Language Constructs for Programming by Example

Robert V. Rubin

Department of Computer Science
Brown University
Providence, Rhode Island 02912

Abstract

Systems for programming by example permit the specification of algorithms through the use of *demonstrations* that manipulate *examples.* This paper analyzes systems for programming by example from a language point of view. Examples are analyzed as data abstractions, and demonstrations as abstractions for evaluation and control. Criteria are introduced for evaluating both the computational power and the expressiveness of the abstractions. The analysis demonstrates the existence of several previously unconsidered approaches to the more difficult problems associated with programming by example.

1. Introduction — Programming by Example

Programming by example is a mechanism whereby programs, or algorithms, can be specified via examples of their application. Consider, for example, the specification of a push operation for manipulating a stack. An element, serving as an *example* of an input argument, is composed with a second argument, serving as a canonical example of a stack, to form a new example of a stack. The composition of the two example elements corresponds to a *demonstration.* The operational components of this illustration are two: the example and the demonstration. This paper examines these operational components from a programming language point of view. Examples are treated as data abstractions, and demonstrations as abstractions for evaluationa and control.

A system for programming by example has several properties distinguishing it from a programming language environment. An example is a data abstraction with an equivalent visualization. A demonstration is a pattern of control, or evaluation, defined by *direct manipulation.*[1] Languages for programming by example describe examples and demonstrations. Conventional programming languages need consider neither the visualization of an abstraction, nor the direct manipulation associated with the demonstration. While several declarative and applicative languages have certainly been designed with a subset of similar considerations in mind, most languages are not explicitly concerned with "how" an abstraction is to be specified. Systems for programming by example embed representation and manipulation in the language constructs underlying examples and demonstrations.

Casting the operational components of programming by example in terms of programming language constructs permits the use of well established criteria in evaluating and contrasting such systems. Two such criteria are particularly relevant: *expressiveness*, and *expressive power.* Expressiveness is a measure of generality, flexibility, and applicability. Expressive power is a hard computational metric. The visual depiction of an example may be attributable to its expressiveness, and reflect little on its expressive power. On the other hand, a construct of deep expressive power may be of little use because of its poor expressiveness. This paper generates specific criteria for expressiveness and expressive power, and applies them to the language constructs used to specify examples, and demonstrations.

Algorithms and programs have classically been described via examples of their application. Babylonian mathematicians working between 1800-1600 B.C. developed formalisms notationally analogous to programming by example to specify algorithms for the solution of linear equations.[2] They had no algebraic notation equivalent to today's; instead, each algorithm was represented as a set of rules organized systematically around a set of numerical examples or diagrams containing reference points and numerical data (in sexagesimal notation). (See Figure 1).

This work was supported by a fellowship from the Graduate Education Engineering Program of Digital Equipment Corporation.

Although there are Babylonian algorithms without accompanying numbers serving as examples, apparently they are rare. Babylonian mathematicians also have conventions for specifying demonstrations that correspond to conditionals and iteration. Conditionals were not introduced explicitly in a program; instead, separate procedures with a distinctly different data set were provided. The correct procedure was identified from the example, and then the rules were systematically evaluated. Iteration was introduced in what notationally constitutes a macro expansion. If a computation was to be performed repeatedly, every computation using the example was specified explicitly, until the desired result was obtained. These methods differ little from some of the mechanisms used today in systems for programming by example.

Al-Khowarizmi, an Arabic mathematician from the 9th century A.D., (and from whose name is derived the term "algorithm"), authored a treatise on Algebra[3] where he prescribes computational abstractions fundamentally operational in nature. His definitional mechanism for an abstraction relies on specific examples. Al-Khowarizmi specifies a procedure encapsulating a set of computations via precise textual descriptions and labeled geometric figures. Notational abstraction in an algebraic sense did not exist in 9th-century Babylonian mathematics. Al-Khowarizmi describes the solution of quadratic equations by systematically referring his algorithm to six geometric diagrams. For mathematicians of the day, the operational abstraction was intimately connected with an example of its visualization and demonstrations of its application.

Modern programming languages and environments provide capabilities stylistically similar to the notations developed by the Babylonian mathematicians. Systems for programming by example are extraordinarily consonant with the specification techniques of Al-Khowarizmi and his Babylonian predecessors. Some of the same flavor of programming-by-example is retained in work in inductive inference, where a procedure is synthesized from examples of a program's behavior. This paper does not address program synthesis via induction.[4] Nevertheless, it is interesting to note that some of the early work in program synthesis contained definitional mechanisms complete within specific complexity classes; Essentially, the systems provided all the components of a programming language.

The systems considered here are enumerated in Figure 2. This paper provides a methodological approach to language design for programming-by-example. Specific features of each system are explored; no attempt is made to explore all features of these systems, nor is this particular list of systems assumed to be comprehensive.

2. Examples and Data Abstraction

The question "What is an example?" is categorically answered here as a form of data abstraction. The examples may represent data, records, variables, a sequence of variables, or perhaps an expression in the relational algebra. The semantics of the abstraction is dependent on the language in which the example is formulated.

The criteria for analyzing the expressiveness of the examples in a system are primarily visual:

[1] Can the semantics of an example be completely specified diagrammatically?

[2] Are multiple views of the abstraction possible? Is there a function mapping each visualization to a specific component of the semantics?

