Formal Aspects of Computing (2002) 13: 196–213
© 2002 BCS

Formal Aspects of Computing

POP, A Broad-Spectrum Programming Language, 1967–2002

Robin Popplestone [1]

Department of Computer Science, University of Massachusetts, Amherst, MA, USA

Abstract. This paper discusses the POP-2 language and Multipop time-sharing system developed during the second half of the 1960s. POP-2's expressiveness spanned numeric and symbolic programming and supported experiments in logic of programming languages (by Boyer and Moore) and programming language concepts (for example, Michie's memoisation). The paper also discusses how the goals of POP-2 might be achieved in a successor language.

Keywords: Functional; Imperative; Programming language; Proof; Symbolic

1. Introduction

When Rod Burstall and I designed POP-2[2] in the late 1960s we were attempting to produce a language that would express both symbolic computations (as supported by LISP as it then was [McC60, MAE62]) and also numeric computations (as supported by Fortran and Algol). Among the applications we envisaged for the language was robotics: in effect we intended to use it to implement embedded systems. In general our intent may be summarised as to create a *broad-spectrum language*.

In our specification of the language we were strongly influenced by the ideas on the use or adaptation of the λ-calculus advocated by Peter Landin, which had already had a strong effect on the design of CPL [Str67, Lan64, Lan65, Lan66]. We were also concerned to provide an interactive language in which compile time and run-time were flexibly interleaved. Finally, we tried to provide a syntax which we felt to be closer to a transcription of normal mathematical notation than was LISP – a syntax which despite its idiosyncrasies was closer to that of Algol 60 and CPL. However, we tried to avoid locking up capabilities in syntactic constructs which could be provided with greater generality via function calls. In particular, as in LISP, storage allocation was achieved with function calls rather than through special syntax as in Algol and many of its descendants. In general it may be said that we tried to avoid locking up generally useful capabilities in the

Correspondence and offprint requests to: Robin Popplestone, 6 Doune Quadrant 1/2, Glasgow G20 6DL, Scotland, UK

[1] This work was supported in part by the NSF (CDA-9703217), by AFRL/IFTD (F30602-97-2-0032), and by DARPA/ITO SDR/MARS. This paper was written while the author was visiting Edinburgh University.

[2] The name 'POP' was bestowed by Donald Michie upon the first of the series, POP-1, which the author had always referred to as 'Cowsel', for Controlled Working Space Language.

system in a form which made them accessible to users in only a limited way, except in circumstances in which system integrity would be compromised.

Features of the language and its implementation were:

1. *Support of the imperative paradigm through doublets.* Rather than try to create a pure functional language, which, with the technology of the time would have been be unable to support the range of applications we had in mind, we sought to provide an encapsulation of mutable memory through *doublets*, which consisted of selector–updater pairs. Potentially, all POP-2 functions were doublets, though many (such as `sin`) would raise an error if an attempt was made to use them in the update sense. Thus, for example, a matrix A might be represented by a doublet `A`. Used in a normal expression `A(i,j)` would evaluate to the representation of the i, jth entry in the matrix, but used as a target of an assignment, the i, jth element would be updated. This was quite a flexible approach – for example, various mappings of the matrix to memory could be handled by the same code. Given the late binding of the mapping to memory, this flexibility was bought at the price of performance.
   Here is an example drawn from the 'Silver Book' [BCP71]:

