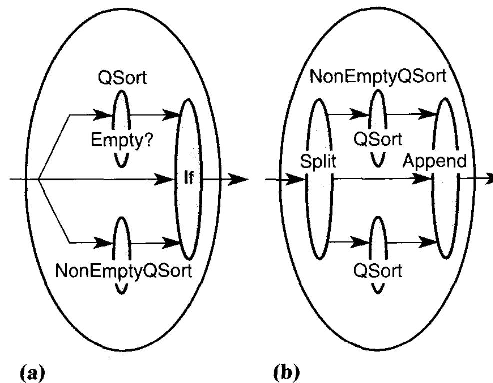


Figure 1. Addition consumes pairs of input tokens and outputs a new token for each sum.

Dataflow does not treat functions as first-class objects (that is, nodes do not construct and produce new functions). Dataflow equations require continuous evaluation, producing a stream of output values rather than just a single value. As with the form-based paradigm, dataflow dependency graphs include all directed graphs, not just hierarchically structured graphs. Philosophically, the dataflow approach defines a program in terms of data flowing through nodes, each of which consumes its input data and produces new output data.

The dataflow paradigm supports parallelism. The data dependencies are only those that are natural to the algorithm. The paradigm makes these dependencies explicit automatically. There is no

Dataflow paradigm



(a) **(b)**

QSort expects a stream of unsorted vectors and produces a stream of sorted vectors. The definition of the QSort node (a) consists mainly of an If node. Whenever a result flowing out of Empty? is true, then the corresponding original vector of values is produced by If and thus by QSort. Otherwise, the result of node NonEmptyQSort is produced. In NonEmptyQSort (b), unsorted vectors are routed through a node Split, yielding three separate vectors of values respectively less than, equal to, and greater than the first (pivot) element. Once split, two of these streams flow into QSort nodes for sorting. The resulting streams and the third stream from Split are then appended to generate the final stream of sorted vectors.

The emphasis on dataflow relations rather than control flow is easily seen in visual representations. This has led to the development of visual dataflow languages such as Show and Tell.[1]

Reference

1. T.D. Kimura, J.W. Choi, and J.M. Mack, "Show and Tell: A Visual Programming Language," in *Visual Programming Environments*, E.P. Glinert, ed., IEEE CS Press, Los Alamitos, Calif., Order No. 1973, 1990, pp. 397-404.

Constraint programming

[x_i where $x_i \le x_i+1$ for all x_i]

A group of elements *x* is sorted if the above constraint is met. A constraint specifies a relation that must be maintained. The programmer is responsible for specifying the relation, and the system is responsible for maintaining it.

If the constraint is not already satisfied, the system is responsible for taking appropriate action to satisfy the constraint. Currently, general constraint-solving techniques are weak, and thus constraint systems tend to be application-domain specific. An example constraint system is the visual language ThingLab.[1] Within its specific problem domain (simulations), ThingLab implements the constraint paradigm in a manner that, while not strictly definitional, successfully encourages a definitional problem-solving approach.

Reference

1. A. Borning, "Graphically Defining New Building Blocks in ThingLab," *Human-Computer Interaction*, Vol. 2, No. 4, 1986, pp. 269-295.

explicit control sequencing; the order of computation can be determined strictly from the interdependencies in the data. This paradigm is often used to research parallel computing, and parallel machines based on dataflow architectures have been proposed.[7] In addition, the dataflow paradigm is naturally expressed visually, making it popular for use in visual programming.

Constraint programming. The central idea of the constraint-programming paradigm is to sufficiently constrain the solution value set such that only the desired solution or solutions are possible. Then, ideally, an intelligent system will employ some means to realize what the solution must be.