We define a ***structure diagram*** as a two-dimensional figure that contains a potentially isomorphic mapping to the clauses used to represent the object. The counterpart to structure diagrams, *logic diagrams*, are defined in[5] as two-dimensional geometric figures with spatial relations that are isomorphic with the structure of a logical statement. A simple form of logic diagram is one that expresses a set of transitive asymmetric relations. The evolutionary scale, for example, provides a graphical depiction of "is derived from" relations. The logic diagram is well established as a useful tool that neither supplants nor detracts from other notational abstractions.

Criteria for evaluating expressive power are classically applied to the evaluat on of programming languages:

[1] What is the relevant notion of aggregation? Are there corresponding notions of ordered, or unordered aggregates? Can all classes of n-ary trees be represented? Finite, or infinite?

[2] What facilities are available for expressing the relationships between different examples?

[3] Does the language permit the expression of type information? When and where is typechecking applied? Can polymorphic operators be defined?

How powerful a notion of aggregation is desirable? S-expressions in Lisp are sufficient and mathematically elegant (defined recursively or inductively) and can be used to represent all classes of trees.

What language facilities can be used to characterize relationships between examples? Are the expressions/predicates generated in the system first- or second-order? First-order languages are not as expressive as second-order languages. For example, transitive closure cannot be expressed in the first-order predicate calculus or the relational algebra.[6] However, first-order expressions can be evaluated more efficiently, and, as described below, contain visual properties second-order expressions do not.

Is type information important as a characterization of expressiveness, or expressive power? Lisp originally contained only two types: atoms and dotted pairs. Pascal, considered by some as more expressive, has facilities for user defined types. Yet its expressive power is identical to that of Lisp.

Questions may be raised about the formulation of these particular criteria. Languages have offered different solutions to these problems, cast often within different models of computation. The justification offered here is that these provide a basic platform for comparison.

2.1. Clauses in QBE, ThinkPad, and PHD

The examples manipulated in Query-by-example (QBE),[7] ThinkPad,[8] and PHD[9] represent *clauses* or *terms* in the relational algebra, Horn-clause logic, and a second-order logic language by the name of Omega, respectively. Examples in all of these systems are manipulated in the context of structure diagrams, which depict the set of clauses associated with a particular algorithm.

The design of QBE is based on the observation that the relational algebra can visually be represented by a table. The table in QBE is a structure diagram. A translation function maps the data associated with the structure diagram to expressions in the relational algebra. An example in QBE is a variable, or a set thereof, defined by its membership property in the structure diagram, and its relation to other examples. Minimally, an element in each row in a table corresponds to a tuple. All examples represent terms within an expression in the relational algebra. If the identical variable is used in more than one structure diagram, it provides a pattern that must be properly formulated in the translation to an expression in the relational algebra.

The expressiveness, and consequently the elegance of QBE lies in its use of a table as a structure diagram. The table provides the user with a model that is highly intuitive. The power of QBE lies in its expressiveness.

QBE's expressive power includes the ability to specify arbitrary n-ary trees, although its underlying language also includes integrity constraints constraining a tree from containing cycles. The integrity constraints ensure three properties of an acyclic tree: that examples within a tree are irreflexive (e.g. an employee cannot be his own manager), and, asymmetric on the transitive closure of an example (e.g. an employee cannot be both somebodys subordinate and his manager), and unique (e.g. an employee cannot have more than one manager). Examples representing any arbitrary position in the tree may be specified.

An example in ThinkPad is a structure diagram that embodies a set of clauses constraining the particular example. The expressiveness of ThinkPad is derived from a clausal representation of structure diagrams translated into Prolog clause (Horn-clauses).[10] ThinkPad, unlike QBE, provides some of the graphical tools for the design of an arbitrary representation (i.e. structure diagram) for a set of clauses. It is from the ability to design an arbitrary representation that ThinkPad gains its expressiveness. For example, a structure diagram may represent a data structure containing typed elements or substructures. Each type has a graphical depiction, each subcomponent a specific spatial location. Substructures of the same type as the structure diagram are defined recursively. Recursive structure diagrams contain sufficient expressive power to generate all classes of trees.

Structure diagrams in ThinkPad are not necessarily isomorphic with their clausal representation. For example, a table may be expressed in n-ary or binary normal form. To solve this problem, a normal form is automatically selected before the diagram is translated into a set of clauses.[11] Once a normal form is selected, the mappings become isomorphic. The algorithms for translation between structure diagram and Horn-clauses are described in[12]

Type information and the membership properties are modeled by specific asymmetric transitive Horn-clauses. Containment within a structure corresponds to an *is-member* clause. Identification as a type is modeled as a *is-a* clause. An n-ary normal form is used for mapping containment predicates. Type information in ThinkPad is problematic. Variables in Prolog, and consequently, their operators, are generally polymorphic. For example, the "<" operator may be applied to strings as well as integers. In ThinkPad, the is-a relationship precludes polymorphism.

Structure diagrams in ThinkPad may have clauses associated with them that are boolean, arithmetic, or relational (arithmetic and algebraic) expressions. The subcomponents of the structure diagram may represent bound or unbound variables, constants, terms, or composite functions. Any arbitrary location in a tree or structure diagram can be used as an example. The translation algorithm is capable of generating the set of clauses describing the path automatically.

PIP[13] preceded ThinkPad with an environment (albeit not the underlying language) for manipulating examples. In PIP the editor provides the typing mechanism, whereas the underlying FP interpreter does not. Structure diagrams in PIP have type and containment relations that are maintained solely by the editor.

