CHAPTER 6

POPLOG's Two-level Virtual Machine Support for Interactive Languages

Robert Smith, Aaron Sloman*, John Gibson
School of Cognitive and Computing Sciences,
University of Sussex, Brighton, England
**Now at School of Computer Science, the University of Birmingham, England.*

ABSTRACT

Poplog is a portable interactive AI development environment available on a range of operating systems and machines. It includes incremental compilers for Common Lisp, Pop-11, Prolog and Standard ML, along with tools for adding new incremental compilers. All the languages share a common development environment and data structures can be shared between programs written in the different languages. The power and portability of Poplog depend on its two virtual machines, a high level virtual machine (PVM—the Poplog Virtual Machine) serving as a target for compilers for interactive languages and a low level virtual machine (PIM—the Poplog Implementation Machine) as a base for translation to machine code. A machine-independent and language-independent code generator translates from the PVM to the PIM, enormously simplifying both the task of producing a new compiler and porting to new machines.

1. INTRODUCTION

During the late 1970s and early 1980s the AI group at Sussex University needed an AI environment for teaching and research. We adopted the policy of using general-purpose computers (initially PDP11, then VAX (first running VMS, then later Unix), then Sun and other workstations) rather than Lisp machines partly because we could not possibly afford enough Lisp workstations for all our staff and students, and partly because we desired maximum flexibility and independence from particular manufacturers.

The system had to be approachable enough for totally naive first-year students, including non-numerate Arts and Social Studies students, yet powerful enough to support advanced teaching and our own research and software development. It had to be usable both on time-shared machines with dumb VDU interfaces, and on powerful workstations with bit-mapped displays, including those supporting the X Windows system. Both our research and teaching required access to a variety of AI languages, and some of our projects required mixed language programming (e.g. Pop-11 and Prolog) so the system had to support several different languages sharing a common development interface, common data-structures, etc. Because so many AI projects do not start from a well-defined specification, we wanted a system that supported rapid prototyping and exploratory development for the purpose of clarifying a problem, which meant using interpreters or incremental compilers, preferably with the convenient interface of an integrated editor. Because we were a large and growing AI community facilities for sharing and re-using software and documentation were important.

An interface to subroutines written in conventional languages such as C, Fortran or Pascal, or libraries such as NAG, was also required especially for work in speech or vision.

Moreover, because AI research continually points to the need to develop new languages tailored to specific applications, the system had to make it easy to implement new languages with a good development environment.

And finally, because new machines were becoming available it had to be comparatively easy to port.

This demanding array of requirements is, as far as we know, not met by any other system. Some operating systems, like VMS and Unix provide mixed language development environments, but they directly support only "batch-compiled" languages, not incremental compilation. That is to say, program files have to be compiled to object files, then later all the object files are linked into a runnable image. By contrast an incremental compiler allows the source language to be used as a command language and allows procedures to be edited and re-compiled or new procedures added without re-linking the whole system. This enormously speeds up development and testing of software.

Unix does have the advantage of being portable, but anyone developing a new compiler to run under Unix will normally have to write the code-generator for each new host machine. This is not required if, for instance, an interpreter is written using a language like C, which is normally provided with Unix systems, but for many purposes an interpreted language will run too slowly, compared with a compiled version.

Several AI systems based on Lisp have provided more than one language, in an interactive development system, but this is usually done by writing, in Lisp, interpreters for the other languages. Although programs written in Lisp can run fast in such a system (provided that the Lisp system is well engineered) programs written in the other languages will be slowed down significantly on account of being interpreted. By contrast the Poplog architecture does not favour one language over others. All the languages are compiled to machine code for maximum speed of execution (though of course interpreters can be written if required, e.g. for reduced space or increased flexibility).

Poplog allows data-structures to be shared between programs written in different languages because they run in the same process, with the same address space. For some kinds of programs, especially where there is no requirement for very rapid communication between sub-systems, it might be preferable to have different languages running in different processes, which would allow distribution over different machines.

The remainder of this paper describes the two-level virtual machine architecture of Poplog, which makes it possible to meet all the requirements sketched above.

2. THE POPLOG VIRTUAL MACHINES

Like other compiler systems that must accommodate several languages and be portable across a wide range of computer architectures, Poplog employs an intermediate virtual machine. Steel [1960] proposed the use of an UNCOL (UNiversal Computer Oriented Language) to reduce the effort required to implement a new language or to produce code for a new architecture. Using this, to implement L languages on M machines requires L front ends (each translating one source language to the intermediate language) and M back ends (each translating the intermediate language to one machine language) rather than having to write L*M complete compilers.

The implementors of the Amsterdam Compiler Kit (ACK), Tanenbaum et al. [1983], claim that the original UNCOL concept failed because of a desire to accept all languages and all machine architectures using a single virtual machine language. The success of the ACK, which uses a compiler intermediate virtual machine, EM, is claimed to be due to restricting the system to algebraic languages and byte-addressable machines. Poplog is currently only implemented on byte-addressable machines with a 32-bit word size. This restriction could be relaxed (although there may be some efficiency penalties as discussed in section

C.2.). So far, however, no processor without these attributes has appeared for which there was a need to port Poplog.

The range of languages supported by Poplog is much more diverse than that of the ACK. These are Pop-11 (a Lisp-like language with a Pascal-like syntax), Prolog, Common Lisp, Standard ML and sysPOP (an extended dialect of Pop-11 used for the Poplog system sources, as described later). Users have implemented additional languages and tools, such as Scheme (a statically scoped dialect of Lisp), KDL (a frame-based knowledge description language), Flavours (an Object-Oriented extension to Pop-11), KERIS (a collection of knowledge engineering tools extending Common Lisp), RBFS (a Rule Based Frame System produced by Brighton Polytechnic), a vision toolkit developed at Reading University, and various special purpose languages, e.g. a control language developed for a real time expert system project. FLEX, a Prolog-based expert system toolkit developed by Logic Programming Associates, is now available on Poplog, as is RULES, a rule induction program produced by Integral Solutions Ltd.

One apparent disadvantage of Poplog at present is that from the point of view of users it does not support languages, like C, that permit direct pointer manipulation, since the automatic garbage collection mechanism could at any time re-locate an object thereby invalidating a pointer to one of its fields. This restriction is removed in sysPOP, which allows knowledgeable system programmers to manipulate pointers and offsets, since these mechanisms are required for implementing Poplog in any case. It would also be possible to add a pointer data-type to enable user programs to simulate pointer manipulation. This would make it possible, for instance, to add an incremental compiler for a language like C to Poplog, in order to speed up development of C programs.