To illustrate constraint programming, consider the following program to convert centigrade temperatures to Fahrenheit:

$$F = 32 + 9/5 \times C$$

This program appears to be an equation much like one we might write in other paradigms. However, in other paradigms, such a program can only be used to produce Fahrenheit values from centigrade values. In constraint programming, if the value of *C* is known, it can be used to compute *F*. Likewise, if the value of *F* is known, it can be used to compute *C*. Thus, this program is not a statement of some computation to be performed, but a true equation expressing a relationship between *F* and *C*. Furthermore, it is entirely equivalent to the following program:

$$C = 5/9 \times (F - 32)$$

This ability to solve for any variable, given the rest, is also present to some degree in logic programming and to a lesser extent in pseudologic programming if the rules are carefully defined for multiple uses.

A program is specified by a collection of such constraining equations, called constraints. Given a collection of such constraints, it is up to the underlying system to find a solution that satisfies them. Constraints are not limited as equations are in form-based and dataflow paradigms. Instead, the problem for constraint-satisfaction systems is how to find a solution.

One approach, called equation solving, employs algebraic manipulation. This approach has been used to solve complex equations. For instance, equation-solving techniques can be used when constraints are expressed as a system of simultaneous linear equations.

The most common equation-solving method is based on Gaussian elimination. This method has also been extended to include "slightly nonlinear" simultaneous equations, in which most of the equations are linear. The solutions of the linear equations provide enough information to transform the nonlinear equations into linear ones.

Linear programming is another equation-solving technique that is sometimes used to find an optimal solution to a group of simultaneous linear equations. However, in general, equation solving is limited and can only be applied to specific problem domains.

Another approach is constraint satisfaction. Prominent techniques include local propagation of known states, relaxation, and local propagation of degrees of freedom. The first technique propagates known values to other equations (that is, if we can solve for *A*, then we propagate *A*'s value by substituting the value of *A* for *A* in all equations).

The relaxation technique is often employed in situations involving circularity. A guess is substituted for some variable that is circularly dependent and the computation is continued until the circularity causes the computation of a new value for the variable. The system then interpolates between the initial guess and the computed value, obtains a new guess, and repeats the process. If the difference converges to zero, a solution is reached.

The third technique, propagation of degrees of freedom, temporarily removes parts of a graph to try to solve the remaining graph. The parts chosen for removal are based on their degrees of freedom from dependencies. Once the remaining graph is solved, the removed parts are returned and solved.

The combination of these three constraint-satisfaction techniques is again sufficient for slightly nonlinear equations. For truly arbitrary sets of equations, general constraint-satisfaction techniques are inadequate.

Constraint programming is highly independent of order. Although this fact should make it a good candidate for parallel applications, little research has been done in that area. The most important deficiency of the paradigm results from the weakness of general constraint satisfaction techniques. This keeps it from being used in general-purpose programming. However, domain-specific languages have been successful.

Demonstrational paradigms

When programming under the demonstrational paradigm (also called by-example or by-demonstration programming), programmers neither specify operationally how to compute a value set nor constrain the solution value set. Rather, they demonstrate solutions to specific instances of similar problems and let the system generalize an operational solution from these demonstrations. Individual schemes for generalizing such solutions range from mimicking an operational sequence to inferring intentions. How and what a by-example system should generalize is the major issue for this paradigm and the active area of research. Approaches can be categorized by whether or not inferencing is involved.

Inferential systems attempt to generalize using knowledge-based reasoning. For instance, given the sorting example in the sidebar, the main question to answer is why the particular selections were made at each step. Possible guesses include positional order or value. Given such possibilities, the system may proceed in a number of ways. It might simply pick the guess it weighs most likely. It might maintain all possibilities, asking for further examples that would help to discriminate further. It might ask the user which is meant. The very real possibility remains, of course, that none of the deduced reasons is correct. In this case, the only possibility is for the user to override the system to describe the correct intentions.

One inference-based approach attempts to determine ways in which a given group of data objects are similar, drawing generalizations from these similarities. Another approach is programmer assisted: The system observes actions that the programmer performs; if they are similar to past actions, it attempts to infer what the programmer will do next. Myers[8] discusses the issues of demonstrational systems using inference.