It is unclear that the expressive power of both QBE and ThinkPad are equivalent. The uniqueness constraint in QBE is relaxed in Prolog, where more than one binding may be possible. On the other hand, queries in Prolog are not necessarily satisfiable. The examples in ThinkPad can be used to model queries in QBE. This is not surprising given that Prolog has also been used as an extended implementation language for QBE.

ThinkPad is designed to permit the user the graphical flexibility of depicting arbitrary data structures, and this includes, for example, tables, trees, lists. QBE is limited in its expressiveness to tables. The expressiveness of these systems may or may not be equivalent, depending on the particular conceptual models associated with an application.

PHD is a system that also relies on a translation between structure diagrams and a clausal form to represent examples. The structure diagrams in PHD are called *forms*. The language for forms is based on a second-order logic language called Omega.[14] PHD was designed to provide more expressive power to a system like QBE. It included a more generalized control structure, called a homomorphism (described below), and its logic language is second-order, permitting the expression of more complex relationships between data than the relational algebra. The structure diagrams in Omega permit the definition of all first-order structures, i.e. all classes of trees. Recursive data types in Omega are considered as Σ-algebras. Lists are inductively defined as:

$$List = Null + Cons(Car{:}Numb;Cdr{:}List)$$

This algebraic formulation states that objects of type *List* are either of type *Null* or of type *Cons*. The data type *Cons* is a record containing a *Car* (which is a *Numb*) and a *Cdr* (which is a *List*). *Null* and *Numb* are referred to as the *generators* of the Σ-algebra, and *Cons* is its only *operation*. This corresponds to the notion of a list embodied as an s-expression in Lisp.

Omega also permits the specification of existentially and universally quantified relationships between examples. Translating a first-order expression into a structure diagram is a problem reducible to selecting a normal form; this is not the case with second-order descriptions. Second-order descriptions are capable of describing the transitive closure of a relation. The structure diagram used in PHD is not inherently capable of describing such a relationship. Consequently, forms in PHD rely on a highly textual notation inside of structure diagrams.

Remarkably, it appears that although PHD provides greater expressive power, it lacks the expressiveness of QBE because it is heavily reliant on textual notations within a form to depict its relationships. Under the operative assumption that diagrammatic properties are important to expressiveness, PHD is problematic. PHD provides no mechanisms with which to represent second-order expressions in a structure diagram. The structure diagrams in Omega permit the definition of all first-order structures, i.e. all classes of trees. There is no function in PHD for mapping second order clauses into a structure diagram. Providing such a mapping appears to be an open research question.

Examples in these systems are facilely represented as variables within a collection of terms or clauses. Tables and structure diagrams contain mappings between clauses and visualizations, although representationally, second order clauses are problematic. Type information provides for diversity in visualizations; On the other hand a simple clausal approach for type characterization precludes operator overloading. The integrity constraints used to maintain acyclic graphs are quite possibly more restrictive than necessary. Evaluation of the terms within another model of computation may relax these restrictions.

2.2. Instances as Examples in SketchPad, ThingLab and SmallStar

SketchPad[15] and ThingLab[16] are the seminal pieces of work in the domain of graphics-based programming. The programming language abstraction for an example, shared in common with SmallStar — a programming by example system designed for the same domain as QBE — is an object-oriented one: An example in these systems is an *instance* of a *class*. The SketchPad prototype was completed before the introduction of Simula-67; ThingLab and SmallStar are written in Smalltalk. ThingLab heavily influenced later revisions of the Smalltalk language. The class/instance metaphor for an example is a powerful but general radicalization of records as expressed in Algol-derivative languages. We explore the properties of a class/instance of each system.

Objects in Sketchpad are *picture definitions* used to model a hierarchy of graphical objects. Picture definitions could be combined with other picture definitions to compose new picture definitions. Several *primitive types* were also available for defining the fundamental elements in a structure. These types were purely graphical: *points, lines,* and *circle arcs.* Two kinds of relationships could be established in Sketchpad among picture definitions: *master/instance*, or *copy.* A master/instance relationship maintained the same picture definition. An *instance* had the same shape, but could be translated, rotated, and scaled. The private parts of a picture description were designated *attachers* — each instance contained its own values for a particular attacher. If a picture definition for a master was modified, all instances of the master were also modified. A copy of a master no longer held a relationship to the master. A copy was simply a new master whose hierarchical description is identical only initially to the originals. Revisions to the hierarchy of the copy are not reflected in the original. Hierarchical descriptions could be embedded within other hierarchical descriptions. This provides sufficient to model arbitrary tree structures.

Within the context of programming by example, picture definitions are structure diagrams with geometrical constraints. A master generalizes or encapsulates a specific set constraints germane to an example; instances are examples of masters with specific values for the attachers.

ThingLab extended the expressive power of these language features in several significant ways. Objects contained in ThingLab are not constrained to geometric forms — ThingLab uses Smalltalk's[17] more general class-instance structure where instances contain internal variables that can hold any instance state. ThingLab also introduces constraints on objects. A constraint specifies a relation that must be maintained. Constraints provide a mechanism for describing relationships in a *part-whole* hierarchy. Constraints are also integrated in the description of inheritance hierarchies. (Multiple inheritance was introduced in ThingLab before it was introduced in Smalltalk.) Multiple inheritance provides the expressive power to model asymmetric transitive relations between classes.