An improvement to the original UNCOL strategy is the use in Poplog of two virtual machines rather than one. The connection between these is shown in Fig. 6.1.

All the compiler front ends produce code for the same high level Poplog Virtual Machine (PVM). The high level VM is strongly language oriented and consequently the front ends are straightforward to write. The high level VM code is translated to code for the low level Poplog Implementation Machine (PIM). The PIM is machine oriented, and translating from its virtual machine instructions to target machine code is trivial: indeed many only require one instruction on the VAX.

As will be explained later, there are actually two different compilation routes between the PVM and the PIM, one used for incremental compilation of user procedures, and one used for batch compilation of the system sources, which are mostly written in the sysPOP dialect of Pop-11. Much effort can be spent on improving the phase of code

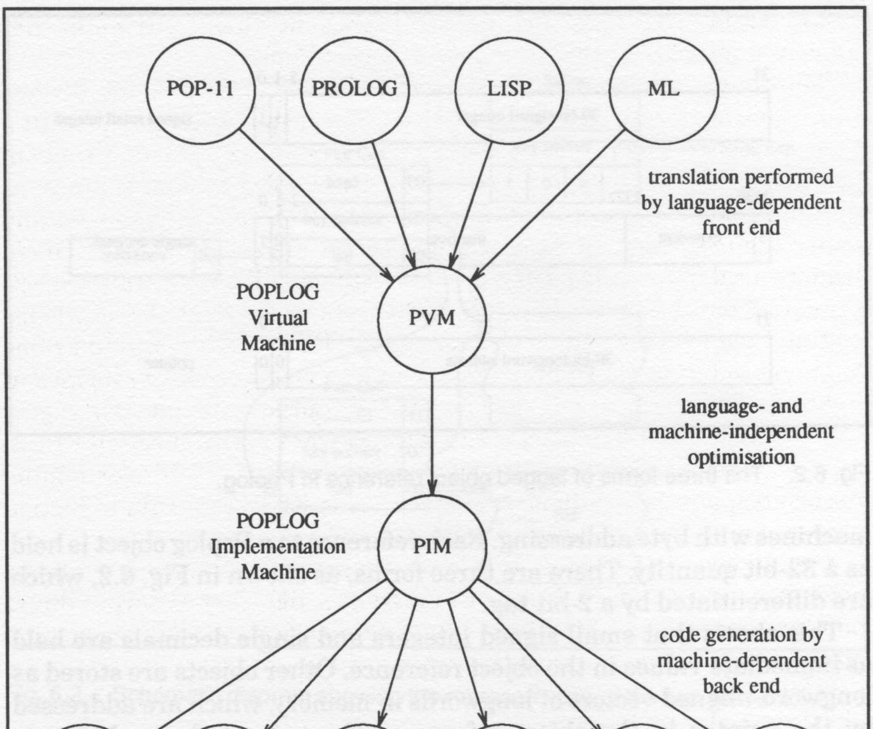
POP-11
PROLOG
LISP
ML
translation performed by language-dependent front end
POPLOG Virtual Machine
PVM
language- and machine-independent optimisation
POPLOG Implementation Machine
PIM
code generation by machine-dependent back end
VAX
680?0
SPARC
80386
CLIPPER
MIPS

Fig. 6.1. The Poplog Virtual Machines.

generation between PVM and PIM because this translation is both language and machine independent, so work done at this level benefits all implementations.

The next two sections describe the data areas in Poplog and the scheme for data representation. This will help in understanding the more detailed description of the two virtual machines which then follow.

2.1 Object Representation

Most of the Poplog languages are untyped and thus type-checking must occur at run time. The following scheme for data object representation is used for all current implementations of Poplog, these being on 32-bit

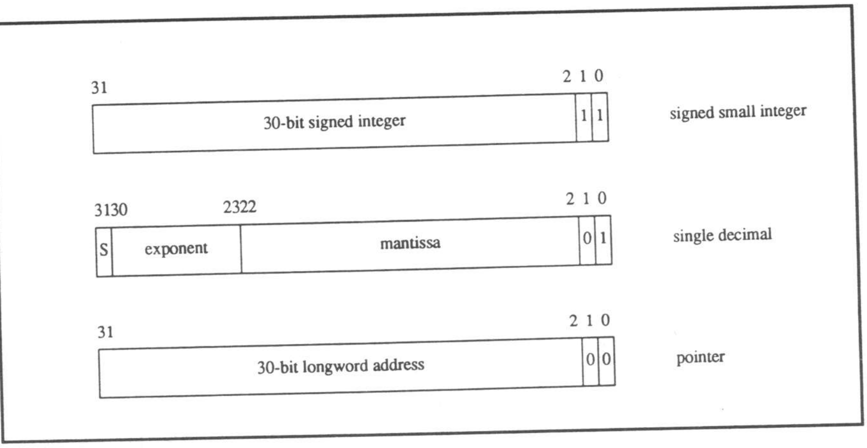

Fig. 6.2. The three forms of tagged object reference in Poplog.

machines with byte addressing. Each reference to a Poplog object is held as a 32-bit quantity. There are three forms, as shown in Fig. 6.2, which are differentiated by a 2-bit tag.

This shows that small signed integers and single decimals are held as immediate values in the object reference. Other objects are stored as longword-aligned vectors of longwords in memory, which are addressed by the pointer in the object reference. The type of these objects is determined by the contents of the second field of the vector, the "key" field, which is actually itself a pointer to a structure that defines the type of the object. If two objects have a matching key field then they are of the same type. The object reference actually points to the third longword of the object. This equates to the start of data for strings and other vector-based types, thus allowing these to be passed unmodified to and from "externally loaded" (i.e. imported) foreign procedures written, for example, in C or Fortran.

Not all of the object vector need contain other object references. Signed and unsigned bitfields, bytes, words, longwords and machine instructions (in the case of procedure objects, which are first class objects) can be present, and information about this structure, held in the key, will be used by the garbage collector when tracing live objects. As an example, Fig. 6.3 shows the representation of the two element list ['foo', 1].

