SOME FEATURES OF PPL, A POLYMORPHIC PROGRAMMING LANGUAGE

Thomas A. Standish, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania 15213. 1969 May 13.

INTRODUCTION

I want to spend the first portion of my presentation giving you a brief summary of some of the features of the Polymorphic Programming Language.

A Thumbnail Sketch of PPL

1. Conversational
2. Has base language similar to APL
3. Extensible--has definition facilities for extending:
   a. Data Space
   b. Syntax
   c. Control Structures

First, PPL is one of the few extensible languages designed to be conversational. Conversational languages normally provide rapid response to trivial requests. For this property alone, conversation is worth having.

PPL has the standard structure of an extensible language. Namely, it is supplied with a base language and some definition facilities. The base language, in the case of PPL, is highly similar to Ken Iverson's APL, except that the mechanics of arrays are subtracted out since they can be recovered by extension.

As with other extensible languages, one's programming effort tends to go into the programming of definition sets. Programming definition sets includes defining the relevant data, the appropriate operations and a proper notation. Programming a good definition set is amazingly hard. There is a nontrivial amount of labor involved partly because the number of alternative designs is enormous, and partly because achieving conciseness and elegance in a programming language conflicts with achieving generality and power.

Extensible languages do not, in my opinion, solve the problem of how to design a good programming notation. They merely put some tools into the hands of programmers for exploring alternative definitions with greater ease than presently available. One of their principle contributions, I believe, will be to reduce the level of skill and the amount of effort needed by a programmer to implement linguistic constructs of his own choosing.

DATA

One type of definition facility provided in the base language is a notation for data definitions. One starts with atoms, which are the indecomposable data primitives, and then one constructs compound structures from them. Thus, using either atoms or previously defined data classes, one can define new data classes. As with the data definition facilities in some other extensible languages, these classes may consist of: (1) Heterogeneous Compound Structures--that is, structures with several parts of different types, (2) Homogeneous Sequential Structures--that is, structures consisting of elements of only one type indexed by an initial segment of the positive integers, (3) Unions of Other Classes, and (4) References to Members of Another Class--this last data type consisting in a formalization of the notion of an address or location in a form suitable for adsorption in a high level language. Some specific examples of data definitions in PPL are given later.

Data Definition Language

Using either atoms or previously defined data classes, one can define new data classes. These classes may consist of:

1. Heterogeneous Compound Structures
2. Homogeneous Sequential Structures
3. Unions of Other Classes
4. References to Members of Another Class

SYNTAX

In PPL there also exist mechanisms for defining extensions of syntax. These occur at several levels. First, one can introduce new monadic and dyadic operators. Second, there are syntax macros, and third, there are syntax-directed compiler-compiler techniques. I have liberally borrowed ideas from Burt Leavenworth and from Brooker and Morris in adding these last two features even though the ideas appear in PPL in slightly different dress than in the original work of Leavenworth and Brooker and Morris. One reason for this is that I feel that too few of us in designing programming languages make use of each other's work. This leads to mass production of weak versions of work already done. A second reason for including ideas from Leavenworth and Brooker and Morris is that I believe that syntax macros and compiler-compiler techniques belong in the syntax part of extensible languages. More strongly, I believe extensible languages must include the power of compiler-compilers if they are to realize their stated goals.

Syntax Extension Mechanisms
1. Introduction of new monadic and dyadic operators
2. Syntax Macros (like Leavenworth's)
3. Compiler-Compiler mechanisms (like those of the Brooker-Morris System)

I will give later an example of the form in which the syntax macro occurs in PPL. More interesting compiler-compiler examples are given in "Some Compiler-Compiler Techniques for Use in Extensible Languages," which is included after this paper as a supplement to it. (Page 55.)

CONTROL

In PPL, I have made an attempt to provide some tools for defining control structures. By control structures I mean such things as co-routines, pseudo-parallel processes, clock-driven simulation, backtracking, conversation, monitoring with interrupt capabilities, and so forth.

Control Structure Mechanisms
1. Parallel Processes
2. Continuously Evaluating Expressions
3. Control Structure Macros
4. Interrupts
5. Traps

Basically each procedure can be thought of as a pure procedure or a piece of re-entrant code. When a procedure is called, an activation record is set up for it containing information such as the values of local variables, its control counter, its global environment, its status--telling whether it is suspended or active, and so forth. Thus, two or more processes may share the same instruction set, each process being in a different stage of execution. Processes may create activation records for other processes, suspend them, resume them, pass parameters to them, and delete them. By pasting these primitives together into macros, one can write such controls as multiple parallel returns, as would be the case if a called routine returned control several times, instead of just once, to respective copies of its suspended caller. One can write co-routines, parallel calls, routines which monitor others for the attainment of a specified condition, and so forth.