In SmallStar, examples are referred to as *data descriptions.* SmallStar data descriptions are instances of a class. Every example in SmallStar is a member of a predefined class. All objects in a class have the same properties. All instances of data descriptions in a class contain the same property list. The values for properties of a particular instance may differ from those of another instance. Unfortunately, SmallStar data descriptions are closed entities. It is not possible to dynamically add or delete properties of a class. The property sheets are not computationally complete. For example, the class *Folder* contains a method for choosing the *first* or *last Document*, but not the *nth Document*. No facility is included for adding *nth* to the property sheet. Though the *nth* document may be computed using iteration, this is a liability from an expressiveness point of view.

The class *Container* is the superclass in SmallStar where aggregation of objects is possible. The class *Container* in SmallStar appears to be inherited only by *Folder* and *Filedrawer*. *Container* appears not to permit the aggregation of *Containers*; as a consequence it cannot be used to model all classes of trees. For example, can *Folders* be inserted in *Folders*? Can they be inserted in *FileDrawers*? Can *Filedrawers* be inserted in *Filedrawers*? In no example in SmallStar is this capability found. A major increase in expressive power could be gained with the inclusion of such a minimal data structure.

The emphasis in the design of SmallStar was placed clearly on expressiveness. Experimental studies accompanied expressiveness considerations in the design of the user-interface. The data-descriptions in SmallStar are depicted using structure diagrams that supports various levels of abstraction. Data descriptions are depicted using the property sheet metaphor established for the Star workstation. The property sheets for classes are organized hierarchically; information is encapsulated within a property sheets according to the hierarchy established for the class. It appears that given more consideration to expressive power, SmallStar data descriptions can embody within a completely different framework computational expressiveness and power equivalent to QBE.

The importance of master-instance and class-instance language mechanisms to programming-by-example cannot be understated. Semantically the class-instance relationship is well prescribed. An instance corresponds closely to our intuition of an example. Hierarchy, inheritance, multiple-inheritance, and constraints permit the definition of both symmetric and asymmetric transitive relationships. Multiple-inheritance, in particular, lends itself to reuse of object classes. Distinguishing between classes in ThingLab corresponds to creating an instance (copy) of a class and specifying a new set of constraints. Creating a new class definition involves the addition of new asymmetric transitive relationships. New classes can be programmed by example by copying relationships and constraints belonging to instances of other classes.

3. Demonstrations and Control of Evaluation

In a programming by example system, a user manipulates examples to demonstrate a desired pattern of evaluation. The demonstrations correspond to the language constructs for evaluation and control in the underlying system. This section analyzes the expressive power and the expressiveness of these language constructs. Figure 3 provides a table of these control structures.

What language constructs are embedded within a demonstration? In some systems, a demonstration is the establishment of an additional relationship between examples. Often, this corresponds to the introduction of an additional clause or a conditional. In that case, the demonstration differs little from the characterization of an example, as discussed above. Yet another form of demonstration incorporates iteration. Demonstrations are also a form of procedure invocation. For example, grouping two objects may correspond to the invocation of an append operation. The different models of procedure invocation provide equivalent computational power. The expressiveness associated with the underlying model of computation gives a significantly different feel to the user interface.

This section examines the expressiveness and the expressive power of programming by example constructs for iteration, procedure invocation, and conditionals. Expressive power is a measure of the computations possible in a given language. The use of complexity metrics in analyzing practical programming environments provides insights on the fundamental nature of the computation. An oft cited example is that the clean theoretical semantics of the relational model underlying QBE are insufficient for expressing the transitive closure of a relation. This computational difficulty is overcome by the introduction of practical language constructs. A language containing bounded loops, generic variables, and equality[18] is sufficient for computing the *primitive recursive functions.* (Virtually all functions of practical value are primitive recursive.) On the other hand, *decision* problems associated with a language may still remain. Where relevant, we discuss these considerations.

Whereas expressive power is a hard computational metric, criteria for the expressiveness language structures for evaluation and control remains a metric with properties that are difficult to define. Expressiveness, is intended as a measure of both general applicability — e.g. modularity — and representativeness. As a metric for representational properties, the criteria for evaluating examples established in section 2 are applicable. Iteration in a flow chart, and recursion in a dataflow diagram may in some ways be viewed as of equivalent expressiveness. Both contain functions providing mappings to particular implementations of a structure diagram. On the other hand, a construct incorporating iteration and a conditional is equivalent in expressive power to a construct for recursion, and yet by the representation metric, may also be said to be equivalent in expressiveness. The definitional properties associated with constructs for recursion fulfill the language requirements for encapsulation, modularity, and scope; Iteration constructs often do not contain these features. In the following discussion, we distinguish between demonstrations embedding procedure invocation, and demonstrations embedding iteration and conditionals.

3.1. Demonstrations Embedding Operators, Conditionals, and Iteration

In PHD, a user selects a form representing an example, and then creates a *homomorphism* that operates over the example. The homomorphism primitive embeds an operator, conditionals, and iteration in a single control structure. (The homomorphism primitive has its origins in theoretical work done by Burstall and Landin.)[19] In the PHD system, an *operator* is applied to a list using the *map* primitive. The algebraic properties of the operator, as well as those of the list (including conditional properties associated with the list) are embedded within the homomorphism. Map is an iteration construct that applies an operator to a list of arbitrary length. The semantics of a homomorphism (in Lisp notation) as specified by Aeillo[20] is provided in Figure 4. The computational power of a homomorphism can express at least all of the primitive recursive functions.

