24

The Design Philosophy of POP-2

R.J. Popplestone
Department of Machine Intelligence and Perception
University of Edinburgh

INTRODUCTION

POP-2 (Burstall & Popplestone, 1968) represents a fairly far-reaching revision, extension and systematization of the author's POP-1 (Popplestone, 1968). The thoughts expressed here consequently represent a point of view elaborated jointly with my co-designer of POP-2, Dr R. M. Burstall.

AIMS

POP-2 is a language to be implemented on real machines, using modest resources of manpower. An implementation of the language must be possible which permits large problems to be tackled. This implementation must not be too inefficient in its use of machine time, or too profligate in its use of store. The language must also take into account such properties of real machines as overwritable store–that is to say it must not be a purely constructive language: it must allow assignment.

POP-2 handles a large range of structures such as list cells (cf. LISP), functions (cf. CPL) and records (called beads in AED). Efficiency is important here–LISP is perfectly general, but its representation of such structures in list-cells is inefficient in its use of store.

Bearing in mind the need for a wide range of structures, the concepts should be as few and simple as possible. An example of this simplification is that such apparently diverse things as ALGOL procedures, ALGOL arrays, binary operators, and AED 'components' can all be represented in a natural fashion as POP-2 functions. Functions in POP-2 differ in their semantic properties, and in the syntactic properties of their names, but they are all the same in essence: not only are they all handled in the machine as data structures, but they constitute a single class of data structure.

This brings us to the subject of items. Anything which can be the value of a variable is an item. *All items have certain fundamental rights.*

1. All items can be the actual parameters of functions

2. All items can be returned as results of functions
3. All items can be the subject of assignment statements
4. All items can be tested for equality.

Equality does not, however, follow the usual mathematical axioms, e.g. CONS(2,3)≠CONS(2,3), where CONS is the LISP list-cell constructor. The impracticability of providing a computed version of mathematical equality is clear if one considers the case of functions. Nevertheless, care is taken in the definition of the language to make sure that the meaning of the equality is clearly defined. It corresponds to the LISP 'EQ', or the concept of equal address in mechanistic terms.

It should be noted where existing languages fail to provide an adequate 'item's charter of rights': ALGOL arrays cannot be returned as results of procedures. AED components cannot act as parameters of procedures. Certain consequences follow from the charter of rights given above. Thus in POP-2 we can say SQRT→PIG; and later discover that PIG(144) has the value 12. It should be noted also that certain words and symbols in POP-2 are *not* the names of items. Thus, while +, LOG, CONS, are the names of items,), THEN, and GOTO, are not. Statement labels are not items: i.e. 'computed gotos' are not permissible. The same effect can be obtained by exploiting the freedom to assign to functions. The role of non-item words is syntactic.

A consequence of clause 2 of the items' charter of rights is that POP-2 requires a more general form of storage control than is provided by a stack or push-down store. In fact a generalized form of the LISP scheme is provided. An ALGOL array can be placed on a stack precisely because it cannot survive the activation of the procedure or block in which it occurs. This has a profound effect on the class of problems which it is possible to tackle in a natural way using POP-2. An important corollary is that the space occupied by the code which represents functions is itself under the general storage control scheme.

The syntax of the language should be neat, unobtrusive and simple, rather than elaborate. Additional semantic power should not be locked away from the user by syntactic devices. Thus a POP-2 array is created by applying a function NEWARRAY to a description list, and assigning the result to a variable, none of which operations involves any new syntax.

POP-2 is designed for on-line use. The language features which reflect this are (1) the ability to have expressions executed immediately, and (2) the limited block structure of the language. A POP-2 program tends to be a sequence of independent function definitions, which can be redefined independently.

ITEMS

Items are simple or compound. This distinction is made for practical reasons: a simple item is an economical quantity of information to move around a

machine, a compound item is not. Thus the limit to what is to be regarded as a *simple* item has been set with an eye to the amount of information that can be put in one word of a machine.