The continuously evaluating expression, invented by one of our graduate students, David Fisher, is the means used to implement monitoring tasks, and Fisher has an intriguing, efficient way of computing them. Conceptually, the value of a continuously evaluating expression is recomputed instantly whenever the value of any of its constituent variables changes.

A small time-sharing interpreter executed steps on a round robin basis of all PPL processes designated as active, thus achieving the effect of pseudo-parallel processes.

SYSTEM FEATURES

PPL has some standard conversational system features. One can save one's definition sets in libraries and one can save data sets and files of programs. There is dyn-

amic debugging including tracing, monitoring, and the setting of breakpoints. When the program halts or is suspended, one may reset values of variables and then resume computation. There is text editing by context and by position. Because the system is time-shared, response time is proportional to the amount of computation requested, and, in particular, this is supposed to mean rapid response for trivial requests. Syntax can be dynamic. Namely, a certain chosen subset of defined notation is in effect at any given moment, and changes in this subset can be programmed or administered conversationally. In particular, a given syntax can be shared amongst two different meanings in effect at separate times, the switching between meanings being dynamic.

System Features

1. Libraries of Notations
2. Files and Data Sets
3. Dynamic Debugging--Tracing, Monitoring Break Points, Resetting
4. Text Editing by Context and Position
5. Response Time Proportional to Demand for Computation
6. Dynamic Syntax
7. Interpreter and Compiler--with Optional Declarations

Finally, because of the fact that the slow speed of interpretation eventually kills you in interpretive conversational systems, we provide a compiler for the base language. Although no declarations are required in either the interpretive or the compiled text, the option of giving declarations to the compiler to allow it to compile efficient code is available.

EXAMPLES

Some examples of extensions that users can create with PPL are language extensions for formula manipulation, list processing, string manipulation, file processing and simulation.

Also, new types of arithmetics may be added. For example, arithmetics to manipulate polynomials, rationals, vectors, matrices and complex numbers. All of these arithmetics may, if desired, share a common syntax for arithmetic expressions.

This ends my short sketch of the language. Let me now give a few simple examples of some of the notation and some of the mechanics.

```
    Δ SUM = ∇(N,P);S;I
[1] S←0
[2] I←1
[3] (I>N)→(→7)
[4] S←S+I↑P
[5] I←I+1
[6] →3
[7] SUM←S
[8] Δ

    SUM(10,1)
55
    SUM(6,2)
91
```

A Base Language Program

Here is a base language function for computing the sum of the Pth powers of the first N positive integers. The text of the function is highly similar to APL. There is no block structure and there is only one base language statement per numbered line.

Why use this simplistic format? There are several reasons. First, if a programmer suspends an active program, or if the system detects an error, the system must have some way of telling the programmer where it got suspended or where it stopped. There must also be available in the language convenient points at which suspension and interrupts may occur. On a machine with a low data rate, such as a teletypewriter, there must be available a terse message telling the exact point of suspension. If the machine types SUM[3], you know immediately that it halted on the third line of the function named SUM. Text broken into numbered steps presents digestible chunks for syntax directed compilation techniques. One can also study differential compilation and control structures with more ease in this relatively simple medium. Finally, I feel that the base language of an extensible language should be simple and uncomplicated for the reason that I find it hard to add new syntax to a complicated old syntax in a noninjurious way--that is, so that it blends smoothly.

Some of these reasons for starting with a simple, APL-type text are invalid once one moves out of the context of conversational languages, or once one has a device such as a scope with a high data rate.

One other point should be made here which was originally put forth by Carlos Christensen. That is, if you start with a simple base language, you must often apply a nontrivial amount of labor to extend it in nontrivial ways. For example, if you start with a base language without block structure, there is a nontrivial amount of labor involved in putting it in.

The base language program above uses a variable of accumulation S to accumulate the sum, and a controlled variable I within a simple loop. In the loop, S accumulates an additional term of the form I^P, and I is incremented by one each time. When the value of I exceeds N the accumulated value in S becomes the value of the function. The dyadic right arrow → is a conditional operator and the monadic right arrow → is a go to operator. Thus, the statement (I>N)→(→7) is synonymous with if I>N then go to 7. The ∇ is a synonym for λ, the whole function being a version of a λ-expression with bound (N,P). The phrase ;S;I declares S and I as local variables. The whole definition of SUM is enclosed in deltas, Δ:::Δ, which are non-nestable brackets enclosing all types of definitions in PPL.