   ```
   VARS DICT; ASSNIL -> DICT;
   "CHAT" -> ASSOC("CAT",DICT);
   ASSOC("CAT",DICT) =>
   ```

   The first line declares a variable `DICT` and gives it a null-association as value (note the left to right assignment). The second line makes an entry in the dictionary using ASSOC in the update sense. The third line uses ASSOC in the normal sense, printing out the answer `"CHAT"` using the postfix `=>` operator.
2. *Users could create classes of opaque records and vectors.* For records, this was achieved by a call of the function `recordfns` which took as its argument a descriptor of the fields of the record class, and returned as a result constructor and destructor functions together with a suite of selector–updater doublets.[3]
   A user could create a class of *strips* (in effect 1-indexed, 1-dimensional arrays) by using `stripfns` to provide an 'initialiser' function which would call the storage controller to give a strip of the chosen size, together with a 'subscriptor' doublet to access the components of the strip. Strips too were opaque, being only accessible through the selector doublet.
3. *One of the first garbage collectors that allocated arbitrary sized blocks of memory, providing storage compaction to prevent fragmentation.*[4] Providing fully automatic storage control was (and remains) an important tool in supporting good software engineering for those advanced systems which by their nature cannot have memory allocated either during system initialisation or in synchrony with function activation. In particular, it was essential for supporting the integrity of any record-class facility in which the life-time of a record was not co-extensive with that of the Procedure Activation Record of the function that created it.[5]
   This garbage collector relied on what were in effect two garbage-collect methods associated with each object[6] class: a *mark method* used to follow the pointers existing in an object, and a *relocate method* used to update pointers during the storage compaction phase.[7]
   These methods were located in a descriptor block shared by all records of the class, called a *key cell*.[8]
   Among the entities subject to the control of the garbage collector were the machine-code blocks into which the language was compiled. This capability supported the interactive environment, since users could redefine a function or functions knowing that the old code would be garbage collected.

[3] The extension of ALGOL to provide record classes was under discussion in the community, see [WiH66].

[4] Reynolds had one working earlier for his Gedanken system [Rey70].

[5] While allocating memory for many numerical computations can be decoupled from actually performing the computation, the same is not true for symbolic computations, for which the size of the result cannot be determined without actually doing the computation.

[6] We'll use the term 'entity' for what in the system could be a value of a variable. Rod called these 'items'. We'll use 'object' for those entities which were held in blocks of store.

[7] Since the relocation mapping (from old location to new location) required space resources, a fixed area of memory was allocated to make room for this mapping. This restricted the number of cells that could be relocated in a given compaction. However, compaction was only required to prevent storage fragmentation, since the free cells derived from the mark and sweep passes were chained up and could be reused without compaction.

[8] This approach to managing store was derived from the Absys/Abset systems [EFG71], created by Foster, Elcock et al. in Aberdeen – our original plans called for us to have a 'zoo' in which different data-types were confined in 'cages' of memory, so that the system knew the type of an 'animal', (i.e. a data-structure) by the cage in which it was to be found.

4. *The creation of higher-order functions was facilitated by a technique later formalised as* **lambda lifting**. Essentially, where it was necessary correctly to handle a variable that was non-local, but non-global, rather than providing a special closure mechanism, users were expected to λ-bind the variable and 'partially apply' the resulting function.[9] Requiring special syntax, this was not exactly an elegant solution to the provision of higher-order functions, but it did work. This mechanism did require the garbage collector, since partial application depended on generating a short machine-code block which pushed the partially applied arguments on the stack before jumping to the code for the function. These blocks had to be allocated in the garbage-collected area, so that a given function could be partially applied arbitrarily many times.

   Here is a definition of the higher-order function `twice` taken from [BCP71].[10]

   ```
   function twice f;
   lambda x f; f(f(x)) end(%f%)
   end;
   ```

5. While I (at least) did not understand the issue of lazy evaluation, still less know how to implement it in general, we hoisted aboard Landin's preaching about streams at least to the extent of providing what we called *dynamic lists* – any lists in POP-2 could in effect be lazily extended in the `tl` (or `CDR`) direction. This was handy for writing the compiler: the compiler proper operated on a token-list which it could parse functionally.
6. POP-2 was intended to support the activity of proof. A number of resolution-based theorem provers were written in the language, but the most enduring proof system was that written by Boyer and Moore [BoM75] for proving the correctness of LISP programs.[11]
7. We provided a *macro* capability. Macros in POP-2 were ordinary, user-defined, named functions with the sole distinction that the identifiers were tagged to cause the compiler to call them as soon they occurred on the input token stream. Having access to this stream they were able to scan it and modify it, thereby providing the user with an ability in effect to modify the language syntax. The wisdom of providing this facility might be questioned, not least because of possible interactions of macros written by different authors.

   Another POP-2 capability that has passed into Prolog and SML was to allow users to define their own infix operators with specified precedence. One could pass an infix operator as a parameter, using the key-word `NONOP` (cf. `OP` in SML).
8. To provide a mechanism to structure name-space beyond that structure provided by local variables we developed a module concept we called *sections*. Sections were far from being an unqualified success, being somewhat inconvenient to use. The syntax for referring to a variable v within a section s was the rather clumsy s`$-`v. We didn't have the option of using the '.' notation in POP-2, since that was a post-fixed synonym for function application, for example `l.hd`. Moreover sections were never well integrated with the debugging environment.