There are two major criticisms of inferential systems. First, if unchecked they may produce erroneous programs that appear to work on the test examples but fail later on other examples. Second, their inferential abilities are so limited that the user must either guide the process by working at a very low level or edit the result at a level that amounts to designing the program. The most successful inferential systems have been in limited areas where the system has application-specific semantic knowledge.

Without inferencing, the problem of how and what to generalize is really the problem of how the user should instruct the system to generalize from concrete examples. There are several approaches for doing this. (For a more complete discussion, see Ambler and Hsia.[9])

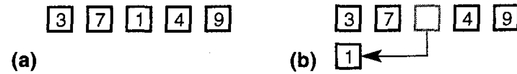
One approach is to work with abstract data rather than specific data. This prevents the user from picking a particular value and operating on it. The user must express constraints on the abstract data (such as, "Pick *X* where *X* is the largest element of the unsorted vector"). Such constraints provide selection criteria. Once a case is suitably constrained, the programmer demonstrates the desired action. In this example, *X* is copied to the new vector (into a constrained position *Y*). This process is repeated for each possible set of constrained values. See Rubin[10] for an example of this abstract data approach.

Working with abstract data has certain limitations. Users tend to work more accurately on a specific example than they do in the general case.[8] The reasons are clear: In the general case, the human must think of every possible situation that could arise, while on a specific example, only the situation at hand must be considered.

One approach to working with concrete data requires the user to demonstrate selections as well as actions. For example, in demonstrating the sort algorithm, the user is required to demonstrate a procedure for making the first selection of an element from the unsorted vector. For instance, the user might indicate that the value is selected by satisfying a predicate that tests for the minimum value of a vector.

A major problem with demonstra-

Demonstrational paradigm



Using the sorting example, a user might proceed as follows. Given a vector with unsorted numbers shown in (a), the user first creates an empty vector of equal size and then begins to copy numbers from the unsorted vector into the new vector. At each step the user selects a number from the unsorted vector and places it in a location in the new vector. In particular, as shown in (b), the user selects unsorted element 3 with value 1 and copies it to new element 1, then unsorted element 1 to new element 2, and so on.

At issue is what knowledge the system was able to extract from this demonstration. On the surface, all we know absolutely is that an operation the user calls "sorting" rearranges vectors of length 5 into a new order corresponding to the third, first, fourth, second, and fifth elements of the original vector. Some approaches to generalizing from this demonstration rely solely on user demonstrations and expect the users to demonstrate algorithms with sufficient generality. Others expect to generalize using inference. An example of a demonstrational language without inference is PT[1] and one with inference is Metamouse.[2]

References

1. Y.-T. Hsia and A.L. Ambler, "Programming Through Pictorial Transformations," *Proc. Int'l Conf. Computer Languages 88*, IEEE CS Press, Los Alamitos, Calif., Order No. 874, 1988, pp. 10-16.

2. D.L. Maulsby and I.H. Witten, "Inducing Programs in a Direct-Manipulation Environment," *Proc. CHI'89*, ACM Press, New York, 1989, pp. 57-62.

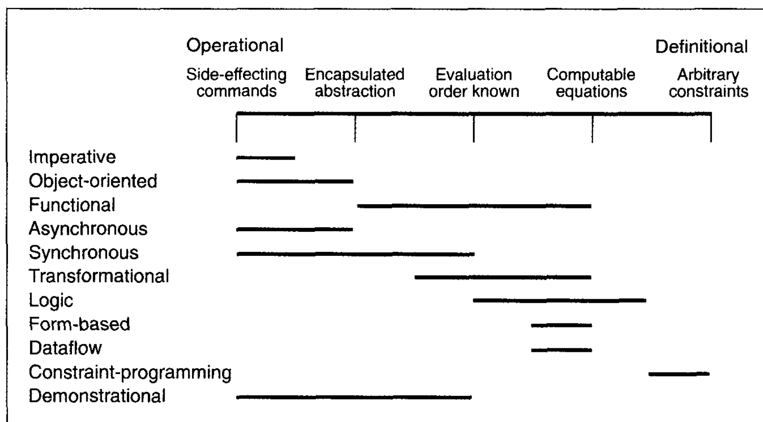


Figure 2. The operational-definitional continuum.