Items are classified into *types*. The words *item class* and *type* are synonomous. It is possible to determine the type of any item at any time. Except in the case of functions, it is not possible to restrict the range of variables to any particular type of item. This means that type checking must be done dynamically, that is to say, all functions must check that they have been supplied with appropriate arguments. This leads to reduced efficiency. The other reason for attaching types to variables is to permit errors to be discovered at compile-time. The conceptual type structure of the programmer is very much finer than the type structure of languages like ALGOL. Thus when we say 'let *s* be the speed of the vehicle *v*' or 'let *w* be the weight of the house *h*', *s*, *v*, *w*, and *h*, are, at one level, of different type, and a language that fully mirrored the thought processes of the human would allow this distinction to be made. Thus it is felt that if POP-2 were to have type restrictions on variables at all then it must have a type structure extendable by the user.

However, type structure is not simple. For instance, it is meaningful to say 'let *w* be the weight of the vehicle *v*' but it is not meaningful to say 'let *s* be speed of the house *h*'. Thus any extended type structure would be too complex to put in what is conceived of as essentially a simple language. Not only would it be too complex, it could be too cramping. One could not define a function MAPLIST to apply any function to all members of any list if one had to specify the types of its arguments. Indeed, the ability to write 'metaphorical programs' where a set of functions for, say, performing a search procedure can be used with widely differing types of object being searched for, is very important in machine intelligence. Thus on the one hand, Burstall's (1968) program for fact retrieval uses one set of functions for both syntax analysis and deduction, while on the other hand the Graph Traverser, an ALGOL program (Doran and Michie, 1966) is handicapped by operating solely on rectangular arrays.

Numbers, integer and real, are simple. For reasons of efficiency the truth values TRUE and FALSE are represented as the numbers 1 and 0 respectively, and as such are simple. There are also operations which allow numbers to be treated as bit-patterns.

Type conversion between integer and real items is performed automatically at run-time.

Compound items can be thought of mechanistically as pointers to areas of store. Certain standard compound items are provided. These include

List cells

List cells and operations of the LISP type are provided. There are, however, a class of entities, which are variously called streams or files, the processing of which is sequential, and which it is desirable to handle in a manner analogous to the manner in which lists are handled. It is not practicable to convert these

things into lists because of the prohibitive amount of store they would occupy. Landin (1966) proposed a representation in terms of functions without side effects. In POP-2 a more efficient solution is provided: *dynamic lists*. These are lists of the LISP type, except that the last cell in the list contains a function. There are operations HD and TL on dynamic lists, which, like CAR and CDR of LISP, select respectively the first member and the remainder of the list. When the last cell of a dynamic list is reached, the resident function is applied to no arguments, and the result is taken to be the next member of the list.

The list is terminated when the function produces TERMIN as its result. In this case TRUE is written into the first position in the last cell. The function NULL which tests for the emptyness of a list, returns the result TRUE when it is given a list cell with TRUE in its first position and a function in its second.

As an example, suppose *f* is a function to read a character off an input tape. The statement

```
FNTOLIST(F)→X;
```

using the standard function FNTOLIST, will set X to be a dynamic list built from F. The function COUNT defined by

```
FUNCTION COUNT X;
  VARS N; 0→N;
LOOP: IF NULL(X) THEN N EXIT
  IF HD(X)=HD(TL(X)) THEN N+1→N CLOSE
  TL(X)→X; GOTO LOOP
END
```

which counts the number of pairs of identical adjacent characters on the input tape, is only fractionally more convenient than the same function written using F directly, but the gain increases with the complexity of the operations being performed.

Words

System routines are available for converting sequences of symbols into data cells called words. These contain the first eight characters of the word. Characters are converted into words according to the conventions of the POP-2 compiler. Thus CAT), +, and ++ are words. Quotes are used to distinguish words from identifiers e.g. "CAT" denotes the word CAT while CAT denotes the current value of an identifier CAT.