Each key structure contains information about the class of objects which it defines, including functions defined for that data type: e.g. constructor, accessor and print functions. A large number of data types are provided, and the user is free to create new compound types of vectors or records, and to change some of the class-specific functions,

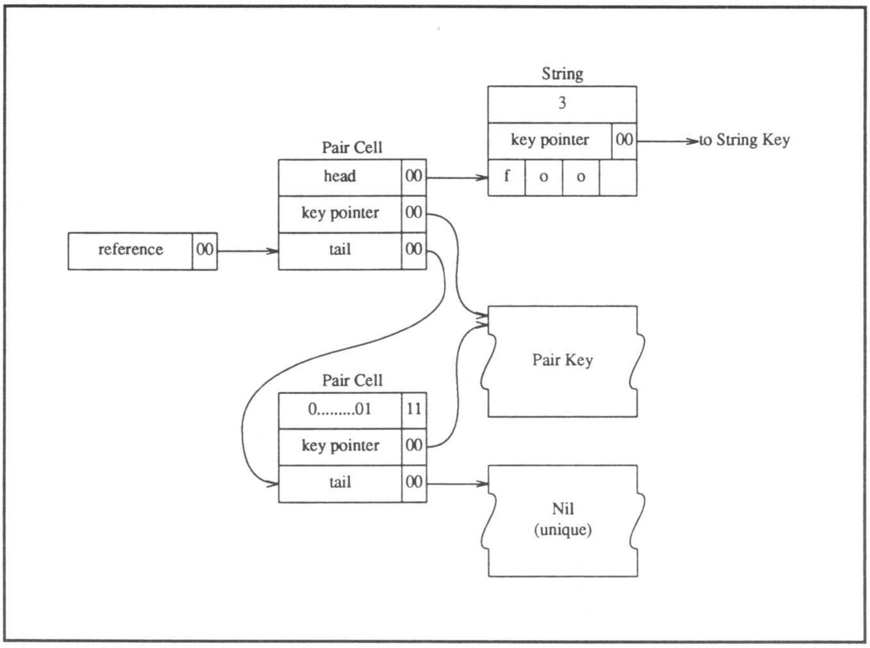

Fig. 6.3. Schematic diagram showing the representation of the two element list.

e.g. functions for printing, equality testing, or the "class_apply" function which determines what is to happen if a structure is treated as a procedure and executed.

The advantages of the current scheme of object representation are:

- A pointer to an object (including the tag bits) is equivalent to the address of the third longword of that object, and thus no tag manipulation is required in this case. Additionally, only the least significant bit of the reference need be tested to determine if a reference is a pointer or a simple object.

- Most architectures can test, insert and extract the 2 least significant bits efficiently with quick arithmetic/logical instructions. Other tagging schemes generally require shift operations to implement, which are generally slower.

- The 2 tag bits of a pointer reference are available to the garbage collector to hold the mark and shift bits during garbage collection.

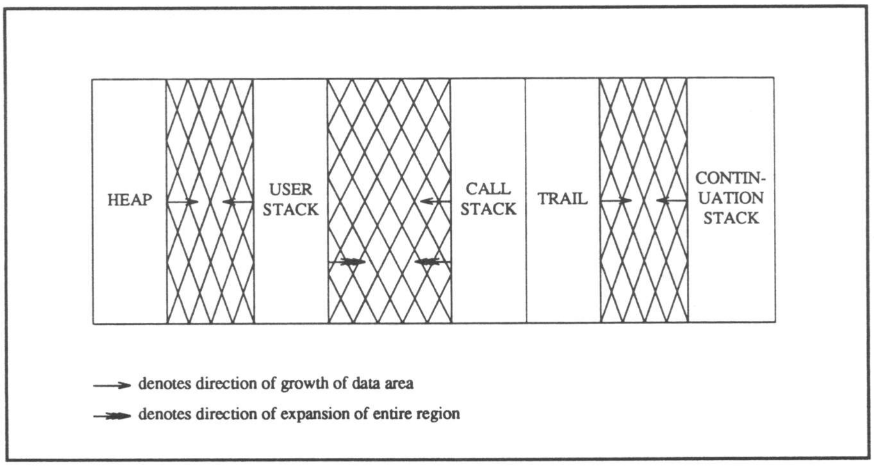
HEAP
USER STACK
CALL STACK
TRAIL
CONTIN-UATION STACK
→ denotes direction of growth of data area
→ denotes direction of expansion of entire region

Fig. 6.4. The Poplog Data Areas.

- The key field of an object can be used to hold the new address of that object during the compaction phase of garbage collection.

Some implementations of AI languages on 32 bit machines have used tags larger than two bits, to enable fast identification of frequently used data-types. This strategy was rejected for Poplog on the grounds that the reduction in available address space would be intolerable for some of the larger potential applications.

2.2 Data Areas and Control Mechanisms

There are five data areas in Poplog, which are contained in two segments of contiguous memory as shown in Fig. 6.4. The base of the heap is positioned after the shared Poplog system code and data, which has not been shown in the diagram. The segment containing the call stack need not grow towards the base of memory as shown, its orientation being determined by that of the call stack.

The "heap" contains all the dynamically allocated objects in the system. The heap is shown as a single entity but in fact it can be segmented, and this allows for the dynamic allocation of memory for "externally loaded" procedures. (For simplicity, the possibility of external segments in the heap is not shown in the diagram.)

Although all live structures in the heap will be scanned during garbage collection, only those above a certain point are eligible for compaction (or removal if dead). The ability to set this point to be the current top of the heap—called “locking” the heap—provides a method of reducing the garbage collection overhead. This is usually done after a large program has been loaded, when all its procedures and data are required to remain live.

In the system data area at the base of the heap is the “dictionary”. It performs the role of the symbol table of the conventional batch compiler, which in an interactive system must exist at run time. Unlike strings there must be a unique structure, a “word”, for each distinct source language identifier. The dictionary is a hash table which maintains these. Each word record may point to an associated identifier record which defines the run time identifier to which that word (name) is currently bound. The identifier record contains syntactic information about the identifier, and its value, if any. For program modularity, a given word can point to different identifiers in different “sections” of a program.

The “user stack” is used for stacking references to objects. In Pop-11 it is directly accessible from the language, and can be used to build elegant programming solutions. It is used by all languages for passing arguments and results between procedures and for holding temporary values during expression evaluation.

An implementation dependent aspect of maintaining the heap and user stack is that of overflow detection of both entities and underflow detection of the user stack. In operating systems that allow arbitrary memory pages to be made inaccessible, such as VMS and SunOS 4.0, the solution is to create an inaccessible memory page between the two areas and at the base of the user stack. Thus these conditions are detected automatically, and the relevant action is invoked on receipt of the memory violation signal.

Unfortunately this cannot be done with most Unix systems, but aligning the base of the user stack at the end of the data segment makes automatic detection of stack underflow possible. Explicit checks must still be used for overflow detection however, but the overhead of these is not large.

The “call stack” is the conventional call stack of the processor. To allow compatibility with either upward or downward direction of growth of the call stack, Poplog allows the whole segment to be implemented in either direction. However, Poplog enforces its own stack protocol for several reasons:

- Poplog procedures support not only lexically scoped but also dynamic binding of local variables, and the host protocol will not normally cater for this;

- arguments and results are passed on the user stack thus mechanisms for passing these on the call stack are not required;

- and most importantly, various parts of Poplog (e.g. garbage collector, abnormal procedure exit mechanisms) work with stack frames as explicit data structures, and therefore a standard machine independent format is required for these.

