Layered Architecture of Multiple Programming Language System for Multiparadigm Programming

Shigeo Sugimoto, Tetsuo Sakaguchi, Koichi Tabata
University of Library and Information Science
1-2, Kasuga, Tsukuba, Ibaraki, 305, JAPAN

abstract

This paper describes a layered architecture that can afford unified environment for multiparadigm programming on sequential processors. The architecture gives open-endedness to a system for programming in multiple languages based on multiple paradigms. Language specific constructs such as control structures and variable definitions are given in an upper layer. Lower layers describe concepts common to all languages, such as data types and reference protocols. Since communication among languages is handled in the lower layers, the upper layers provide an open-ended abstraction for multiparadigm programming.

1. Introduction

In 1980's, new programming paradigms such as logic paradigm and object-oriented programming gained wide acceptance. From the general perspective of programming, problems comprise heterogeneous collection of subproblems. This makes the idea of multiparadigm programming attractive, since each sub-problem can be modelled and programmed in an appropriate paradigm. Zave [10] presents multiparadigm programming solutions for a communication system. Various multiparadigm programming languages have been proposed[5]. LOOPS[5] provides four paradigms for knowledge representation; object-oriented, data-oriented, rule-oriented, and procedure-oriented. TAO[9] provides a unified environment for object-and logic-oriented paradigm based on LISP. Common LISP includes Common LISP Object System (CLOS) for object-oriented programming[7]. The authors had implemented a knowledge representation language for a multi-media knowledge-based system based on Prolog and Smalltalk-80[48]. However, these languages do not afford open-endedness because they are designed to be self-contained. On the other hand, conventional operating systems provide us with language processors and linkage facilities; that is, compilers and interpreters provide interfaces to call programs written in foreign languages, and linkers provide language processors with protocols for inter-language communication. However, conventional inter-language communication facilities only allow sending and receiving of primitive data such as numbers and characters, so that program components written in different languages cannot share information involving more complex data structures.

This paper presents a multiple programming language system that can afford open-ended environment for multiparadigm programming. The system is based on the notion of programming language complex (PLC). The following paragraphs describe the features of a PLC:

1. Programmers can choose appropriate component languages in accordance with the problem at hand.
2. Component languages are given inter-language communication protocols defined independently of the languages. Any component language needs no ad hoc primitives to call program modules -- such as functions, procedures, and predicates -- written in other component languages. This means that even if a module is re-written in another language, we need not necessarily modify other modules which call the modified one.
3. Programmers need only know the languages in which their programs are being written.
4. A PLC can accept new component languages based on new paradigms. These features, which make a PLC open-ended, are quite fruitful for multiparadigm programming, but they are not present in current multiparadigm oriented languages nor in conventional linker-based programming environments. To facilitate the development of a PLC, this paper shows an analysis of constructs from conventional programming languages and proposes to organize them into three layers. The programming paradigms discussed here include only paradigms
for sequential programming; those for parallel and concurrent processing are beyond the scope of this article.

2. Layered view of programming languages

2.1 Programming language constructs

Communication between different languages is impossible if there is no common data type nor proper type transformation rules. Fortunately, most programming languages share common data type concepts such as numbers, characters, and truth values. It is also easy to find common concepts among composite data types such as arrays and strings. On the other hand, control structures are language specific (or paradigm specific). For example, assignment is an essential concept of the procedure-oriented paradigm but is not included in the logic-oriented paradigm. We consider that "request for computation" is a common concept among languages, even though computation mechanisms are language specific. In general, languages have expressions that describe request for computation and certain units of description that define how computation is to be performed, i.e., operators, functions, procedures, methods, predicates, and so on. In this paper, such a unit of description is called a stub of computation (or simply, a stub).