The homomorphism is particularly expressive when used with a general-purpose operator (e.g. summation, see Figure 5). Two algebraic properties are requisite in the design of operators incorporated within map: operators must contain an identity function, and operators must accept as input a list of any arbitrary size. The identity property permits the determination of the type of the result where fields have not been completely specified. The requirement that an operator accept an arbitrary number of elements implies that functions composing the example and the operator must specifically handle structures with no elements, structures with one element, and structures with many elements. (An equivalent algebraic viewpoint has been implemented by Goguen et al.[21] in the language OBJ and its variants.)

The algebraic regime for operators seems to increase the expressiveness of the language. Certainly the algebraic definition provides a precise characterization of the corresponding language notions of applicability and generalizability. It is in this light, however, that we note the absence of a facility for defining the algebraic properties of new operators. It appears that only operators defined within the system meet the algebraic criteria and new operators do not. The design of a facility for the demonstration of the algebraic properties of operators is an open research problem. The homomorphism primitive is closed under composition. From an expressiveness point of view, a composition facility adds a mechanism for encapsulation. This increases the expressiveness of a language by encouraging modularity and reusability.

Apply is a control structure formulated for use with ThinkPad. Like a homomorphism, the control structure derives its expressiveness by providing a facility for embedding examples (in clausal form), conditionals, and, iteration.[22] The apply control structure is derived from the "universal"[6] control structure

for tuple *t* **in** relation *R* **do** <*statement*>

where <*statement*> may modify relation *R*. Intuitively, it is designed to traverse a well-defined route over a two-dimensional data structure while operations are executed at each point along the route. The route itself may be modified by the result of an operation.

The apply function contains three components: a *Route*, an *Operation*, and *Modify.* Evaluation of the clauses representing the Route yields the universal closure of the clause (whose binding may be deferred); this Route corresponds to R above. An Operation object iterates over each element in the route. The Operation component maps a predicate to each tuple in the Route. The modify component specifies the order in which clauses are to be considered in the route.

Apply is of expressive power capable of computing polynomial, or primitive recursive functions. The expressiveness of this control structure is that all the components are explicitly available for manipulation. A route is specified via an example. The same example used for the route is subjected to the demonstrations associated with an operation — this may include specifying additional relationships, or incorporating the clause within a predicate. Unlike a recursive definition of breadth-first search where the stack is an implicit structure the apply primitive forces the user to select the representation of the data structure for use as the route object. For example, the route may be a tree, a list, a stack, or an ordered or unordered set. When the route is explicitly defined as a stack, manipulations are permitted only via "pop" and "push" operations.

SmallStar includes a *set iteration* construct. To add the primitive for set iteration an example is selected, and demonstrations applying operators to the example are recorded. (The transcription mechanism is described below.) After the demonstrations are recorded, the statement *Repeat with everything matching* is inserted into the prerecorded sequence of statements. The data description describing the "*match*" is simply the class that can be extracted from the example used in the demonstrations. Unfortunately, it appears that the only classes for which iteration can be defined consist of examples belonging to the class *container* or the class *Table.*

In defining the SmallStar control structure, the emphasis was once again on the expressiveness of the structure. The SmallStar strategy for introducing iteration was the result of several prototyping and testing efforts. In the initial prototype a user began an iteration with a *for each do* statement and then proceeded to define the manipulations. It was felt that inserting iteration after the procedure was defined was more useful. The computational power is equivalent to a language containing bounded loops and equality.

Homomorphism, apply, and to some extent, set iteration, are constructs designed from equivalent points of view. They are similar in many ways to the "generator" constructs in modern languages. The basis for their construction lies in their encapsulation of an aggregate to operate on individual members of the set in a constructive order. This in direct constrast to the approach taken in QBE. QBE contains special purpose operators incorporating iteration and few langauge constructs for encapsaulating and parameterizing queries. Examples of such operators include average, sum, maximum. The homomorphism, and apply constructs explicitly attempt to define a generalized control construct.

3.2. Procedure Encapsulation and its Mechanics

Procedures are a language mechanism used primarily to encapsulate a sequence of statements and as a vehicle for introducing recursion. Adding recursion to a language containing either apply, or a homomorphism, or for that matter, a construct embedding conditionals and iteration, provides an alternate approach to specifying functions of equivalent computational power. In such cases it is a syntactic device used to encourage modularity.

In this section we discuss the mechanics of how a procedure is encapsulated because of the semantic nature of the editing commands used for encapsulation. Some of editing commands are equivalent to procedure invocation, and are as such, explicit demonstrations. Other editing commands are simply a device for encapsulating a procedure, and are not particularly relevant.

Procedure encapsulation appears to be an integral part of the (misguided) folklore surrounding programming-by-example. The actual mechanics of how examples are used to define a program are in some ways irrelevant to the power of the program, but can be interesting in their own right. They provide a perspective on the synergy between a language and the editing technology used to prepare a program in that language.