Besides the normal call stack discipline supported by most languages Poplog provides a variety of additional mechanisms for manipulating the call stack, without which it would have been impossible to implement Common Lisp or Pop-11.

A commonly used mechanism permits abnormal exit from a procedure, for example if there has been an error interrupt, or if it is required to abort the current procedure and replace it by a call of another, a facility provided by "chain" in Pop-11. User programs can specify that the call stack should be "unwound" up to a specified point and then a new procedure invoked at that point, and so on. A variety of types of condition-handlers can use such mechanisms, including the "catch" and "throw" of Lisp.

It is possible for a procedure to trap such abnormal exits and take action to ensure that necessary tidying up is done, using "dynamic local expressions" as described below. A special case of this is the provision of dynamic local variables in Lisp and Pop-11, which are essentially globally accessible variables whose values are always restored on exit from procedures that declare them as local, a technique that is useful for temporarily altering the interrupt behaviour, or the standard printing channel, or a counter that records the current depth of procedure calls, etc.

In addition, Poplog, like a number of other AI systems, supports a "lightweight" process mechanism. That is to say, it is possible to create a process in which a procedure is run, which may call other procedures and then be suspended. The process record will include information about the state of the call stack and local variables at the time of suspension. Later the process can be resumed, then suspended again, resumed again, and so on. This allows co-routining, and in conjunction with timed interrupts permits a scheduler to control a collection of sub-processes sharing a data-environment — a feature of Poplog that has been used for teaching undergraduates operating system design techniques. It can also be used in AI systems requiring several communicating sub-processes simulating different independent agents or different parts of a single agent.

An unusual, and possibly unique, feature of the Poplog VM is the support for "dynamic local expressions" in conjunction with the above mechanisms for abnormal procedure re-entry and exit. A special case of the mechanism is required for the "unwind protect" facility of Lisp which can trap abnormal procedure exit, but a more general version is available via the "dlocal" construct of Pop-11 in conjunction with the process mechanism as well as abnormal procedure exits.

The "dlocal" construct allows the user to define entry and exit actions to be performed whenever a procedure is entered or re-entered, or exitted. For example, the contents of some datastructure can be stored on entry and restored on exit, or special trace printing can be done to help with debugging of complex control structures. In a system that allows "abnormal" procedure entry or exit these effects cannot be achieved simply by including appropriate instructions in the procedure body since exit or re-entry may be the result of actions invoked by other procedures, such as a scheduler suspending or resuming a process.

Dlocal expressions can distinguish several different kinds of entry and exit, as described above, and vary their behaviour accordingly. An extreme example would be two Pop-11 processes which run separate Prolog systems which would have to be set up on re-entry to the processes and saved on suspension, all of which could be done within a single Poplog process. In simpler cases, the syntax for dlocal, combined with the fact that Pop-11 procedures have updaters, allows very clear and economical expression. For example a procedure that requires the 15th element of a vector V always to be saved on entry and restored on exit, could simply have the following declaration, which will cause the vector access procedure -subscrv- to be run on entry, and its updater run on exit:

```
dlocal % subscrv(15,V) %;
```

The dlocal construct can be used for saving on entry and restoring on exit the value of any expression. Thus even a lexically scoped variable, e.g. declared lexically local to a file or an enclosing procedure, can be treated dynamically by a procedure using it. Most languages that distinguish dynamic and lexical variables treat them as mutually exclusive, whereas Poplog treats the lexical/non-lexical and static/dynamic distinctions as orthogonal.

Additional control facilities are provided by the "continuation stack" and "trail", both required for Prolog. The continuation stack holds "backtrack" information i.e. records describing states of execution that must be restored if a Prolog clause fails. The trail is a stack of references to Prolog variables that have been bound by unification, and thus at some later stage may need to be unbound during backtracking. The continuation stack and trail are located at the base of the call stack, and they can be allocated more space by translating the trail and the call stack in the direction of growth of the call stack.

Although the continuation stack is currently used only by Prolog, it could be used by other languages, and some Pop-11 programmers have invoked the mechanisms directly, as is done in the Pop-11 programs used to implement the Poplog Prolog compiler.

2.3 The High Level Virtual Machine (VM)

Like many other compiler intermediate virtual machines, the Poplog VM is stack based. However, it is higher level than most and thus the production of code for it by a compiler front end is straightforward. Some of the most notable features of this high level virtual machine language are:

- Arithmetic operations do not exist as distinct VM instructions. For example, an addition would be represented as a CALL of the procedure whose name is "+". This is because in Pop-11 new operators and syntactic constructs may be defined (along with their operational semantics) as well as existing ones changed, thus at the source language level no operation can be considered primitive. When producing code for this call at the VM level, the procedure associated with the name "+" will be fetched using the dictionary, and the relevant call planted or inline routine substituted. Note that the dictionary is used only at compile time, not when the procedure call is executed.

- References to addresses do not appear in the VM language. This is due to the fact that most structures are relocatable heap-based objects, and the requirements of incremental compilation. Thus at the VM level, objects are referred to either by name or directly by object references. It is the responsibility of the lower levels of the compiler, using the dictionary, to substitute identifier names with runtime object references.

In total there are 46 VM instructions. To describe them all in detail would require more space than we have, but the following is a brief description of some of the instructions that will hopefully give the flavour of the language.

1. Stack operations:

PUSH <word>

Push the value of the identifier associated with the word <word> onto the user stack.

PUSHS
: Duplicate the item on the top of the user stack.

POP <word>
: Pop the item from the top of stack into the value of the identifier associated with <word>.

2. Procedure calls:

CALL <word>
: Call the procedure that is the value associated with <word>.

UCALL <word>
: Call the updater of the procedure. All procedures may have an updater, e.g. "hd" called normally returns the head of a list, and when called in update mode it updates the head of a list (cf "setf" in Lisp).

3. Conditional and Boolean instructions:

IFSO <lab>
: Jump to the label <lab> if the top item on the stack is not the Poplog item "false". Remove the item from the stack in any case.

AND <lab>
: Jump to the label <lab> if the top item on the stack is the Poplog item "false", otherwise remove the item from the stack and continue. Leaving the item on the stack allows boolean expressions to return values.

4. Directives:

LVARS <word> <idprops>
: Define <word> to be a lexically scoped local variable, with given property <idprops>.

PROCEDURE <props> <nargs>
: Start code generation for a new procedure whose printname is <props> and with <nargs> arguments.