For example, we usually use pointer variables to represent lists in the C language. On the other hand, binary lists are a primitive type in LISP. Provided that binary lists are primitive, C programs can simply call the primitive operations as functions, e.g., car/cdr. C programmers need not know how a binary list is implemented. For another example, assume the following Prolog rule and LISP function are given,

\[
\text{p}(x) :\text{foo}(x), \text{q}(x), \text{and} \\
\text{(defun bar (a) (foo a)) .}
\]

Both definitions include an expression to call a stub whose name is foo, i.e., \text{foo}(x) and \text{foo}(a). A stub named foo can be written in any language as long as it can receive parameters and return values in accordance with a request for computation, i.e., \text{foo}(x) or \text{foo}(a). In addition, if foo is a generic name, both of these expressions may or may not mean to call an identical stub.

In order to discuss the layered architecture, we classify language constructs into the following three categories.

1) Syntactic elements

This category is further classified into three subcategories: macros, definitions/declarations, and statements/expressions. Macros are to be preprocessed before compilation or interpretation. Definitions/declarations are such elements as stubs of computation, variables, constants, classes, and data types. Statements/expressions are such elements as control structures, assignment, input/output instructions, and expressions of various data types.

2) Instances

This category is also classified into three subcategories: data instances, program codes, and working spaces. There are two types of data instances: modifiable instance and read-only instance. Modifiable instances are variables, arrays, structures, class instances, and so on. Constants are read-only. Such elements as runtime stacks and heaps are working space.

3) Virtualization functions

This category includes the functions to virtualize physical resources in order to create logical environments where instances are realized for execution of programs. The functions are representation transformation for input/output, memory management (allocation and reclamation), and reference management (or linkage management), and so on.

2.2 Layered model of a programming language complex

Based on the discussions above, the following paragraphs show the view of layers of a programming languages to define a PLC.

(1) Common Constructs (Bottom Layer): Identification mechanism of instances must be common among component languages. This fact implies those mechanisms to define and create instances are also common. However, this does not imply the fact that any type of data instance must be interpretable in any component language. The bottom layer includes definition and creation functions of both data types and instances, and the virtualization functions except those to be included in the middle layer.

(2) Reference Constructs (Middle Layer):

This layer contains the functions to reference stubs of computation. It also contains those to access global variables shared among stubs. These functions cannot be realized without the instance identification function of the bottom layer.

(3) Language Specific Constructs (Top Layer):

Each component language has its own interpretation rules, which are to be realized as a
language processor, i.e., an interpreter and/or a compiler. Language processors of component languages are placed in the top layer, since they need to share definition of instances and to make reference between stubs written in different languages.

The layers described above are called the data abstraction layer, the reference layer, and the language layer, respectively. Fig.1 illustrates the layered model of PLC. Fig.2 and Fig.3 illustrate the model of conventional linker-based systems and the model of multiparadigm languages, respectively. Fig.2 shows that languages do not share definitions of data types, so that communication among them becomes restrictive. Fig.3 shows that they can provide a harmonized multiparadigm environment but they have no open-endedness. In the case of the layered model, all component languages share the instance space, so that a program stub written in a language can access any instance of any data type. The stub need not know the structure of the data type but only needs to know what request is applicable to the data type. Since foreign requests are passed through the reference layer, the stub need not necessarily know what stub would accept its request nor what language the accepting stub is written in.

3. Definitions of layers

Fig.4 illustrates the layers based on illustration scheme of layered architecture of computer networks. A layer contains "entities" represented by rectangles, and provides upper layer(es) with "services" represented by diamonds. In addition to the three layers introduced above, Fig.4 assumes the physical layer below the data abstraction layer and the application layer above the language layer. The former contains hardware and software environments and the latter user programs.

The Data Abstraction Layer (DL) realizes primitive constructs common among languages. We give the layer the name Data Abstraction since it provides definitions of data types that are

![Diagram](image-url)

Fig.1 Layered Model of Programming Language Complex

![Diagram](image-url)

Fig.2 Layered Model for Language Processor / Linker System

![Diagram](image-url)

Fig.3 Model of Multi-Paradigm Programming Language
abstractions of the physical environment. As shown in Fig.4, DL provides upper layers with services \textit{InstanceSpace} and \textit{PrimitiveOps}. Fig.5 shows the services and entities of DL. \textit{InstanceSpace} is a space composed of all instances, and \textit{PrimitiveOps} is a collection of operations that are primitively defined. There are three kinds of instance, DataTypes, Values and Variables. DataTypes is a set of instances that define data types both primitively given and user defined. Values is a set of immutable instances; for example, numbers, characters, truth values, program codes both compiled and to be interpreted, and so on. Variables is a set of instances whose contents are modifiable, e.g., simple variables, arrays, structures, sets, and so on.