tional systems is knowing when a program is correct. For operational systems, this decision is made by studying the algorithm representation as well as the results of sufficient test cases. For demonstrational systems, the algorithm representation is maintained in some internal representation. Studying this internal representation would somewhat defeat the purpose of these systems. Yet without some representation of the generated algorithm, the correctness decision must be made based solely on the algorithm's performance on specific test cases.

Is demonstrational programming really a paradigm? It does affect the mechanics of programming, but does it encourage a different way of thinking about problems or only a different way of communicating the operational details to the computer? If we look at the most successful approaches (that is, nontrivial programming using concrete examples), we can say that demonstrational programming is generally a concrete bottom-up approach, and as such takes advantage of our natural ability to think concretely. This is opposite to programming in most other paradigms, which tend to encourage a top-down abstract way of thinking about a problem. Thus, we include demonstrational programming as a separate paradigm.

Operational-definitional continuum

We can view the programming paradigms discussed here on a continuum based on the relative degree to which control sequencing is expressed. While this continuum is not the only perspective on programming paradigms, we feel that it helps clarify the relations between various paradigms.

Figure 2 shows the continuum. At the far left is the state-oriented approach to operational paradigms. This approach explicitly controls evaluation ordering. Further, because of side effects and the lack of any form of encapsulated abstraction, it is difficult to remove unnecessary ordering by means of analysis. With encapsulated abstraction, at least some measure of module independence is achievable, allowing some elimination of evaluation ordering. On the other end of the continuum is a totally abstract specification of the solution using arbitrary constraints. For this extreme definitional-paradigm approach, it is difficult to derive any evaluation ordering. A more practical approach involves using computable equations for which evaluation ordering can be readily derived. The middle position denotes the crossover between explicit evaluation ordering and derived evaluation ordering.

Programming paradigms affect our thought processes for solving problems. They provide a framework and determine the form in which we express solutions. A number of programming languages have evolved based on particular programming paradigms. We feel that stepping back from these languages to understand the underlying programming paradigms independently can broaden our programming skills, develop new perspectives on how to use particular programming languages and styles, and perhaps stimulate curiosity about alternatives to our favorite programming paradigms. ■

References

1. R. Floyd, "The Paradigms of Programming," *Comm. ACM*, Vol. 22, No. 8, Aug. 1979, pp. 455-460.
2. W. Finzer and L. Gould, "Programming by Rehearsal," *Byte*, Vol. 9, No. 6, June 1984, pp. 187-210.
3. R.H. Perrott, *Parallel Programming*, Addison-Wesley, Reading, Mass., 1987.
4. N. Carriero and D. Gelernter, "How to Write Parallel Programs: A Guide to the Perplexed," *ACM Computing Surveys*, Vol. 21, No. 3, Sept. 1989, pp. 323-357.
5. P. Hudak, "Conception, Evolution, and Application of Functional Programming Languages," *ACM Computing Surveys*, Vol. 21, No. 3, Sept. 1989, pp. 359-411.
6. R. Kowalski, "Algorithm = Logic + Control," *Comm. ACM*, Vol. 22, No. 7, July 1979, pp. 424-436.
7. Arvind and D.E. Culler, "Dataflow Architectures," in *Annual Reviews in Computer Science*, 1, Annual Reviews Inc., Palo Alto, Calif., 1986, pp. 225-253.
8. B.A. Myers, "Invisible Programming," *Proc. IEEE Workshop on Visual Languages*, IEEE CS Press, Los Alamitos, Calif., Order No. 2090, 1990, pp. 203-208.
9. A.L. Ambler and Y.-T. Hsia, "Generalizing Selection in By-Demonstration Programming," Tech. Report 92-3 (1992), Computer Science Dept., Univ. of Kansas, Lawrence, Kan.
10. R.V. Rubin, E.J. Golin, and S.P. Reiss, "ThinkPad: A Graphical System for Programming by Demonstration," *IEEE Software*, Vol. 1, No. 2, Mar. 1985, pp. 73-79.