ENDPROCEDURE
: Terminate compilation of a procedure expression and create a procedure record which is then pushed on the user stack.

EXECUTE
: Execute any instructions currently planted at execute level, i.e. commands entered to the top level which are not procedure definitions.

5. Miscellaneous:

LABEL <lab>
: Define the label of the next instruction planted to be <lab>.

FIELDVAL <num> <spec>
: Access field number <num> of object whose structure is defined by <spec>.

2.4 Compilation Using the Two Virtual Machines

In this section we show the relationship between the two virtual machines, the PVM and PIM, by examining what happens when a procedure is compiled using the interactive incremental compiler. There are actually two versions of the PIM known as I-codes ("I" can be taken as standing for "intermediate", "incremental" or "interactive") which are the intermediate form for user (i.e. normal) procedures and M-codes (think of "M" as referring to "machine"). The latter is only used when compiling the sysPOP system sources and will be discussed further in section 3.1. The rest of this section is concerned only with the incremental compiler and I-codes.

The result of procedure compilation is a procedure object containing executable code which will reside in the heap. These procedures can then either be invoked directly by user commands, or by other procedures compiled before or after them. Commands entered at the top level are also formed into heap-based procedures. However these will be executed immediately after being formed, and as they are not referred to by any other object the space used will be reclaimed when necessary by the garbage collector.

The production of a procedure record involves three stages of compilation as was shown in Fig. 6.1. The function of the language-specific compiler front ends is to plant VM code, which they do by calling VM code procedures: there is one procedure for each VM instruction. The method for achieving this varies between the languages, and this is mainly determined by the characteristics of the language. At one extreme there is the ML front end which creates a parse tree for each function definition and then plants VM instructions whilst walking the parse tree. At the other extreme there is Pop-11 whose compiler plants VM instructions as it reads in Pop-11 expressions, and whose only state information is contained in the local variables of the currently active compiler procedures.

The scheme for compiling Pop-11 to VM code could be very restrictive on the syntax permitted, and so a one-deep VM instruction buffer is maintained which allows for reinterpretation of the source code based on context. Thus when compiling the source code for:

```
a * b(1) -> c;
    ;;; multiply "a" by "b" applied to 1 and
    ;;; assign result to "c"
```

at the point that the opening parenthesis is found, the VM code emitted is:

```
PUSH "a"
PUSH "b"
```

However, the presence of the opening parenthesis indicates a procedure call, so the second instruction (still in the buffer) is retracted, and the code planted after the closing parenthesis is found is:

```
PUSH "a"
PUSHQ 1
CALL "b"
```

To allow lexically scoped procedures, a stack of procedures in compilation is maintained. When an enclosed procedure definition ends, the state of compilation of the enclosing procedure is restored and compilation of the outer procedure can continue.

As each VM procedure is called, the corresponding VM instruction is appended to the list of instructions being compiled for the current source procedure. The VM code produced by the front ends will obviously be inefficient, and so the VM code must be optimised, but rather than having this as a separate pass which would slow down the compiler, it is performed as the code is planted. For example, when a user stack POP instruction is planted using the POP procedure, the compiler will check for a preceding PUSH, and if found the PUSH will be overwritten and the two instructions replaced with a MOVE instruction.

There is no MOVE instruction at the PVM code level however, but such instructions are available in the PIM. The incremental compiler translates programs to the PIM via instructions referred to as I-codes. There are about 57 I-codes, some of which play an intermediate role in the sense that they are transformed into lower level I-codes during compilation. Some I-codes represent specialisations of the VM instructions which are differentiated according to their actual arguments, e.g. I_CALLPQ for a CALL to a constant procedure (and thus the type of the called object need not be checked). Other I-codes exist for non-checking structure access and arithmetic, procedure prologue and epilogue code, multiway branches, etc.

The type of optimisations performed when the PVM instructions are called are:

- Removing redundant stack usage, e.g. replacing a "push" followed by a "pop" with a "move".

- Replacing certain procedures with inline code, e.g. non-checking integer arithmetic and non-checking object field access.

- Optimising conditional and boolean expressions to use inline comparisons.

- Collapsing branch chains, i.e. if a branch target is another branch instruction, then the target of the first is the target of the second.

From version 14 a number of compile time flags will be made available to users allowing further optimisations to be made at the cost of safety, for instance turning off procedure entry checks, or backward jump checks.

Upon reaching the end of a procedure definition the result is a list of I-codes together with some extra information such as local and global variables used. At this point it may not be possible to produce code for the procedure, e.g. if it contains a jump to a lexically enclosing procedure, in which case code generation must be delayed until the enclosing procedure has been compiled.

When all such references have been satisfied, the I-code list plus declarative information are passed on to the routine which is responsible for planting machine code in a procedure record. This routine makes two or more passes through the I-code list, and for each I-code calls a corresponding function that will emit binary code for that instruction. The first pass allows labels to be translated into an offset from the start of the procedure, and at this point source language variables are translated to their runtime representations. During the second pass the machine instructions are planted in the procedure record. If there is a choice of instruction branch sizes or displacement address operand sizes, then an extra pass will be required if assumptions about the required size are proved incorrect in the first pass.

The dependence of displacement address operand size on the procedure object size is due to the presence of a literal table in each procedure object. This table holds pointers to all the heap-based objects that the procedure references. During garbage collection, the collector can mark from this table, and after compaction the contents are updated to reflect the new state of the heap. Thus all heap accesses performed by the procedure must go indirectly via this table. If heap accesses were done directly then the garbage collector would have to locate and change all embedded pointers in procedures, which would considerably slow down garbage collection.

3. Porting Poplog

Before explaining the work involved in porting Poplog to a new architecture it is necessary to describe how the system code implementing the base Poplog system is compiled using POPC, the compiler for sysPOP system sources. This has to be treated differently from the interactive compiler (a) because, for efficiency, some additional facilities are provided for the dialect of Pop-11 used to implement Poplog and (b) because instead of compiling executable procedure records, the system compiler has to produce files of assembly language instructions that can be used for rebuilding the system. This means that some problems can be left to the assembler and linker provided with the host machine whereas they have to be resolved immediately by the incremental compiler.

Moreover, a requirement for interactive development and testing of programs is that compilation be very fast, whereas re-building a Poplog system can be a slower process, allowing scope for additional levels of optimisation. Other differences are explained below.

3.1 Compilation of System Code

The code production route using I-codes as described in section 2.4 is taken by all user code, including all explicit and autoloadable libraries, most of the Prolog and Lisp compilers and all of the ML compiler. However, when Poplog is re-built or ported to a new machine, the core of the Poplog system is compiled by a different route to form assembler sources which will form mainly static objects at the base of the heap.