The most significant role of DL is to define the space of instances, \textit{InstanceSpace}. \textit{InstanceSpace} is a set of instances:

\begin{align*}
\text{SERVICE} & \{ \text{InstanceSpace, PrimitiveOperations} \} \\
\text{InstanceSpace} & = \text{Set of Instances} \\
\text{PrimitiveOperations} & = \text{Set of Primitive Operations}
\end{align*}

\begin{align*}
\text{InstanceSpace} & = \bigcup (\text{DataTypes, Values, Variables}) \\
\text{DataTypes} & = \bigcup (\text{PrimitiveTypes, UserDefinedTypes}) \\
\text{PrimitiveTypes} & = \bigcup (\text{SimplePrimitiveTypes, AggregatedPrimitiveTypes}) \\
\text{Variables} & = \bigcup (\text{SimpleVariables, AggregatedVariables})
\end{align*}

\begin{align*}
\text{ENTITIES} & \{ \text{InstanceAllocation/Deallocation, TypeDefinition, MemoryManagement, Representation Transformation (for Input/Output), InstanceAccess, PrimitiveTypeFunctions} \} \\
\bigcup (A, B, \ldots) & \text{ represents a set union.}
\end{align*}

\textbf{Fig.5 Services and Entities in Data Abstraction Layer}

gives unique identifier to every instance. Equality between instances, which is one of the most atomic predicates, is defined based on their identifiers. The entities to allocate instances (InstanceAllocation/Deallocation), to manage memory space (MemoryManagement), and to perform input/output (RepresentationTransformation) give the base of the instance space. PrimitiveType-Functions gives a set of primitive functions coupled with primitive data types. TypeDefinition gives functions to create data types. InstanceAccess gives access functions to instances.

The Reference Layer (RL) realizes the reference mechanism to variables and stubs for computation (Fig.6). The reference mechanism is defined independently of each component language. RL provides two types of reference for stubs that are designated reference and generic reference. For generic reference RL receives requests for computation and selects appropriate stubs. There are several parameter passing rules such as call-by-value and call-by-reference. The scheme to cope with the difference of the rules among languages is based on unification. The scheme is discussed in the next chapter in detail.

The Language Layer (LL) realizes interpretation and/or compilation rules of component languages (Fig.7). Component languages have their own definitions of types and/or classes. Data type in DL and type/class in LL are distinct concepts. They would allow users to define new types and/or classes. These types and classes are to be defined based on the primitive data types given in DL. In the case of object oriented languages, a class description gives both the structure of and the operations on the instances of the class, i.e., a set of variables and methods. The data type definition in DL that corresponds to a class gives only its physical structure and primitive operations to create and access its instances. On the other hand, class descriptions are to be used for execution in LL; methods are executed (or compiled) by the entities of LL and execution of methods is defined based on the primitives of DL.

Processors of the component languages may be compilers and/or interpreters. Object programs produced by compilers include code requesting services of lower layers. Interpreters are defined based on services of lower layers.

4. Realizing layered architecture of programming language complex

This chapter describes a programming language complex system called MPS realized on the layered model.

4.1 Data abstraction layer

The instance space is defined in this layer, and every instance is given its own unique identifier in the space. Equality between instances is defined based on their identifiers. In the case of the MPS system discussed in 4.4, since it is implemented in Common LISP, the physical layer corresponds to a Common LISP system on a UNIX workstation.

4.2 Reference layer

RL provides four services, InstanceSpace, VariableReference, DesignatedReference, and GenericReference. InstanceSpace is the same as that is given by DL. VariableReference provides access mechanisms to variables. DesignatedReference provides calling mechanism of computation stubs explicitly designated. In this case, a calling expression is assumed to know all conditions that must be satisfied to call a target stub. The following paragraphs present the mechanism of GenericReference since other services are trivial.