Words are *standardized*. That is to say, on construction of a word, a check is made to see whether a word having the same sequence of letters has been encountered before, and if so the previous incarnation is used. Constant words are quoted in program, and "DOG"="DOG" always produces TRUE, because of this standardization. There is a function MEANING on words which associates an item with a word (similar to the property list of a LISP atom). "CHIEN"→MEANING ("DOG").

This function allows one to construct an updateable dictionary, the entries of which are undefined until assignment is first made to them. This example of assignment is analogous to X→HD(Y).

Users may themselves define classes of compound items.

Records

Records are data cells with a number of components fixed over a *record class*. Thus the class of list cells is a record class each member of which has two components, namely, HD and TL. The amount of information in each component of a record is specified in a description list, which is handed to the function RECORDFNS which is used to create a record class. The description list for list-cells would be [0 0], 0 in a description list being used to indicate a component which can be a full compound item. The result of RECORDFNS is a number of functions for manipulating the records of the class being defined. Thus the class of list-cells would be defined by

```
RECORDFNS("LIST", 1000, [0 0])→BACK→FRONT→DEST→CONS;
```

where FRONT and BACK are CAR and CDR in LISP, and DEST is a function that 'explodes' a list-cell into its components.

Strips

Strips are cells with a number of components which is variable over a class of strips. The components of a strip are all of the same size, and are accessed by a subscripting function associated with each strip class. Thus, if S is a strip, W is an integer and SUBSCR is a subscripting function for the class of strips to which S belongs, then SUBSCR(N,S) produces item N of S.

Arrays are functions which have strips attached to them as work-space. The way in which this attachment is performed will be described in the section on functions.

There seems to be a need for classes of objects with properties intermediate between strips and records. These would have the variable size of strips, and the different component types of records. Thus, a record of a person might have a selector-subscriptor NAMECHAR, such that NAMECHAR(I,P) took the I'th character of the name of the person P.

FUNCTIONS

Functions are compound items, and as such are the values of variables. LAMBDA X; X*X END→SQUARE; (for lambda notation *see* Landin, 1966) assigns that function which takes X onto its square to the variable SQUARE. A 'sugared' version of this is

```
FUNCTION SQUARE X; X*X END
```

Binary operations like + − <> (<> is the infixed version of concatenate as applied to lists) are identifiers which are the names of functions. These identifiers are recognized by the compiler as naming infixed operations. The

modifier NONOP causes them to be treated as normal functions, e.g. NONOP * → PROD; causes the value of PROD to be set to the multiplication function. Note that only the semantic, not the syntactic, content of * passes to PROD in this assignment, so that to apply the new function we must say PROD(X,Y). If we had wished PROD to be an infix, we would have proceeded as follows:

```
VARS OPERATION 3 PROD;
NONOP * → PROD;
```

now X PROD Y=X* Y gives the value TRUE. The numeral in this declaration sets the precedence.

Some functions have a meaning when used to the right of an assignment arrow. Thus 4→A(3,2) is meaningful if A is an array, or 34→HD(L); if L is a list. Every function has a component called its UPDATER. Suppose we are keeping records of people in lists, giving details of their ages and sexes. Thus: [SOCRATES MALE 2436] or [SATAN MALE 100000]. One can define the functions AGE and SEX on these records. Thus:

```
FUNCTION SEX X;
  HD(TL(X))
END
```

Everyone who has an age has birthdays. According to *Malleus Maleficarum* (Institoris, 1948), Satan, for reasons which need not detain us here, needs to change his sex frequently. The functions AGE and SEX as defined above will not perform this updating operation. However the statement

```
LAMBDA X Y; X→HD(TL(Y)) END→UPDATER(SEX);
```

will change the UPDATER component of SEX in such a way that if we say

```
[SATAN MALE 100000]→P;
```

then later

```
"FEMALE"→SEX(P);
```

will update the record P as required.