There are several differences from compilation of normal user programs:

- As previously explained, an extended language sysPOP is provided for system sources, with its own compiler POPC. POPC is an augmented version of Pop-11, built using the compiler tools available to the user.

- Instead of generating executable procedure records containing machine instructions, POPC outputs assembler files for the target machine (which may be the same as the machine on which it is running or a different machine).
- A different intermediate language, M-code, is used for the low level PIM.

- A different compilation strategy is used to generate M-codes, permitting considerably more optimisation than for user procedures.

- A small subset of the system is written in hand-coded assembler.

- Several of the requirements for system compilation are different from those for normal compilation of user procedures. For example, greater efficiency is required, and in many cases possible, and system procedures and structures cannot be re-located by the garbage collector, so more efficient accessing methods are possible.

The remainder of this section elaborates on these points.

The majority of the system sources are written in sysPOP. This is a dialect of Pop-11 that has been extended to allow the representation of and the operation on: machine integers, addresses, C-like structures, call stack frames, etc. Other additions include only allowing garbage collection and checks for interrupts at specified points, access to operating system calls and calls to assembly language routines.

The compiler for sysPOP, POPC, uses the normal Pop-11 front end (extended to accept the sysPOP language) to produce Poplog VM code. However, rather than this being rewritten down into I-code, the resulting VM-code list is passed almost unadulterated to the optimisation and code generation routines in POPC which generate M-code PIM instructions as follows.

The VM-code list is optimised in a more thorough manner than that for the incremental compiler, and tables of translations between operations (such as machine integer addition) and low level M-code instructions that implement those operations are used to create a list of equivalent M-code instructions.

M-code is similar to the ISP-like "register transfer expressions" that many retargetable compilers use to represent their low-level machine-independent code (Davidson & Fraser [1980]). Some examples are:

M-code	Equivalent register transfer
M_ADD 3 r1 r2	r[2] = r[1] + 3
M_ADD CMP a b EQ lab	if M[a] == M[b] then PC <- lab

There are 47 M-code instructions. A few are more complex than register transfers, e.g. those to create and unwind stack frames on procedure entry and exit respectively and one implementing multi-way branch tables.

The complete M-code list corresponding to a procedure definition is passed to a procedure which steps through the list, and for each M-instruction calls the relevant routine which will emit assembler to implement that instruction. Additional routines also exist to create assembler representations of Poplog objects such as lists and properties.

A small fraction of the system is implemented as subroutines written in hand-coded assembler. These perform operations which are either too primitive or too machine or operating system dependent to be written in sysPOP, or which are so time critical that hand-coding gives a significant overall speed increase.

The first implementation of Poplog was based on the VAX, whose architecture makes it relatively easy for a human programmer to produce good code, and there were quite a large number of hand-coded assembler routines. With a move towards ever simpler architectures, which are harder to program efficiently at the machine level, the sophistication of the sysPOP compiler has been increased, and so more and more of the assembler routines have been replaced by routines coded in sysPOP which are then automatically optimised. However, there remain some which must be written in assembler, for example routines which manipulate the Poplog call stack and therefore cannot be Poplog procedures that would themselves use the call stack.

The hand-coded assembler files are also processed by POPC, and this allows them to use definitions from sysPOP "included" files providing standard definitions and compile time constants, analogous to the use of the "#include" facility in C.

As each source file is compiled, all the names used in the source file, along with their attributes, are recorded in a separate file. When all the source files have been compiled the information about names is processed by a utility called Poplink which is responsible for checking consistency across source files and creating an assembler representation of the Poplog dictionary, and associated word and identifier records. This is necessary because after the new system has been created, access to system identifiers is required at run time, e.g. when user procedures are compiled and refer to system procedures or system global variables. A subset of the names, not required after the system is built, will be marked as not for export, and will not go into this dictionary.

All the assembler files produced by POPC and Poplink are then assembled and linked in the normal manner.

3.2 Work Required to Port Poplog

The architectures to which Poplog has been ported are: DEC VAX, Motorola MC68000 family, GEC Series 63, Intergraph Clipper, Intel 80386 and Sun SPARC. At the time of writing a port to the MIPS processor is nearing completion. Architectures lacking large uniform address spaces (e.g. Intel 80286) are not suitable hosts for Poplog.

The operating systems under which Poplog runs are VMS and several different flavours of UNIX, including Berkeley 4.2, 4.3, SunOS, Dynix, and HP-UX. The four major elements requiring change when Poplog is ported to a new architecture are:

1. System code generator (for POPC, the batch compiler). The routines which produce assembler for M-code instructions must be rewritten. Other changes may be necessary such as producing a routine to emit the binary representation of floating point numbers if a non-standard format is used. In some cases porting involves producing output for a new assembler even though the processor is the same, since different suppliers don't all use the same assembly language format.

2. User code generator (for the run time assembler of the incremental compilers). The routines which produce target binary for I-code instructions must be rewritten for each new architecture. Closures and arrays are held as procedures, and thus routines to plant machine code for them must be rewritten. Arrays could be held as conventional data structures but are held as procedures to make access efficient and also because it is often convenient and elegant to treat arrays as functions.

3. Hand-coded assembler. Roughly 1500 lines of assembler must be rewritten to implement routines which either cannot be implemented in sysPOP or are too time-critical for sysPOP.

4. Operating system interface. Up to 25 sysPOP files may need modifying when porting to a new operating system. The amount of change required depends mainly on the amount of difference between the new operating system and one on which Poplog already runs. (The number of files is larger than might be expected because the system has been deliberately fragmented so that unwanted facilities can be omitted if not required in a "delivery" system.)

The time required to perform the first three tasks can be anywhere between 3 and 8 man months depending on the target architecture and the experience of the implementor. It is not so easy to estimate the time required by the fourth task, but writing for another flavour of Unix would probably take only be a day or so. The initial port to Sequent Symmetry, the first Poplog host with an Intel processor, took just over four months, done by someone who had previously worked on Poplog but had never ported it. After that porting to the Sun386i took about two weeks.

This may seem a long time to perform a port when compared with other systems of comparable complexity that are, say, implemented in C, and can simply be re-compiled, but it must be remembered that porting Poplog includes the production of two back end code generators (tasks 1 and 2). The porting time of a system written in C would not include the time taken to produce the C compiler.

This obviously raises the question why Poplog is not implemented in C. The reasons that this is not so are:

- Various elements of Poplog require direct access to the call stack, which is not possible in C.

- The presence of a garbage collector demands that registers be partitioned into those that can contain pointers to Poplog objects (from which the garbage collector will mark) and those that do not. Such tight control over register usage is not possible in C.