Generic reference is realized by, 1) selecting a target stub, 2) passing parameters, and 3) receiving results. Every stub that can be accessed by generic reference must have a form to accept request (FAR). An expression accessing a stub is translated into a form to request computation (FRC). These steps are performed based on the bi-directional unification between FRC and FAR illustrated in Fig.8. Forward unification passes p

\[ \text{iservice}(RL) = \{ \text{InstanceSpace, DesignatedReference, GenericReference, VariableReference} \} \]
\[ \text{entities}(RL) = \{ \text{DesignatedReference, GenericReference, RefToVariables, PrimitiveOperationCall} \} \]

**Fig.6 Services and Entities in Reference Layer**

\[ \text{service}(LL) = \{ CL_1, CL_2, ..., CL_n \} \quad \text{CL}_i \text{ is a component language.} \]
\[ \text{entities}(LL) = \{ LP_1, LP_2, ..., LP_n \} \quad \text{LP}_i \text{ is a language processor.} \]

**Fig.7 Services and Entities in Language Layer**
parameters from caller to callee, and backward unification passes result in reverse.

There are several parameter passing disciplines: call-by-value (cbv), call-by-reference (cbref), call-by-name (cbn), call-by-value-return (cbvr) and unification (unify). Parameter passing disciplines that imply concurrent/parallel processing are out of the scope of this paper. The difference between cbv and cbref/cbn can be regarded as a difference of parameter types. Cbref requires passing reference values, which are values pointing to locations of actual parameters. Reference values must be defined in DL as a data type, and access mechanisms for parameters are shared among those languages that have cbref. Cbn requires passing a chunk of code to be used for evaluation. The data type of the chunk must be given as well as reference values. The difference between cbv and cbr is in the direction of parameter passing. Cbvr includes both cbv and cbr. The parameter passing rules of unification is quite different from those of call-by-X; actual parameters may have no values (uninstantiated), and formal parameters may be constant. By defining rules for uninstantiated actual parameters and those for instantiated formal parameters in addition to those of cbvr, we can get general rules for these disciplines. Thus, the bi-directional unification mechanism provides call-by-value-return extended for unification.

In general, it is not possible to directly pass an actual parameter from a cbv language, e.g., an integer value of LISP, to a formal parameter of a cbref language, e.g., an integer variable of FORTRAN, because the data types of their parameters are different. In such a case, actual parameters should be transformed to create an FRC unifiable with a target FAR, or an FAR to accept different parameter passing disciplines should be given. In both cases, FARs and FRCs are to be defined language by language and unifiability is not defined between specific pairs of languages.

Fig. 8 Bi-directional Unification

< FAR > ::= ( < Stub Name > < Parameter Type > ... )
< Parameter Type > ::= < UV > | < V > | < R > | < Q > | < val > | < oval > | < OV > | < VL > |
< Parameter Block > ::= ( < Parameter Type > ... ) . O.( < Parameter Type > ... )

Matching Condition (Notes)
< UV > ::= *UV: type AP may be any value of the specified type or void.
< V > ::= *V: type AP must be a value of the specified type.
< OV > ::= *OV: type AP may be omitted.
< VL > ::= *VL: type FP matches with a list of APs of the specified type.
< val > ::= V: value AP must be specific value.
< oval > ::= V: value AP must be specific value or omitted.
< R > ::= *R: type An AP to receive return value must be given.
< Q > ::= *Q: type An AP to receive return value can be omitted.

Notes
1. AP means an actual parameter in FRC, and FP means a parameter type in FAR (i.e., formal parameter).
2. The type specifier :type specifies data type of parameter. If :type is omitted any type of value is acceptable.
3. Void means a value of uninstantiated variables, i.e., a value of a variable that has no value.