Pygmalion[23] pioneered the use of *program transcription* as a device for encapsulating procedures. To encapsulate a set of operations, a user begins a command such as *Start Recording* and proceeds to manipulate objects. Each manipulation is recorded in the transcript. Manipulations (i.e. demonstrations) include assignment, or calculation of arithmetic or boolean expressions. The set of manipulations that appear in the transcript is defined by the language constructs or icons (procedures) available in the system. To close the encapsulation, a user gives the command *Stop Recording.* The commands entered in a transcript can now be reinvoked, or they can be modified. In Pygmalion, the transcript is stored within a particular icon for playback.

Demonstrations in SmallStar and Pygmalion correspond to either establishing new boolean or arithmetic relationships between examples, or procedure invocation. SmallStar contains a message-passing model of procedure invocation which is based on the object-oriented programming paradigm of

Smalltalk. Objects on the screen belong to a class named *Icon*. All members of the class Icon can evaluate a set of messages which are used for invoking methods that operate on the object. User manipulations are translated into specific messages. For example, the *point* mouseclick is translated into a *SELECT* message which is sent to the icon. Selecting the *Move* button sends the *MOVE* message to the icon. This results in in replacing the cursor with the icon. A subsequent *point* mouseclick translates the command into the message *INSERTAT:pt* that is sent to the *Desktop* icon.

SmallStar provides translation functions to increase "readability" of the transcript. A transcription was more than a transcript of the raw operations. A user was exposed to a transcript that had been parsed and translated into a more readable form. While this had no effect on the expressive power of the system, it appears to be a useful technique for increasing the expressiveness in the representation of a procedure. Transcription in Pygmalion and SmallStar results in a sequential block of operations.

The semantics of message passing coupled with the translation technique described above forms the basis for much of the well-emulated visual programming paradigms in place today. It is executed rather elegantly in the SmallStar user interface.

The Pygmalion model for program transcription appears to provide computational power equivalent to the power of a finite state automaton. Embedded at the nodes are sequences of operations. Arcs contain decision functions embedded within *if then else* expressions. Pygmalion transcripts can accept parameters, can be named, and can be invoked. It is certainly possible to define a recursive program transcription. This appears not to be possible in SmallStar even though virtually identical transcription mechanisms are used. Much along the same lines as Pygmalion, an experimental programming environment, Tinker,[24] has been developed. Tinker examines the finite state transcription, and queries the user for transcriptions where additional transition functions may be included.

New messages cannot be synthesized in the SmallStar system. The SmallStar virtual machine is defined in terms of the predefined set of messages and is a closed system. Although SmallStar does have a procedure encapsulation mechanism, no mechanism for parameter passing is provided. Procedures access data by the access conventions established for data descriptions and icons.

PIP[13] uses a modified form of program transcription to define procedures. The language underlying PIP is functional, and is based on a variant of Backus's *FP*.[25] The syntactic and semantic restrictions of *FP*, i.e. no temporaries, carry through to the program transcription process. A function is initially defined as having several input and output types. Input types are composed with output types to form functional expressions.

In PIP the notion of a transcript is inherently connected to that of a stack. A user initiates a transcript by issuing the *Connect* command. To compose a set of objects, a user selects the set of input objects. Each object selected is then placed on a stack transcription, available for manipulations. The user may connect a set of inputs by selecting an object that represents an operator. Selecting an operator pushes the operator on the stack as well. Issuing a *Done* command pops the stack and completes the transcription.

Although the choice of transcription mechanism in PIP was dictated by the semantics of the underlying language, it is an interesting program construction technique in its own right. It appears to provide the power of a pushdown-automaton in constructing a program. Finite-state-automata are of less expressive power than pushdown-automata. Stack-based transcription is more powerful than a finite-state transcription. While this discussion concerns the preparation of demonstrations for a program and not the expressive power of the program generated, it is possible that with greater computational power in the specification process, more examples may systematically be addressed. This is an area ripe for future examination.

PIP permits the same function to be defined several times, each with a different specification of the input data-structure requirements. This facilitates the definition of functions whose actions are determined by the structure of the input. Each instance of a multiply defined procedure is referred to as a *function variant*. This may, for example, be particularly useful in defining tree manipulation algorithms that handle leaf nodes and internal nodes differently. Permitting a function to be multiply defined makes it easy to generate a procedure whose components are designed to use examples that are semantically unequivalent.

The semantics of a function defined via several *function variants* are derived using a *maximal match* procedure. The specific function variant to be invoked is chosen from those attributes of the passed data structure that fit best with the various input descriptions.

Unlike Pygmalion, SmallStar, and PIP, the encapsulation and demonstration of a procedure in ThinkPad is not based on a transcription mechanism. Procedures in ThinkPad are composed of four collections: a Function collection, an Input collection, a Result collection, and a Demonstration collection. The Function collection represents a type specification of the inputs and the results. The Input collection contains examples conforming to the type specification. Parameter specifications in ThinkPad are strongly typed, and may be part of a class-subclass hierarchy of relations. Most relevant, the Demonstration collection, contains groups of objects with explicit relations between the groups and between the objects in each group. These relations are established through use of demonstrations. Demonstrations in ThinkPad correspond to semantic actions of the commands in an editor. For example, the editing function ***group*** corresponds to either the lisp dot operator (cons), or the addition of a variable or a clause to a predicate.