   In a language that supports function closures and which is not statically typed it is not clear that there is a need to provide a separate mechanism to support modularity, since a module can be regarded as a mapping from names to functions and this can be created within the basic framework.
9. Rather late in the development process we began to worry about providing transfers of control that violated the usual discipline of function entry and exit. Rod, who, more than I, was a diligent reader of, and collaborator with, Peter Landin, wanted to implement his J operator. Eventually a specification was agreed of a `jumpout` function which provided a capability like that of modern exception handling. A call of `jumpout` created a function (in effect an exception handler) which encapsulated a *program point* to

[9] Our use of the term 'partially apply' indicated an imperfect understanding of the λ calculus: theoretically there should be no difference between what we called 'partial application' of a function and normal application. This points to a real limitation in the extent to which the constructive subset of POP-2 could be regarded as a functional language. It's worth pointing out that few, if any, versions of LISP handle application correctly. In both cases the problem arises from the desire to provide *variadic functions*, that is 'functions' which take a variable number of arguments. In LISP for example, `(+ 2)` does not mean 'the function $\lambda x. + x2$' as it would in the λ-calculus, but simply the number 2.

[10] The reference manual [BCP71] used the lower case alphabet; programming was done on teletypes which provided upper case only.

[11] When Boyer and Moore left Edinburgh to return to the USA, they translated their program into LISP – it has been greatly developed since then.

which control would return if the function were ever applied (in effect raising the exception), provided that control had not returned outside the function in which `jumpout` was called.

While `jumpout` provided an equivalent to exception raising and handling (and, as in SML, could be used as a normal, rather than exceptional, control mechanism), prompted by developments in MIT such as Hewitt's Planner [Hew69] and Sussman's Micro-Planner [SuW70] we envisaged a requirement to reactivate a state of the computation that, in the normal course of function entry–exit would no longer exist. In 1970, just in time for the 'Silver Book' [BCP71] edition of the POP-2 manual, we agreed on a specification of functions that would allow a user to encapsulate the current state of a computation in a form which would allow the state to be restored from any place in the program. In implementations of Scheme this kind of capability was to be provided by `CALL/CC`. This state-saving capability rather stressed our stack-based implementation of the language – implementations of Scheme in which Procedure Activation Records are allocated on the heap can provide `CALL/CC` with very little overhead. In POP-2, by contrast, state-saving was achieved by copying the main and auxiliary stack into a heap-allocated data structure, an expensive process. To mitigate the cost of this, at the expense of complicating the user interface, we provided a 'barrier' capability to limit the scope of the state-saving. It should be noted that state-saving applied only to the functional aspects of the language – mutations of data structures that occurred after a state was saved were not undone when it was restored.

In achieving these aims we sacrificed a number of features which are desirable:

- Firstly, while our run-time type checking was made quite discriminating by the provision of user-definable record classes, it has not proved possible to retrofit a compile-time type system to the language. A major problem arises from the 'open-stack' architecture, which makes POP-2 resemble the later Forth [MoL70] language in some respects.[12] I have attempted to address this by treating POP-11 functions as in effect parsing the stack to obtain their arguments and generating their results (so that the signature of a function is in effect a quotient grammar), but the results have been of limited utility.
- While POP-2 did produce machine code for the 4130 computer, its performance was limited by the late binding of types: all operations could only be translated into subroutine calls. This was made particularly necessary by the fact that POP-2 was the implementation language of the Multipop timesharing system, whose basic security depended on the correct policing of illegal operations on data structures. Within Multipop, peripherals were encapsulated using functions partially applied to buffers.
- The language did not use lexical binding of local variables. This was a mistake shared by early LISP implementations. I did give some thought to lexical binding, but rejected it on two grounds. Firstly it would have required at least one index register to be allocated to providing access to the current Procedure Activation Record. There was just one index register in the 4130 which doubled as a stack-pointer. I didn't see any way of using it for both purposes, given the unpredictable way we allowed users to manipulate the stack. Secondly, the variable-binding technique we used allowed users to have access to the values of their local variables when an error occurred. This was seen as an advantage for debugging.[13]
- Real-time behaviour was rendered unpredictable by the garbage collector. This restricted us to what might be called quasi-static control of robots – all the inner loops were implemented using analogue technology while the programmed outer loops ignored dynamics by incorporating long dwell times to allow the inner loops to stabilise.
- Finally, rather late in the design process Rod wanted to incorporate a defined abstract syntax, which could be quoted, into the language. I resisted this because it would have meant making a big change in the compiler architecture. In retrospect, Rod's desire was well founded and would have served the language better in the long term, but, given that we were trying initially to fit the language, operating system and multi-user space into a 192 kilobyte memory, I believe that my resistance to providing this feature was appropriate since we needed to provide an implementation that people could use in the environment we had.