Fig. 9 Definition of FAR
Fig. 9 and 10 show the definitions of FAR and FRC of the MPS system, respectively. Several examples of FAR and FRC are given in Fig. 11. An FAR consists of stub name and a list of parameter type specifications. A parameter type specification defines acceptable type or value. (Matching conditions are described in Fig. 9.) An FAR, in a sense, is an abstract description of a stub. RL has the library of abstract descriptions of stubs, and selects stubs in accordance with requests for computation. An FRC consists of a stub name, an actual parameter list and a return value specification. An actual parameter is a value with or without a replacement symbol. A parameter with a replacement symbol is replaced by a value in the backward unification step. Forward unification succeeds for an FAR and an FRC if their stub names are same and the types of all parameters are consistent. FAR parameters are replaced if their corresponding variables are assigned during evaluation of the stub, and the modified FAR is used for backward unification. Thus, the reference mechanism requires testing for equality of instances and consistency of data types, which are defined in the bottom layer.

In forward unification, data types of parameters are tested based on supertype-subtype relationship. If an FAR contains a value parameter and its corresponding actual parameter is a value, an equality test is performed depending on their data types. Backward unification fails only if a return value parameter is specified in the FRC and equality between the result and the return value parameter is not satisfied. If RL finds more than one FAR unifiable with a given FRC, it selects the FAR that has the most specific definition of parameters. If there are more than one FAR that are the same, RL selects an FAR based on the precedence rule between component languages.

4.3 Language layer

LL is a collection of processors of component language. No communication protocol is defined between certain pairs of component languages since the lower layers define communication protocols and provide services for communication.

\[
\text{FRC} ::= (\text{<Stub Name> } \text{<Parameter> } \ldots \text{<ReturnValueType>}) \\
\text{<Parameter>} ::= \text{<Replaceable Parameter> } | \text{<Unreplaceable Parameter>}
\]

\[
\text{<ReturnValueType>} ::= *o: \text{type} | *r: \text{type} \\
\]

Notes
1. Value means an actual parameter value.
2. \( <\text{Replaceable Parameter} > \) is to be replaced by the corresponding formal parameter value in the FAR with result.
3. \( *o \) means corresponding return value parameter in FAR is optional.
4. \( *r \) means corresponding return value parameter in FAR cannot be omitted.
5. \( : \text{type} \) means type of return values if specified.

Fig. 10 Definition of FRC

<table>
<thead>
<tr>
<th>FAR: Forms to Accept Request</th>
<th>LISP/Prolog</th>
<th>Prolog</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>(defun foo (x y) &lt;body&gt;)</td>
<td>(foo *V *V *R)</td>
</tr>
<tr>
<td>2.</td>
<td>p(+x \ a) : &lt;condition part&gt;</td>
<td>(p *UV :a *O)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FRC: Forms to Request Computation</th>
<th>LISP/Prolog</th>
<th>Prolog</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.</td>
<td>(p 1)</td>
<td>(p 1 !a *r) Unifiable. *r is to be given a. Return value from p is unused.</td>
</tr>
<tr>
<td>4.</td>
<td>(p 1 \ a)</td>
<td>(p 1 !1 !a *r) Unifiable. *r is to be given True/False.</td>
</tr>
<tr>
<td>5.</td>
<td>(p 1 \ 2)</td>
<td>(p 1 !1 !2 *r) Not Unifiable. Forward unification fails.</td>
</tr>
<tr>
<td>6.</td>
<td>foo(a b)</td>
<td>(foo a b *o) Unifiable. *o is to be given return value of foo.</td>
</tr>
<tr>
<td>7.</td>
<td>foo(a b c)</td>
<td>(foo a b c *o) Unifiable if return value of foo is c.</td>
</tr>
<tr>
<td>8.</td>
<td>foo(*x\ a)</td>
<td>(foo !void a *o) Not Unifiable if *x is not instantiated.</td>
</tr>
<tr>
<td>8'.</td>
<td>foo(*x\ a)</td>
<td>(foo !value a *o) Unifiable if *x is instantiated.</td>
</tr>
</tbody>
</table>

Fig. 11 Examples of FAR and FRC
The following paragraphs discuss several aspects of communication among languages of different paradigms.