There are two methods of creating new functions dynamically. One is POPVAL, which is a function which compiles program, in the form of a list. The other is *partial application.*

Suppose that F is a function of n arguments: we may 'partially apply' it to an argument list of only m elements by writing F(% X_1, X_2, ... X_m %). The result is a new function of n-m arguments, obtained by 'freezing' the values of the last m arguments to the values possessed by X_1, X_2, ... X_m at the time of the partial application. Thus suppose DIST is a function for finding the distance between pairs of places, and EDINBURGH is a place, then DIST(% EDINBURGH %) is a function for finding the distance of a place from Edinburgh, and we would be at liberty to assign it, e.g. DIST(% EDINBURGH %) →MILEAGE. MILEAGE(LONDON) now gives 380.

The following illustrates the use of partial application in defining a functional. Suppose ZERO is a function for finding one zero of a function. Consider the function INVERSE defined by

```
FUNCTION INVERSE F;
  AUXINVERSE(% F %)
END
```

where AUXINVERSE is the function defined by

```
FUNCTION AUXINVERSE Y F;
  ZERO(LAMBDA X; Y-F(X) END)
END
```

Then, leaving aside the niceties of numerical analysis, INVERSE is a functional which takes a function of one argument over the reals onto its inverse. Thus INVERSE(SIN) is ARCSIN. This is deduced as follows: according to the definitions INVERSE (SIN) is AUXINVERSE(% SIN %), that is, AUXINVERSE with F 'frozen' to SIN, that is a function with the same result as

```
LAMBDA Y; ZERO(LAMBDA X; Y-SIN(X) END) END
```

Given Y, the result of this function will be that X for which Y−SIN(X)=0, that is that X for which SIN(X)=Y, that is ARCSIN(Y).

The above definition of INVERSE will seem unnecessarily complicated on first reading. Could F not be a non-local of AUXINVERSE and the partial application be dispensed with? If this were so, when we came to evaluate ARCSIN the value of F might have been changed, for instance by another application of INVERSE to, say, LOG. However, with the partial application in place, if we say INVERSE(LOG)→EXP, then ARCSIN and EXP are two functions which are obtained from AUXINVERSE by freezing F to the values SIN and LOG respectively.

Landin (1966) uses an equivalent scheme. The frozen formal parameters of POP-2 correspond to the environment part of his closures. While his method, as embodied in CPL, is syntactically more elegant, partial application technique of POP-2 is perhaps more flexible, because the user can decide which variables are to be frozen.

FUTURE DEVELOPMENTS

Now that an implementation of POP-2 is available on a machine (an Elliott 4120) it is time to consider the future. There are three lines of development. The first is to make extensions to the language to make use of backing store and to make more efficient use of machine time.

The second is to build a library of *functions* to be used as building bricks of intelligent programs. The increased power that partial application gives POP-2 is very important here. In languages like ALGOL, one can write learning *programs*: it is difficult to write learning *procedures*. This is because the only *memory* that a procedure can have to itself is in the form of 'own' variables.

Since these have only one incarnation, the learning mechanism enshrined in a procedure can only be used in one sub-problem of the whole problem. In POP-2 on the other hand, a learning mechanism encoded in a function can be attached to data concerned with many sub-problems by partial application. Arrays may be regarded as *rote-learning functions*, a point of view implicit in the 'memo function' concept proposed by Michie (1968).

One can write in POP-2 a function which one might call, by analogy with NEWARRAY, NEWINTERP. NEWINTERP() produces a function called an interpolator. Suppose we are studying devices called tripistors. These pass a current that depends on the applied voltage. If we say NEWINTERP()→ TRIPISTOR; we have a function which we can teach to represent tripistor behaviour. Suppose we have observed that a tripistor passes a current of 10 amps when 1 volt is applied, and 0 amps when 0 volts are applied, then this information is conveyed by the statements:

```
10→TRIPISTOR(1); 0→TRIPISTOR(0);
```

If we now ask for TRIPISTOR(0.5) the machine produces the estimate of 5, by linear interpolation. As further information is fed in so better interpolations will be possible. Meanwhile, in another part of the program, the same interpolation mechanisms have been learning the behaviour of livertrons. When these lessons have been adequately learnt, they can be applied in predicting the behaviour of a circuit containing both tripistors and livertrons.