Some of the capabilities provided by `eval` in LISP, which depends on the user's ability to create LISP

[12] An important step in the evolution of POP-2 stemmed from Rod's construction of a bolt-on parser that mapped the token stream of POP-1 from an infix notation to the reverse-polish, open-stack, notation of the POP-1 language.

[13] We didn't consider the possibility of preserving enough compile-time information to be able to dissect all PARs for the user in the manner of modern language implementations. Even if we had, it's doubtful if we could have spared the space to carry over the information.

abstract syntax, were provided by the `popval` function in POP-2 which took as its argument a list of tokens. This turned out to be quite useful in creating interactive programs in which a user could type in mathematical expressions to be evaluated.

2. Implementations of POP-2 and its Descendants

POP-2 and its derivatives have had a number of implementations since 1968.

2.1. The Implementation on the Elliot 4100 Series

This was the original implementation. It provided a stand-alone language environment and operating system running on the Elliot 4100 series machines.

A copy of the whole Multipop system as it existed in 1974 is extant [PDR74]. The architecture of the 1974 version does not differ significantly from the 1968 version, but it was almost completely recoded between 1968 and 1974. Its rise to a reliability sufficient to sustain a research program was due mainly to the efforts of R. D. Dunn, using device drivers written by D. J. S. Pullin, with subsequent improvements being made by R. Rae. Without the excellent environment, both physical and intellectual, made available in our early years in Edinburgh by the leadership of Donald Michie none of this could have happened.

Multipop in 1974 consists of 284 pages of code which comprise almost the *total* system code loaded into the machine. A supervisory process called the *landlord* determined resource allocation (both memory and time) for the user processes. The landlord ran in a double stack[14] fixed in memory. Each user process contained a double stack and its own dictionary mapping the each user variables to the location where its values were held.

The loading process is described in [PDR74] as follows.

The system is bootstrapped into core by a NEAT[15] program called POP2. This loader then loads the rest of the system.

The NEAT program has created cells for SYSDIC, WORDKEY and SYSPROGKEY before it inputs anything. This initialisation process involves reference to FUNDEF, UNDEF, SETPOP and NIL, and these are assumed to be the first 4 systems references and so must be the first 4 variables to be set up.

The landlord stack extends from 576 to 1599, and the systems reference area from 1600 to 3512 with locations 256 to 575 being used for references which need to be outside the control of the garbage collector.

This loader then loads the rest of the system.

The system was written in POP1.5, which was originally compiled by a POP-1 program into loader code. POP1.5 allowed one to include loader code directly, to write functions that would be present as built-in functions in POP-2, and to create variables that would be present in POP-2. Assembly code could be included, otherwise a reverse-polish format was used. All functions, built-in or user defined, were translated into machine-code subroutines. An example of the reverse-polish form of POP1.5 is the default printing function `GENPR`:

```
FUNCTION GENPR;
    TWELVE CHARPR DATAWORD PR FOURTEEN CHARPR
END;
```

This acts as a default print method for user-defined data structures. It prints first character with code 12[16], that is '<',[17] followed by the `DATAWORD` associated with the record class, followed by the character '>'.

A primitive function such as `EQ` was entirely in assembly code.

```
FUNCTION EQ;
    $MVB 1; $SUBS:M 0; $JZ LO; $CLS:M 0; $JI 0;
LO:
    INCS:M 0; $JI 0;
BASICEND
```

[14] The double stacks of POP-2 consisted of an operand stack and a saved-variable stack.

[15] The manufacturer's assembly language.

[16] It was more economical to hold small integer literals in global variables.

[17] The POP-2 character set was not ASCII.

Here BASICEND indicates that the code generated for the function should not be wrapped up with the usual function entry/exit code sequences.