**Nondeterminism:** Nondeterminism that implies backtracking is an important feature of Prolog. However, it is not always possible to request backtracking to the stubs defined in other languages. In general, it is difficult to add nondeterminism to deterministic languages. Nondeterministic languages should determine by themselves whether an expression is to be backtracked or not. For example, pls which is a Prolog-based component language of MPS does not backtrack atomic formulae that mean to reference external stubs.

**Return value:** Not all computation stubs return values. Some stubs return values through parameters. For example, a Prolog predicate does not necessarily return a truth value since successful completion of a proof procedure implies a true value. However, some stubs written in other languages would require return values from Prolog predicates. The bi-directional unification for generic reference bridges this gap.

**Types and class concepts:** All component languages share a set of types and classes but need not be able to interpret every element of the set. For example, LISP functions need not be able to interpret Smalltalk objects but must be able to call Smalltalk methods to manipulate the objects. Classes of object oriented languages are defined to be user-defined type in DL. Superclass-subclass relationships in LL are mapped to supertype-subtype relationships in DL.

**Polymorphism and inheritance:** The generic reference mechanism provides the polymorphic feature. The inheritance concept of object oriented paradigm implies to inherit structural and control features of superclasses. Definition of a data type in DL includes structure of an instance of the type, so that the structural inheritance is realized in DL. The generic reference mechanism of RL that has the precedence rules for stub selection provides the control inheritance. (Object oriented component languages may have their own precedence rules that are to be applied to internal reference.) Inheritance is realized by the lower layers, so that superclass-subclass relationships can be independent of languages so long as they are consistent. This means that a subclass is able to be defined in a language different from those used to define its superclasses.

### 4.4 Example: MPS system

We experimentally implemented a language complex system named MPS. The MPS system has three component languages Is, sts and pls whose semantics are defined based on LISP, Smalltalk-80 and Prolog, respectively. Fig.12 shows a small example program. Two classes person and group are defined in sts. The Is function list-members makes a list of persons of a group whose firstnames are same as a given value. The pls predicate get-members makes a list of persons of a group whose firstnames are same as that of the first person of the group. LL of MPS contains three interpreters for Is, sts and pls. Each interpreter creates a FRC requesting a generic reference from RL when it finds that an expression being interpreted means to call a stub unknown to itself. Thus, the interpreters are given no function to directly communicate with each other.

5. Discussions

We consider that open-endedness is the key aspect to make multiparadigm programming successful, because the advantage of multiparadigm programming is that programmers are freed from programming paradigms. The following paragraphs give discussion on the areas where the layered architecture has potentials.

**Software Prototyping:** Software prototyping is a process to clarify what components constitute a piece of software and how software components work[8]. Though the layered architecture imposes performance penalties, its open-endedness for multiparadigm programming is advantageous for prototyping. This is because in early stages of prototyping, programmers generally have no definite image what paradigm is appropriate to implement each component and they can select appropriate paradigms in accordance with components and/or development process. However, in conventional systems, it is difficult to rewrite stubs in other paradigms since interface definitions are to be changed. The layered architecture is primarily designed to remove ad hoc interfaces for inter-paradigm communication. From experience involving the MPS system, one of the authors found it difficult to directly get purely object oriented codes from a model; the model is called a school object base consisting of objects representing a school, professors, students, classes, lectures, units and so on, and it has tasks to update objects and to answer queries. The reason was, that, though he
intuitively knew what objects the model should have and what tasks should be performed, it was difficult for him to appropriately divide the tasks into fragments and distribute them into classes. He built the first prototype by writing sts classes for the objects and 1s functions for the tasks. Then, he augmented the class descriptions after he got a clear image from the first prototype, and he wrote queries in pls. In this case, his programming style is partly top-down and partly object oriented (or bottom-up).