Once the function is defined it may be executed conversationally. If we type SUM(10,1) and press carriage return, we call the function with arguments (10,1) and compute the sum of the first ten integers: 55. Typing SUM(6,2) and pressing carriage return computes the sum of the first six squares: 91.

```
      ΔSUM[4.5]
[4.5] PRINT(S)
[4.6] Δ

      □SUM

      ΔSUM←∇(N,P);S;I
[1]   S←0
[2]   I←1
[3]   (I>N)→(→8)
[4]   S←S+I↑P
[5]   PRINT(S)
[6]   I←I+1
[7]   →3
[8]   SUM←S
```

The text of a function may be modified by text editing. In the example here we insert a new line, PRINT(S), after line 4 of the original text. We then display the text of SUM with the expression □SUM serving as a display command, and we discover that not only has the text been renumbered, but underlined addresses in the text have been changed accordingly. These addresses are elastic, so to speak, and change according to changes in the numbering of text in such a way as to keep their target lines invariant. One can also use labels as in Algol.

In the next example we define some simple syntax macros changing the form of conditional and go to statements. In doing this, we make use of an already given syntax for expressions <EXP> and statements <STMT>. These syntax macros would cause a statement such as if I>N then go to 8 to be replaced with its base language equivalent, (I>N)→(→8). The last statement in the example, <STMT> =+ <IFST> <GOST>, causes the newly defined syntax, <IFST> for conditionals and <GOST> for go tos to be added to <STMT>, the current syntax in effect for statements. Thus, for example, we could go back and alter the text of the function, SUM. In the following example, we open the definition of the function, SUM, at line [3] using the command SUM[3]. The system responds by typing back line [3] and awaits our input. We type IF I>N THEN GO TO 8 as the new text for line [3] and then press carriage return signalling the end of the line. The system responds with the next line number in sequence, line [4]. On line [4] we specify that we wish to delete line [5] by typing [~5] and pressing carriage return. The system

```
      Δ<IFST> = IF <B:EXP> THEN <C:STMT>
[1]   (B) → (C)
[2]   Δ

      Δ <GOST> = GO TO <E:EXP>
[1]   → E
[2]   Δ

      <STMT> =+ <IFST>|<GOST>
```

```
     ΔSUM[3]
[3] IF I>N THEN GO TO 8
[4] [~5]
[6] [7]
[7] GO TO 3
[8] Δ
     □SUM
     ΔSUM=∇(N,P);S;I
[1] S←0
[2] I←1
[3] IF I>N THEN GO TO 7
[4] S←S+I↑P
[5] I←I+1
[6] GO TO 3
[7] SUM←S
     Δ
```

deletes line [5] and types back line [6] as the next line on which we may wish to enter new text. We specify [7] on line [6] causing the system to skip making any alterations on line [6] and to sequence to line [7]. On line [7] we enter a new statement, GO TO 3. On line [8] we close the function by typing Δ, and this terminates the editing process. If we now call for the revised text of the function SUM to be displayed, using the display command, □SUM, we observe that the statement, PRINT(S), has been deleted and that the new forms of the conditional and go to statements have been substituted for the old.

In the example to the right, we type the text of a function, NEWSUM, using a for-statement to replace the loop given previously in the text of the function, SUM. If we attempt to execute this function by, for example, typing NEWSUM(10,2), the system halts as step NEWSUM[2] and informs us that the statement FOR I←1 TO N DO S←S+I↑P is not in the current extension. To put a for-statement in the current extension, we define it as a syntax macro, as in the definition below, and we add the syntax, <FORST>, to the current syntax in effect for statements, <STMT>, using the command, <STMT> =+ <FORST>. We may then call NEWSUM(10,1), and its constituent for-statement works without incident, yielding the proper answer, 55, as the sum of the first ten integers.

```
     ΔNEWSUM = ∇(N,P);S;I
[1] S←0
[2] FOR I←1 TO N DO S←S+I↑P
[3] NEWSUM←S
[4] Δ

     NEWSUM(10,2)

NEWSUM[2]FOR I←1 TO N DO S←S+I↑P
         ^
***STATEMENT NOT IN CURRENT EXTENSION***
```

```
     Δ<FORST>= FOR <V:VAR>
     ←<L:EXP> TO <U:EXP> DO <T:STMT>
[1] V←L
[2] (V>U)→(→NEXT)
[3] T
[4] V←V+1
[5] →2
[6] Δ

     <STMT> += <FORST>

     NEWSUM(10,1)
55
```