Both arguments are on the stack at entry. The second one is unstacked to location 1 by the MVB instruction – the value is also left in the main accumulator. This is subtracted from the second argument (on the top of the stack) by the SUBS instruction. If the result is zero then we jump to label L0, where the top of the stack is incremented (and so must have value 1, indicating TRUE, and we leave the subroutine by an indirect jump through location 0. If the result of the subtraction is non-zero, we clear the top of the stack (indicating FALSE) and return.

By 1974 most of the system had been converted to assembly code – a retrograde step from a modern perspective. It consisted of:

- A complete multi-access operating system including interrupt routines for console teletype, a multiple serial line handler, line printer, paper tape readers and punches, magnetic discs, real time clock, and a link to a Honeywell 316 minicomputer controlling the Freddy robot, and another 316 supporting the Bionics group.
- The central garbage collection and storage compaction routines.
- The definition of the system storage classes.
  - *The key class* of object-class descriptors.
  - *The pair class and null class* used to implement lists.
  - *The reference class* used to hold the value of variables. Users could also create references as updatable 'boxes' to provide an indirect handle on any entity. Because of addressing limitations in the 4100 computers, members of the reference class had to reside in the first 64K words of memory, called the reference 'cage' – a term inherited from our original store-management plan.
  - *The word class*: equivalent to *symbols* in LISP, words were used to represent identifiers.
  - *The function classes.* There were four distinct classes used to implement functions, each with its own key. Two of these classes were closure classes: one for user closures created by partial application by user functions, and one for closures created by partial application by system functions.[18] Built-in and user-defined functions belonged to the other two classes.
  - *The device and facility classes.* These served to present the peripheral devices to users.
  - *The user class and dictionary class.* A user record held the root data for a user process, including a double stack, a dictionary mapping from a user's words to the corresponding reference, user copies of mutable system global variables (such as the current output stream) and data specifying the resources available to that user.
  - *The state class.* This supported state-saving within a user process. There was, unfortunately, no commonality of mechanism between the state class and the user class, despite the similarity of some of the functions supported by them.
  - *Numbers.* Integers and (short) floats were represented in a tagged form, rather than being boxed. Since the word length was 24 bits, and 2 tag bits were used, both were 22-bit quantities – decidedly short in the case of floats. Long (48-bit) floats were also provided as a separate data class.
- Code for creating users' record and strip classes.
- The compiler, comprising a tokeniser and compiler proper with code generator. The compiler proper was available to the user as the function popval.
- *Functions to create arrays.* Arrays in POP-2 were doublets, produced by the function NEWANYARRAY. A simpler but less general interface was NEWARRAY. In POP-11 notation, here is a use of this function:

```
define sum = newarray([0 9 0 9], nonop +);
enddefine;
```

This creates in effect an addition table for the numbers 0 to 9. Thus sum(2,3) evaluates to 5. However if one does 7->sum(2,3), then sum(2,3) now evaluates to 7.

[18] User closures were not opaque, for the user could take them apart by the functions frozval and fnpart. System-generated closures had to be opaque because the code contained therein made privileged access to peripherals.

- *The debugger*. This was the work of Rod Burstall and Ray Dunn. It worked by wrapping up a user's function in a closure which printed out the arguments before applying the function to them, and which printed out the returned value on exit. It was originally a library function, later incorporated in the system.
- *The text editor*. Another library program embodied in the system, this was a typical command-driven editor of the time.
- *Math routines* – `SQRT, LOG, EXP, ^, TAN, SIN, COS, MODULO`. Many of these shared common code for evaluating Chebycheff polynomials.
- *A library mechanism*. Originally this was conceived of as a library of paper tapes that would be loaded on request by a (human) operator.
- Code for generic functions such as `DATALENGTH` and `DATALIST`. The latter function provided a chink in the opacity of records by returning a list of their components.
- *Basic arithmetic functions*: since POP-2 was not statically typed, all arithmetic functions had to determine the types of their arguments, if necessary performing conversions before doing any arithmetic.
- *Printing functions*. Apart from simple entities (integers or short floats), an object would be printed using the print method in its key cell.
- *Error handling functions*. When an error was detected a user-definable error function was called to report the error, or, if the user chose, to handle the error in some other way. However, error reporting was never systematised to make it easy for users to tailor the way in which errors were handled.
- *Definition of extracodes*. These were pseudo instructions – in effect subroutine calls which looked like an instruction with an address. POP-2 used extracodes:
  - to call a function in the update sense, normally to assign to a field in a data structure.
  - to perform a backward jump. This extracode differed from the machine-code backward jump in that a check on use of machine resources by the current user was performed. Since swapping between users could only occur when the system was in a legal state with respect to the storage allocator, it was necessary to check on the use of resources at points in which legality could be guaranteed. The chosen points were backward jumps and function entries. These combined were sufficient to bound the resources available to a given user.