- Poplog requires a global register to point at the top of the user stack. C does not allow global register variables.

- There are benefits in having the system implementation language very similar to Pop-11 such as ease of migrating programs between user and system code, and implementors having one less language to learn.

- The POPC compiler performs optimisations suited to Poplog that could not be expected of a C compiler.

The ease of porting Poplog to an architecture and the quality of the code produced depend on the presence or absence of features in that architecture. Some machine features and their impacts on portability and efficiency are listed below.

- All current Poplog hosts have byte addressing. Whilst a machine without this can support Poplog (indeed, in sysPOP, pointer and machine arithmetic use different operators even though the actions are equivalent for existing machines) the resulting system may be less efficient, e.g. because accessing individual bytes will be more complex.

- Early Poplog ports were all to machines with about 16 registers. Having too small a number of registers (e.g. Intel 80386) can preclude the advantage of register based variables, whereas having a large number of registers (e.g. AMD Am29000) would cause problems in trying to use all the registers effectively.

• A significant proportion of the system's time is spent performing user stack operations and tag manipulation, thus the availability of facilities such as autoincrement/decrement addressing mode (or stack operations using a general purpose register as the stack pointer) and quick forms of arithmetic and logical operations are important.

• Both of Poplog's low level virtual machines have three-operand instructions. Implementations on a two operand machine, which necessarily destroy one source operand, must introduce extra code to save values. An orthogonal instruction set can reduce this problem, and generally makes all code production easier.

• Heap-based procedures must be position independent. Machines without PC-relative branches and data access modes therefore require a procedure base register to be maintained at extra cost. Alternatively garbage collection procedures have to be far more complicated as procedure records will need to be altered when they are re-located.

• Pipelined machines often have delayed branches (the instruction following the control flow change is always executed before the branch target instruction is executed) or delays between loading a register from memory and the point at which the register contents are valid. To produce good code for these machines requires careful scheduling of instructions.

4. THE COMPILER TOOLKIT

A collection of built-in procedures give users tools for building new incremental compilers, or extending the syntax of the existing languages. These tools are also used for the development of Poplog itself including the implementation of the four Poplog languages.

The lowest level of access, and the one that all compilers will use, are the Poplog VM code planting procedures that were described in section 2.4.

Languages created by extending Pop-11 use two main mechanisms: macros and syntax procedures. Macros are words that when encountered by the itemiser will read in source program text from the input stream and then replace it with modified Pop-11text that is subsequently compiled in the normal way. Syntax procedures are what drive most of the compilation of a Pop-11 program. For example, the "if...endif" construct is compiled to VM code by a syntax procedure whose name is "if". Whenever a word is encountered in the sourcecode stream whose value is a syntax procedure, that procedure is invoked to handle the compilation of the construct. The provision of user-definable syntax procedures, unique to Pop-11, allows a wider range of language extensions than macros, since macros must produce code that is legal according to the rules of the language, whereas syntax procedures need only plant legitimate sequences of VM instructions.

Several languages have been implemented in this fashion including sysPOP, the extended dialect of Pop-11 accepted by POPC, the compiler for the system source code of Poplog. To help in defining new syntax procedures, the user is given access to a collection of procedures which recognise and produce VM code for certain syntactic units of Pop-11 such as expressions, statements, sequences of statements, etc.

Of more general use is the itemiser which has procedures for returning the next item from the compilation stream, checking that an item is present in the stream and replacing an item on the stream by invoking a macro. The itemiser itself is modifiable through the use of character class tables. Thus, for example, one could define "$" to be of class alphabetic, which would then allow "foo$baz" to be recognised as a word. One of the character types, which the Pop-11 compiler assigns to the underscore character, allows alphanumeric characters to be combined with characters of other types, e.g. "class_=".

To implement a new language it is not necessary and sometimes not possible to use the built in mechanisms for analysing the input stream. However, Poplog allows a more conventional parser to be implemented which then plants VM code whilst traversing the parse tree as is done in the case of ML. One of the existing languages could be used to write this, and Pop-11 has proved very suitable for this purpose.

5. THE POPLOG ENVIRONMENT

Poplog provides a host of facilities for teaching and research, and for developing and testing versatile professional software. Examples of such facilities include the following.

- Macro definitions allow conditional compilation, "include" files and textual substitution. As well as aiding software development, command languages can be implemented using this facility.

• A library mechanism allows both explicitly loaded and autoloadable libraries. The latter are automatically loaded when the compiler encounters a name that is not currently defined. This allows systems to be kept small (unnecessary procedures are not loaded) but does not force the user to remember to load a long list of libraries that he requires, as this mechanism is transparent. The list of directories to search for both types of libraries can be modified if required, facilitating user-specific or group-specific extensions to the system.

• Interfaces to operating system facilities, dynamic store management, garbage collection, and a rich variety of data-types including indefinite precision integers, ratios, complex numbers, bit vectors, byte vectors, hashed property arrays, and so on, are all automatically provided, and may be accessed by a new compiler via procedure calls to Pop-11 system procedures.

• There are two garbage collection mechanisms. When sufficient memory and/or swap space are available a "stop and copy" garbage collector can be most efficient. However, if there is not enough space available the non-copying garbage collector is invoked automatically. Users can specify that only the copying garbage collector is to be used, and can control garbage collection by directly invoking it at appropriate times, and by locking the heap to reduce the amount of copying required.

• Poplog contains an integrated editor, VED, which can do much more than just edit files. Programs can be compiled from the edit buffer comprising the whole file, a marked range or the procedure containing the current cursor position. On error the cursor will be positioned at the point of error. Programs can also be executed in an edit buffer allowing the examination of output and the reentering of commands.

• A VED buffer is a datastructure that is readable and writeable by both users and programs, with all changes immediately visible on the screen. This enables the editor to be used as a general purpose, terminal independent, user-interface mechanism. This is not so easy when the editor is a separate process communicating with the AI language system via a pipe. Among other things, the editor has been used to provide an electronic mail interface, an electronic news-reader and a simple character-based graphical program e.g. for displaying parse trees etc.

• On-line help and tutorial files exist for the current system and can be inspected using VED, as can program library files. With the aid of search lists, these can readily be augmented by the individual user or by a teacher for a group of students. Because the editor is integrated with the system, it is easy to include executable examples in the teaching files, including examples that users can modify and experiment with.

• Large programs, once debugged, do not have to be recompiled each time Poplog is restarted, but rather "saved images" can be made. Moreover saved images can be "layered" so that a single saved image can be shared by a group of users, and several different saved images created relative to it, containing code compiled after it is run. This is typically how different Poplog languages are implemented as shareable saved images.