The manner in which the syntax macro defining for-statements works is exemplified as follows: First, the syntax macro matches a statement such as FOR I←1 TO N DO S←S+I↑P against its syntax pattern, FOR <V:VAR> <L:EXP> TO <U:EXP> DO <T:STMT>, and derives a formal-parameter/actual-parameter correspondence as indicated below by the double arrows. Then, this formal-actual correspondence is used in an act of substitution to convert the macro body into a piece of relocatable text. The fragment of relocatable

```
[1] V←L                V ↔ I          [1] I←1
[2] (V>U)→(→NEXT)      L ↔ 1          [2] (I>N)→(→NEXT)
[3] T                  U ↔ N          [3] S←S+I↑P
[4] V←V+1              T ↔ S←S+I↑P    [4] I←I+1
[5] →2                                [5] →2
```

text is scanned to detect occurrences of statements requiring further expansion, and after further expansion, if any, it is then substituted for the original for-statement, as shown next. The substituted text is then renumbered and relocatable addresses are bound to conform to proper target addresses in the substitution environment. In par-

ticular, the address NEXT in line [2.2] is a floating address, which gets linked to the renumbered address of the next statement in sequence following the fragment of substituted text.

```
     ΔNEWSUM =∇(N,P);S;I
[1]  S←0
[2]  [2.1] I←1
     [2.2] (I>N)→(→NEXT)
     [2.3] S←S+I↑P
     [2.4] I←I+1
     [2.5] →2
[3]  NEWSUM←S
     Δ
```

```
     ΔNEWSUM=∇(N,P);S;I
[1]  S←0
[2]  I←1
[3]  (I>N)→(→7)
[4]  S←S+I↑P
[5]  I←I+1
[6]  →3
[7]  NEWSUM←S
     Δ
```

We observe that after expansion of the for-statement in NEWSUM, the expanded text is identical to the original text of SUM given earlier.

EXAMPLES OF DATA DEFINITIONS

```
Δ STRING = INDEFINITE×[CHAR]Δ
Δ VECTOR(N,T) = N×[T]Δ
Δ REALMATRIX(M,N) = VECTOR(M,VECTOR(N,REAL))Δ
Δ PRISONER = [NAME:STRING,RANK:CHAR,SERNO:INT]Δ
Δ VECREF = REF(VECTOR)Δ
```

In the first definition, a string is defined to be a homogeneous sequential structure consisting of an indefinite number of characters. The number of characters is specified at the time the representation of the string is constructed. The second definition also defines a class of homogeneous sequential structures called vectors, each having N elements of uniform type, T. The first two definitions exemplify the syntax for defining classes of homogeneous sequential structures. The third definition defines a class of M by N matrices of elements of type real, using a vector of vectors of reals. Thus the second definition has been used in nested fashion to get the third The fourth definition defines a heterogeneous compound format having three components of different types: a name component which is a string, a rank component which is a character, and a serial-number component which is an integer. The final definition defines a class of data consisting of references to vectors.

Constructors

```
V ← VECTOR(5,INT)(3,-4,77,2,89)
M ← REALMATRIX(2,2)((2.3,-4.5),(7.33,-56.2))
P ← PRISONER('DOE,JOHN F.','M',1234567)
```

Selectors

```
V[4]      M[2,1]      NAME(P)
```

Predicates

```
B ε PRISONER
V ε VECTOR(7,REAL)
M ε REALMATRIX
LENGTH(V) > 7
```

A consequence of making these definitions is that certain functions are created over the data classes defined. In particular, for each data class, constructors, selectors, and predicates are created automatically. The constructors manufacture representations of the defined data given suitable arguments. The selectors are the accessing or decomposition functions that select parts of compound structures. The predicates test arbitrary data to ascertain if they are of a specified form.

In the examples above, the constructors build, respectively, a vector of five integers, a 2 by 2 matrix of real numbers, and a record of information for a prisoner. The shape of the argument list must be appropriate to the type of object being constructed and may be nested. The selectors access their components either by ordinal

selection in the case of a homogeneous sequential structure indexed by an initial segment of the positive integers, or by field name, in the case of heterogeneous structures with named components. The predicates test values to see if they are members of defined data classes. Finally, the length of homogeneous sequential structures is available as the value of expressions of the form, LENGTH(V).