(Class person ;
  class = person, superclass = "object"
  class method definition
  (new (ln fn))( (super new) init ln fn ))) ; class method = new, parameter = "ln, fn", no temporary variable

(ClassMethods
  (InstanceVariables lastname firstname)

(InstanceMethods
  (firstname? () ( ^ firstname ))
  (lastname? () ( ^ lastname ))
  (init (ln fn) () (lastname ^ ln) (firstname ^ fn ))) ; instance method init

(Class group ;
  class = group, superclass = "object"
  class method definition
  (new (name) () ( ^ ((super new name) init))) ; creation of a named instance

(ClassMethods
  (InstanceVariables mem n)

(InstanceMethods
  (add-person (p) ()
    (mem ^ (p cons mem)) (n ^ (n + 1)))
  (member? () () (mem))
  (how-many? () () (n))
  (init () (mem ^ (?list new) (n ^ 0)))
)

a. Sts Example Program

(defun list-members (group firstname)
  (check-first-name firstname I))
(let ((I (member? group)))
  (defunls check-first-name (name I)
    (if (null I)
      (1
        (if (eq name (firstname? (car I)))
          (cons (car I) (check-first-name name (cdr I)))
          (check-first-name name (cdr I)))))

b. Ls Example Program

(deflogic (get-members *g *y) :-
  (member? *g (*x . *d))
  (get-last-names (%firstname? *x)(%member? *g) *y))
(deflogic (get-last-names *name () () )
(deflogic (get-last-names *name (*pl . *p) *x) :-
  (get-last-names *name *p *x)
  (get-last-names *name (*p1 . *p) *x))
(deflogic (get-last-names *name (*p1 . *p) *x) :-
  (get-last-names *name *p *x))

Note: The formula
(deflogic (p (%fx a) *x) :- (q (%gx *x) *y)(r (%hx (%jx (*x . *y)))))
is expanded as follows,
(deflogic (p *var000 *x) :- (fx a *var000)(gx *x *var001)(q *var001 *y)
  (jx (*x . *y) *var002)(hx *var002 *var003)(r *var003))

c. Pls Example Program

Fig.12 Example Program
Multiple Paradigms of Data: The data abstraction layer has the potential ability to define data types for those data objects whose physical structures are quite different from those primitively given to languages, i.e., multimedia data such as windows, visual images, sounds and so on. For example, the X-window provides its primary interface only to C, so that we often need to know how to interface with C from other languages. In the case of multimedia equipments such as video disks, video-graphic processors, voice synthesizers and so on, they at most have interface utilities for specific languages, e.g., C, FORTRAN. Since DL is the only layer to define data types for multimedia equipments which usually require low level descriptions, we need not define those interfaces for each component language. Multimedia instances are to be equally accessible for all component languages. In addition, LL can afford sophisticated software modules as classes to realize multimedia information that would be composed of a set of multimedia data, for example a composite image consisting of video images, graphic images, sounds, and texts. These modules are to be shared among all languages.

The persistent object is a new paradigm of data for programming languages[1]. Assume that there is an object oriented database in the physical environment. We can define data types for persistent instances in LL for the database as well as for non-persistent ones. No language dependent interfaces to OODBs are necessary in LL, while existing OODBs have language specific interfaces such as C++ and Smalltalk-80[4]. Moreover, by introducing a query language in LL, all component languages can share its query facilities.

6. Conclusion

We consider that open-endedness is the most important aspect for multiparadigm programming. The layered architecture of computer networks gave us a hint in designing the layered architecture. No language can be a panacea for programming. Our method implies performance penalties. We consider that a programming language complex has the potential to be a platform for the software development process. It can assist programmers to design software modules in their favorite paradigms. It can provide an environment where software can gradually mature. The viewpoint of multiple paradigms of data is also considered important for developing modern systems. High level programming languages are, in general, designed to manipulate logical data, so that an abstraction scheme for physical data of multiple paradigms is expected to become much more important for developing modern systems. We are currently developing of new MPS system that will provide data types for persistent data and multimedia data such as video images.

Acknowledgements

The authors would like to thank Prof. Richard A. Kemmerer, University of California at Santa Barbara and Prof. Kiyoshi Agusa, Nagoya University for their help and valuable comments on this research.

References