• The efficiency of Poplog as compared with some interpreted AI language systems allows it to be used for a whole range of system development, in addition to AI applications. For example a user developed a fast troff previewer written in Pop-11, enabling formatting to be checked without wasteful printing. For procedures requiring even greater efficiency, e.g. for AI vision research, an interface to C or Fortran is available. A recent change to make Poplog pointers address the first data item after the key will simplify this interface (see Fig. 7.3).

• A "lightweight" process mechanism is provided, which can be used in conjunction with timed interrupts to simulate multiple processes. In conjunction with the section mechanism it allows two Prolog processes to be run in tandem, sharing datastructures.

• The Poplog "Flavours" library implements a powerful object oriented programming system, including demons and multiple inheritance.

• There is a Poplog window manager for certain workstations. This is being replaced by an interface to the X Window System, in order to reduce machine dependence.

• For expert system development, toolkits implemented in Poplog are available.

• In future the sysPOP dialect and POPC compiler for it will be made available to users. This "delivery" mechanism will enable them to build reduced versions of Poplog containing only what is required for their applications. It will also be possible to compile user programs to object modules which can be linked as required in combination with other programs. In some cases it will permit cross-compilation.

All the main features are automatically made available for any new language added to Poplog. Moreover, when Poplog is ported to a new machine, all the development facilities are immediately available, and because Poplog is so portable users are not locked into specialised machines but can take advantage of the increased speed and low cost provided by new general purpose computers, for instance RISC-based workstations.

6. CONCLUSION

Although all the mechanisms described above work on a range of machines and operating systems, there is a price that is paid for the flexibility and generality of Poplog as compared with a single-language system. A compiler and store manager dealing with only one language can often make optimising assumptions that are not valid for a mixed language system. For example in a pure Prolog system programs can be optimised to run faster than Poplog Prolog which has to allow for mixed language interactions.

However, this is counterbalanced by the fact that Poplog Prolog users have the option to identify key portions of their programs that could be re-written in Pop-11, often thereby achieving far greater speed increases than using a stand alone Prolog system. Similarly, programs written in a stand alone Lisp system will often run faster than programs implemented using a more general virtual machine. However, most of the commercially available Common Lisp systems require considerably more memory than Poplog Common Lisp, which initially requires a process size of under 2 Mbytes—including the editor and the Pop-11 system as well as Common Lisp. For those using only Pop-11 and the editor the initial size is well under a megabyte on most machines. One reason for the compactness is that many facilities are provided in autoloadable libraries instead of the main system, which means that users are not encumbered with facilities they are not using.

Pop-11 programs in Poplog seem to run at speeds comparable to programs written using the best Common Lisp compilers. Poplog Common Lisp is not yet (Poplog V13.6) as fast, but further optimisations should remedy this, and in any case the small size can make programs much faster simply by reducing the need for paging.

The fact that the Poplog VM uses a stack for passing arguments and results, allows procedures to take variable numbers of arguments and return variable numbers of results. But this means that it is not generally possible to use optimising techniques that transfer arguments and results via registers. However, the fact that modern processors have increasingly large caches implies that if the top elements of the stack are accessed frequently then that portion of memory will be cache resident. This should considerably reduce the overhead resulting from use of a stack, without any complicated and time-consuming compiler optimisations being required.

The success of Poplog's two-level architecture is attested by the following facts:

- Its use for AI teaching, research and development is continuing to grow, in the UK and elsewhere. For instance, in one UK university it is used in six different departments and in another it is used in four different schools. At the University of New South Wales, in Australia it is the basis of a new MSc degree in cognitive science and is used for teaching undergraduates at Griffiths University.

- It is being used in both commercial and academic organisations.

- Its use for AI purposes ranges over such things as speech and image analysis, natural language processing, theorem proving and expert systems, including real-time expert systems. Commercially available tools have been implemented using it.

- Its use is not restricted to AI — e.g. it is being used to develop a variety of software tools such as text-editors, mail front ends, previewers, compilers for new languages, and the ML system is being used for teaching and research in computer science and software engineering.

- A commercial organisation (PVL) is marketing a program validation system based on Poplog.
- It has proved comparatively easy to add incremental compilers for new languages, e.g. the ML compiler, without having to produce a new development environment for each one including editor, help system, etc.

- It has proved relatively easy to port to new architectures. E.g. the initial SPARC (Sun4) port took under three man-months.

The development and maintenance of the whole system, including all four compilers the editor and the window manager, with implementations on a range of machines and operating systems, has required far less effort than would be required for four separate language systems. The team at Sussex University responsible for Poplog has varied between four and eight full time programmers, with part time help from a few others.

For some users a Lisp machine, with its high performance processor and more sophisticated Lisp development environment might be preferable to Poplog. However, for those with more limited resources and more varied needs, a portable multi language system that runs on generally available hardware can be more attractive. For some University departments and some commercial users, Poplog has played an essential role in making it possible to get on with teaching, research and development in AI. In particular, for experienced programmers in languages like Pascal and Fortran, the Pop-11 sub-language of Poplog has proved a convenient bridge from more conventional languages to AI languages.

7. ACKNOWLEDGEMENTS

This paper was written (in 1989) by the first two authors, but the main architect of the two-level virtual machine architecture was the third author. Steve Hardy contributed to the initial design of a common virtual machine. Staff and students at Sussex University designed and implemented other aspects of Poplog, and Integral Solutions Ltd ported it to some of the host machines. The Pop-11 language and some of the core features of the Poplog system owe an intellectual debt to the developers of Pop2, Pop-10 and WPOP in Edinburgh, including Robin Popplestone, Rod Burstall, Julian Davies, Ray Dunn and Robert Rae.

The work was supported in part by grants from the UK Science and Engineering Research Council and the Alvey Directorate.

8. REFERENCES

J. W. Davidson and C. W. Fraser: The Design and Application of Retargetable Peephole Optimiser" in *ACM Trans. Prog. Lang. & Sys. vol. 2*, no. 2, April 1980, pp. 191-202.

Aaron Sloman and the Poplog development team: "Poplog: a portable interactive software development environment", Cognitive Science Research Paper No 100, School of Cognitive and Computing Sciences, University of Sussex, 1988.

T. B. Steel, Jr.: UNCOL: the Myth and the Fact" in *Ann. Rev. Auto. Prog.* Goodman, R. (ed.), vol. 2, 1960, pp 325-344.

A. S. Tanenbaum, H. van Staveren, E. G. Keizer and J. W. Stevenson: "Practical Toolkit for making Portable Compilers" in *Communications of the ACM vol. 26*, no 9, September 1983, pp. 654-662.