ThinkPad provides a translation algorithm that accepts these collections and generates Prolog code. Theoretically, it should be possible to specify operations nondeterministically. However, the satisfiability of a clause or procedure set in Prolog is often dependent on the ordering of its clauses. Consequently, ThinkPad collections retain a left-to-right, top-down ordering. Functions are defined in ThinkPad using a mechanism that resembles syntactically the function variants used by PIP, but whose semantics radically differ. Functions may be multiply defined, with different constraints on the types of data accepted for each definition. The underlying semantics are the Prolog semantics of Horn-clause interpretation using resolution.[26] The semantics provides a mathematically clean approach to defining functions based on the individual examples. If data do not meet the requirements of the description, the predicate invocation will fail, and another function variant selected.

Functions in PHD are forms (structure diagrams) containing a title, a set of input parameters and a field for a result. Forms increase the expressiveness of the language by providing a general purpose mechanism that encapsulates and modularizes homomorphisms. The entries may contain predicates specifying relationships between pieces of data and type information. Forms are in this way, equivalent to function invariants. The result field is derived from a set of computations. Its description as a parameter consists only of a name. The invocation of a procedure corresponds to the visual display of the form displaying the function name, its parameters (and attributes), and its result. Procedures have the additional property of embodying a homomorphism. Homomorphisms are closed under composition. Consequently, procedure invocation is equivalent to the explicit inclusion of a homomorphism specification within another homomorphism.

We have shown that many of the demonstrations performed in these systems may not be readily recognized as a form of procedure invocation. This is attributable to the underlying model used for invoking procedures. For example, the SmallStar and Pygmalion model rely on the message-passing model of procedure invocation. A click on an icon is translated into a specific message that is then passed to the icon. PIP and ThinkPad permit procedure invocations as well, although the semantics of a procedure within each system differs significantly.

Whereas the different models of procedure invocation provide equivalent computational power, the properties associated with the expressiveness of these constructs differs qualitatively. Transcriptions provide a modularized pattern of control. "Extended" transcripts are parameterized and reusable. Function invariants tend to reduce the size of the demonstration sequence by constraining the scope of applicable examples. If a particular data set is not relevant, a new specification based on a different set of examples is in order. Interestingly enough, the transcription tools are of different expressive power — however, the environment is of no impact on the expressive power on the generated programs, and only help to systematically organize the generation of examples.

3.3. Conditionals

The semantics of an example in many of the systems for programming by example significantly reduces the need for explicit conditionals. Data descriptions in SmallStar serve to describe the attributes of a particular example. If the attributes are inadequate, they may either be modified, or an *if then else* conditional may be post-facto inserted in the transcripted procedure. In systems relying on clausal representations of data, the clauses themselves and the function invariants make explicit *if then else* conditionals unnecessary. Function invariants are a form of implicit *if then else*, as are predicates within a function invariant.

4. Conclusions

For the most part, programming by example has been viewed as a user-interface problem. D.C. Smith, the designer of Pygmalion and the primary architect behind the Star user interface, has written about user interface design:[27]

> User-interface design is still an art, not a science. Many times during the Star design we were amazed at the depth and subtlety of user-interface issues, even such supposedly straightforward issues as consistency and simplicity. Often there is no one "right" answer. Much of the time there is no scientific evidence to support one alternative over another, just intuition. Almost always there will be tradeoffs...

Programming by example, like many concepts in mathematics, is to a large extent a methodology born of intuition. Al-Khowarizmi and his fellow Babylonians were hampered not as much by the unavailability of a computing device as by lack of a conceptual and notational framework with which to express a "variable". At the heart of programming-by-example lies a problem in a similar vein: precisely what constitutes an example, and its demonstrations? This question has been approached not as a problem of "user-interface design" but as an applied exercise in computability.

By what principles can better systems be built? The strategy suggested here is to treat the system as a programming language. When programming by example fails to scale up, the cause may often be found in an examination of the expressive power vis a vis the expressiveness. The criteria of computational power and computational expressiveness are useful metrics in examining the relationship between what are seemingly unrelated systems. Ultimately, it is both the expressive power and expressiveness of a language that defines its usefulness. Good language design, as underlies programming-by-example and good systems design, is part science, and part art.

Acknowledgements

Marc Brown, Eric Golin, Joseph Pato, Steven Reiss, and Peter Wegner provided many insightful comments. The anonymous critique of the reviewers on the selection committee were particularly helpful in reorganizing this paper, and are gratefully appreciated.