2.2. The ICL 1900 Implementation

This implementation by John Scott was commissioned by Andrew Colin. It was interpretive, since POP-2 did not map economically into the instruction set of the ICL 1900 series of machines.

2.3. The ICL System 4 Implementation

This was done by John Barnes and Rod Steel [Bar68], using a machine-independent version of the language written in POP-2 [BaP68]. It too was interpretive. A later version was written in IMP by Hamish Dewar.

2.4. The DEC-10 Implementation

This original implementation in 1969 was by Malcolm Atkinson and Ray Dunn, commissioned by Lancaster University. It was in use for a number of years at Edinburgh, supporting robotics work. Later it was enhanced by Robert Rae as *Wonderpop* to be the only development of the language that attempted to generate efficient code for numerical algorithms.

2.5. POP-11 and POPLOG Implementation

This is the major extant implementation of POP [And89]. The main architect of this system was John Gibson.

POP-11 was originally developed at the University of Sussex under the direction of Aaron Sloman. Intended to support teaching as well as research, it had a revised syntax which regularised the bracketing of syntactic constructs (for example `if...then...endif` as compared with `IF...THEN...CLOSE` of POP-2) and a boolean datatype. A comprehensive online help system was developed, along with a screen editor, VED. Generally POP-11 has provided some valuable opening up of the system to the user. For example, the data structure descriptors (*keys*) are now explicitly accessible to the user.

POPLOG was developed initially for the VAX, but has been ported to most major machine architectures, and to many versions of Unix, and is freely available from `http://www.cs.bham.ac.uk/research/poplog/freepoplog.html`. A restricted version is available for Windows NT.

- It was designed from the ground up as a portable system, with as much as possible being written in a *system dialect* of POP-11,[19] which plays the same role as POP1.5 did for POP-2, but with much more sophisticated code generation able to produce quality code for all operations except floating point.[20] The system dialect permitted operations necessary to create the storage management system, in particular pointer arithmetic.
- POPLOG provides an extended library capability via *autoloading*.
- POPLOG provides some capabilities to support the Prolog language.
- Portability is supported through a Poplog Virtual Machine [SSG92]. This can be seen as being derived from the 4130 compiler by replacing the direct generation of 4100 code by a repertoire of code-generation functions. Thus where the 4100 compiler would emit an `MVE` instruction to push an argument on the stack, the Poplog VM compilers call the `sysPUSH` function. This does not necessarily result in an actual stack-push instruction being generated, which is quite inefficient in most architectures. Instead, depending on context, a register load may be issued. In order to port POPLOG to a new architecture, the Poplog VM must be implemented for that architecture.
- POPLOG supports the incorporation of code from C and other languages.
- By virtue of the virtual machine, POPLOG provides a vehicle for language implementation, and has been used to provide Prolog, Common Lisp, SML and Scheme.

2.6. Other Implementations

A number of other implementations of POP-inspired languages have existed. The earliest was H. Townsend's *Basic Pop* on the Eliot 903 computer. Another small-machine version was Bill Clocksin's POPPY for the DEC-11. In the 1980s came AlphaPop for the Mackintosh, a rather comprehensive interpretive system written in C.

Through the involvement of Steve Leach and Chris Dollin, POP-11 has influenced the Spice Language, though their affection for POP-11 style syntax seems to have lost out to a C-like form [DoL98].