Finally, as POP-1 was used to develop POP-2, so POP-2 will be used to develop its successor. What sort of language will this be? One can regard existing algorithmic languages including POP-2 as being *imperative*. They are used to describe the solution to a problem. The next generation of languages will be *indicative*. They will be used to describe a problem.

Both kinds of language have been used by logicians: the lambda calculus is their imperative language, the predicate or functional calculus the indicative languages. Algorithms for interpreting these languages are known – e.g. the resolution principle for the functional calculus. Let us consider the following statements in BNF–which can be regarded as an indicative language:

```
<NP>::=THE <NOUN>
<NOUN>::=CAT|DOG
```

We can regard this as a 'sugared' version of the following statements in functional calculus.

```
∀ x (ISNP(x) <=> (∃ y (ISNOUN(y) and x=[THE] <> y)))
ISNOUN([CAT]) and ISNOUN([DOG])
```

In order to use resolution, these have to be rewritten in conjunctive normal form. Leaving out the => of the <=> we get:

```
(1) not(ISNOUN(y)) or not ([THE] <> y=x) or ISNP(x)
(2) ISNOUN([CAT])
(3) ISNOUN([DOG])
```

These statements can be used analytically or generatively. To use them analytically, suppose it is conjectured that ISNP([THE] <> [CAT]), i.e. that [THE] <> [CAT] is a nounphrase. Then one adds the denial of this statement to 1–3 and asks for a contradiction.

(4) **not**(ISNP([THE] <> [CAT]))

Resolving (4) with (1) gives

(5) **not**(ISNOUN(y)) **or not**([THE] <> y=[THE] <> [CAT])

Resolving (5) with (2) gives

(6) **not**([THE] <> [CAT]=[THE] <> [CAT])

a contradiction, by resolution with $x=x$—an equality axiom.

To use (1)–(3) generatively one need only deny the existence of a nounphrase.

(4a) **not**(ISNP(x))

Resolving (3) with (1) gives

(5a) **not**([THE] <> [DOG]=x) **or** ISNP(x)

Resolving (5) with $x=x$ gives

(6a) ISNP([THE] <> [DOG])

a contradiction with (4a), and an instance of a nounphrase.

While the deduction process in such an indicative language is algorithmic, the choice of deductions to be made is not. Foster, in this volume, proposes a method whereby a user of such a system could control the direction of deductions. With sufficiently rigid control one would have an imperative language.

Work at Edinburgh would be concerned with a heuristically controlled system. The study of imperative systems viewed as highly constrained indicative systems seems important to the understanding of milder heuristic constraints.

Acknowledgments

The research described in this paper is sponsored by the Science Research Council. I am particularly in debt to my colleague Dr R. M. Burstall for his many helpful suggestions and criticisms.

REFERENCES

Burstall, R. M. (1968), A combinatory approach to relational question answering and syntax analysis. *Proceedings of Nato Summer School Copenhagen.* Amsterdam: North Holland Press.

Burstall, R. M. & Popplestone, R. J. (1968). The POP-2 reference manual. *Machine Intelligence 2*, pp. 205–46 (eds. Dale, E. & Michie, D.). Edinburgh: Oliver & Boyd.

Doran, J.E. & Michie, D. (1966), Experiments with the Graph Traverser program. *Proc. R. Soc.* (A), **294**, 235–59.

Institoris, H. (1948), *Malleus Maleficarum* (translated M. Summers). London: Pushkin Press.

Landin, P.J. (1966), A lambda-calculus approach. *Advances in programming and* non-numerical computations, pp. 97–141. Oxford: Pergamon Press.

Michie, D. (1968), *Memo functions: a language feature with rote-learning properties.* Submitted to *Proc. IFIP* 68. Also available as *Research memorandum MIP-R-29*, Edinburgh: Department of Machine Intelligence and Perception.

Popplestone, R.J. (1968). POP-1: an on-line language. *Machine Intelligence 2*, pp. 185–94 (eds. Dale, E. & Michie, D.). Edinburgh: Oliver & Boyd.