Allen L. Ambler is an associate professor of computer science at the University of Kansas. His research interests include visual programming languages, applications of visual

programming including scientific visualization, programming paradigms, and visual software development environments. He led the Visual Programming Languages Design Group in the development of the Forms visual language.

Ambler received a BA in mathematics from the University of Kansas and an MA and a PhD in computer science from the University of Wisconsin, Madison. He is a member of ACM and IEEE.

Margaret M. Burnett is an assistant professor in the Department of Computer Science at Michigan Technological University. Her research interests include visual programming languages, programming languages and paradigms, object-oriented programming, and issues related to the evaluation of functional programming languages. In her recent doctoral dissertation, Burnett developed approaches to abstraction in visual languages, including problems associated with large-scale visual programming.

Burnett received a BA in mathematics from Miami University of Ohio, and an MS and a PhD with honors in computer science from the University of Kansas. She is a member of ACM, IEEE, and the IEEE Computer Society.

Betsy A. Zimmerman is a software engineer for Vortech Data Inc., working on Unix device drivers and X-based application software. Previously, she was a software engineer for General Dynamics, Ft. Worth Division, where she received an Extraordinary Achievement Award for her work on the Alpha distributed operating system. Her research interests include software for mission planning, distributed simulation, distributed operations systems, and system security.

Zimmerman has a BA in mathematics from Harding University and an MS in computer science from the University of Kansas.

Readers can contact Ambler at the Computer Science Dept., Univ. of Kansas, 415 Snow Hall, Lawrence, KS 66045-2192; e-mail ambler@cs.ukans.edu.

Great Work. Great Living.
IBM Burlington, Vermont.

One of the world's most advanced semiconductor operations is what you'll find at IBM's major development and manufacturing facility in Burlington, where continued business growth is matched by a superb living environment. We now have outstanding career opportunities for engineers with the specialized computer skills to make significant impact on RISC microprocessor development.

Logic Design
Responsible for definition, logic design and verification of high performance RISC microprocessors. To qualify, you must possess a BSEE or higher, with an emphasis on Computer Engineering, and be capable of carrying logic design through to physical chip design stage. Minimum of 3 years in logic/chip, CMOS and VLSI design required. RISC experience is key. Background in microprocessor and multiprocessor design desirable.

Circuit Design
Will design CMOS circuitry for RISC-based microprocessor functions. Includes custom SRAM cache design, complex logic dataflow circuitry, random logic, IO, clocking and other circuitry in custom microprocessor layouts. Requires BSEE or higher with emphasis on Computer Engineering or Circuit Design. Ability to design complex CMOS or Bi CMOS circuits and perform circuit analysis and verification is essential, along with minimum of 3 years circuit design experience in industry. CMOS, VLSI, digital circuit design is a prerequisite.

Physical Design
Responsible for CMOS VLSI chip physical design of RISC microprocessor in advanced CMOS technology. Includes using state-of-the-art CAD tools to perform chip layout, wiring and chip timing analysis. A BSEE or higher, with emphasis on Computer Engineering or Circuit Design, is essential, along with at least 3 years of physical design experience in industry. RISC and CMOS, VLSI design experience (chip layout/wiring) necessary. Background in microprocessor design desirable.

Located between Lake Champlain and Vermont's Green Mountains, Burlington offers year round recreation and open space. Unspoiled beauty, affordable housing and a sense of community come together here. This is life at its most enjoyable; technology at its best.

IBM offers salaries commensurate with qualifications and a comprehensive benefit package. For confidential consideration, please send your resume, indicating area of interest, to: **IBM Corporation, Professional Recruiting, 1000 River Street, Essex Junction, VT 05452.**

An equal opportunity employer.