References

1. B. Shneiderman, *Software Psychology: Human factors In Computer and Information Systems*, Winthrop Publishers, Cambridge, Mass (1980).
2. D. Knuth, "Ancient Babylonian Algorithms," *Communications of the ACM* **15**(7) pp. 671-677 (July 1972).
3. Al-Khowarizmi, *Algebra*813-846 A.D..
4. A. Biermann and R. Krishnaswamy, "Constructing Programs from Example Computations," *IEEE Trans. on Software Engineering* **2**(3) pp. 141-153 (1976).
5. M. Gardner, *Logic Machines and Diagrams, 2nd edition*, University of Chicago Press (1982).
6. A. Aho and J. Ullman, "Universality of Data Retrieval Languages," *Proceedings, 6th ACM Symp. on Principles of Programming Languages*, pp. 110-117 (January 1979).
7. M. Zloof, "Query-by-Example: Operations on the Transitive Closure," *IBM Research Report RC-5526*, (1976).
8. Robert V. Rubin, Eric J. Golin, and Steven P. Reiss, "ThinkPad: A graphical system for programming-by-demonstration," *IEEE Software* **2 4N 2** pp. 73-78 (March 1985).
9. G. Attardi and M. Simi, "Extending the Power of Programming by Examples," in *Integrated Interactive Computing Systems*, ed. E. Sandewall,North-Holland (1982).
10. W. F. Clocksin and C.S. Mellish, *Programming in Prolog*, Springer Verlag (1984).
11. C. Beeri, P. Bernstein, and N. Goodman, "A Sophisticates Introduction to Database Normalization Theory," *Proc Int Conf on Very Large Data Bases*, (September 1978).
12. R. Rubin, E. Golin, and S. Reiss, "Compiler Aspects of ThinkPad," in *Workshop on Interactive Envrionment*, Lecture Notes in Computer Science, Springer Verlag (1986).
13. G. Raeder, "Programming in Pictures," PhD Thesis, University of California ().
14. G. Attardi and M. Simi, "Consistency and Completeness of Omega," *Seventh International Conference on Artificial Intelligence*, (1981).
15. I. Sutherland, "Sketchpad, A Man-Machine Graphical Communication System," PhD Thesis, MIT (January 1963).
16. A. Borning, *Thinglab -- A constraint oriented simulation laboratory*, PhD Dissertation, Department of Computer Science, Stanford University (1979).
17. Adele Goldberg and Dave Robson, *Smalltalk-80: The language and its implementation*, Addison-Wesley (1983).
18. A. Chandra, "Programming Primitives for Database Languages," *Proceedings, 8th ACM Symp. on Principles of Programming Languages*, pp. 50-62 (January 1981).
19. R. Burstall and P. Landin, "Programs and their proofs: an algebraic approach," pp. 17-43 in *Machine Intelligence 4*, ed. D. Michie,Edinburgh Univ. Press (1969).
20. L. Aiello and G. Prini, "Technical Correspondence," *ACM Transactions on Programming Languages and Systems* **2**(2) pp. 263-64 (April 1980).
21. J. Goguen and J. Meseguer, "Equality, types, modules and generics for logic programming," *Proceedings of the Second International Logic Programming Conference*, (July 2-6 1984).
22. R. Rubin and J. Gonczarowski, "Extending a first-order language with a construct for two-dimensional programming," Submitted for publication (February 1986).
23. D. C. Smith, *Pygmalion: A computer program to model and stimulate creative thought*, PhD dissertation, Stanford University (1975).
24. H. Lieberman and C. Hewitt, "A Session with TINKER: Interleaving Program Testing with Program Design," *Record of the 1980 Lisp*

Conference, pp. 90-99 (1980).

25. J. Backus, "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs," *Communications of the ACM* **21**(8)(August 1978).
26. M. H. Van Emden and R. A. Kowalski, "The Semantics of Predicate Logic as a Programming Language," *Journal of the ACM* **23** pp. 733-742 4, (October 1976).
27. D. C. Smith, "Designing the STAR User Interface," in *Integrated Interactive Computing Systems,* ed. E. Sandewall,North-Holland (1982).

Figures

The sum of the length, width, and diagonal is 12 and 12 is the area.
What are the corresponding length, width and diagonal?
The quantities are unknown.
12 times 12 is 2,24.
12 times 2 is 24.
Take 24 from 2,24 and 2 remains.
2 times 30 is 1,
By what should 12 be multiplied to obtain 1?
12 times 5 is 1,
5 is the diagonal.
This is the procedure.

Figure 1: Babylonian algorithm for the solution of linear equations.
The procedure was originally analyzed by D. Knuth.

The procedure in Figure 1 presents a Babylonian algorithm to solve a linear equation based on the formula

$$d=\frac{1}{2}(l+w+d)^2-\frac{2A}{(l+w+d)},$$

where

$$A=lw$$

is the area of the rectangle and

$$d=\sqrt{(l^2+w^2)}$$

is the length of its diagonals.

Early Mathematics	Office Automation	General Purpose Programming	Graphical Programming
Linear Equations	QBE	Pygmalion	SketchPad
Al Khowarizmi-Algebra	PHD	PAD (?)	ThingLab
	SmallStar	PIP	Juno
		ThinkPad	

Figure 2: Systems that utilize programming-by-example.

Control Structure	Construct	PBE System
Iteration	*Homomorphism — Map*	PHD
	Apply	ThinkPad
	Set Iteration	SmallStar
Function Invocation	*Parameter Resolution*	ThinkPad
	Function Variants	PIP
	Message Passing	SmallStar
	Homomorphic Form Composition	PHD
Conditionals	*Data Description*	SmallStar
	Filter	PHD
	Predicate Discrimination	PIP
	Nondeterministic Evaluation	ThinkPad
Program Encapsulation	*Transcription (FSA)*	Pygmalion
	Transcription + Translation	SmallStar
	Collections	ThinkPad
	Stack-based Transcription	PIP

Figure 3: Control Structures used in Programming by Example Systems

```
(DE HOM
    (LAMBDA (FNULL FNUMB FCONS)
        (FUNCTION
            (LAMBDA (ARGS)
                (COND
                    ((NULL ARGS) (FNULL)
                    (T (FCONS (FNUMB (CAR ARGS))
                            ((HOM FNULL FNUMB FCONS)
                            (CDR ARGS)))))))))
```

Figure 4: Semantics of a homomorphism

```
(HOM
    (FUNCTION (LAMBDA () 0))
    (FUNCTION (LAMBDA (X) X))
    (FUNCTION PLUS))
```

Figure 5: Using the homomorphism primitive to define a summation operator.