3. The Library

Some of the basic system utilities, including access to files by name, were available as part of the program library. The library also contained a number of stand-alone game-playing programs, two function-memoisation capabilities, a statistics package, a proof checker and some support for computer-assisted learning. Contributors were: A. P. Ambler, D. B. Anderson, S. Arrell, R. M. Burstall, J. E. Doran, R. D. Dunn, D. Marsh, D. Michie, M. N. Mitchison, R. H. Owen, R. J. Popplestone, D. J. S. Pullin, R. H. Rae and S. Weir. Library files listed in the 'Silver Book' [BCP71] are:

1. LIB ALLSORT: an implementation of Hoare's quicksort [Hoa61].
2. LIB ASSOC: provides associative lookup and update.
3. LIB CALL AND EXPLAIN: calls functions interactively.
4. LIB DCOMPILE: Records an interactive session on the disc.
5. LIB DEBUG: wraps functions up in a closure that traces arguments and values.

[19] The relationship with the earlier POP-2 in POP-2 [BaP68] is not clear. There appears to have been continuity in the compiler strategies employed before code generation, though POP-11 provides useful hooks into compiler functions not available in POP-2.
[20] Math functions are imported from the C library.

6. LIB EASYFILE: this is a library program that provides files accessed by names in a directory. It uses primitives built into Multipop.
7. LIB EQUATIONS: provides an interactive check of a solution of an equation.
8. LIB FOR: a macro which provides a for-loop capability.
9. LIB FOURS: a program for playing 3-D, $4 \times 4 \times 4$ noughts and crosses.
10. LIB FULL MEMOFNS: Provides an implementation of Michie's *memo functions* [Mic67, Pop67, Mic68].
11. LIB GRAPH TRAVERSER: supports heuristic search.
12. LIB INDEX: builds an index of references in text.
13. LIB INVTRIG: provides inverse trigonometic functions.
14. LIB KALAH: plays the game of Kalah.
15. LIB MATRIX: provides matrix manipulation using a functional representation.
16. LIB MEMOFNS: provided the first implementation of memoisation of functions proposed by D. Michie [Mic67, Pop67, Mic68].
17. LIB NEW STRUCTURES: generalises records and arrays using LIB ASSOC.
18. LIB PLOT: plots or tabulates a function on the teletype terminal.
19. LIB POPEDIT: provides sequential text editing.
20. LIB POPSTATS: a conversational statistical package.
21. LIB PROOF CHECKER: a system for checking proofs using Robinson's resolution [Rob65].
22. LIB QUIZZING MACHINE: tests knowledge of translation from one language to another by generating a random semantic structure, generating text in one language and comparing the typed-in answer with that generated for the other. Languages tested were English, German, Finnish and elementary algebra.
23. LIB RANDOM: generates pseudo-random numbers by a congruence method.
24. LIB RANDPACK: generates pseudo-random numbers from various distributions.
25. LIB SETS: provides utilities for sets represented as lists.

4. Might POP-2 Have a Broad-Spectrum Successor?

For its time POP-2 had some significant virtues, though these were not always so visible to observers as its idiosyncrasies. I'd like to consider whether there is scope for a successor language that would offer:

1. *Broad-spectrum coverage.* The language should be suitable for specifying computations ranging from low-level control algorithms for embedded systems all the way to computational algebra and theorem-proving systems. Moreover it should appeal to a broad spectrum of potential users.
2. *Verifiable correctness.* The language should be designed to support the verification of programs written in it. This requirement makes *reflexive* our earlier goal of having POP-2 support the activity of proof.

   To reconcile these goals is challenging because:

- The language of proof is foreign to many programming practitioners. We address this by expressing proof in a familiar notation, treating theorems as *methods* which always return true.
- The well-understood techniques of proof constrain a language in its spectrum of applications. This can be addressed by defining the whole language in terms of its functional subset, proof then being conducted in terms of a functional concept of evaluation.[21]

[21] Throughout this discussion I am using 'functional' in the sense that, within a given lexical context, a function or method applied to equal arguments yields equal results. This constrains what is an acceptable concept of equality, but leaves unanswered the issue of whether functions should have first-class status.

4.1. The Problem of Data Descriptors

A crucial problem for language design is that of choosing data descriptors. A data descriptor *denotes* a class of entities. We say that a data descriptor *applies to* an entity if the language permits it to be used to describe the entity, whether or not erroneously. We say that a data descriptor *holds for* an entity if the entity (or what it denotes) belongs to the class denoted by the descriptor. In SML, if `e` is an expression and `d` is a data descriptor, then we write `e:d`, so applying `d` to `e` to claim that `d` holds for the possible values of `e`.

- *Static data descriptors* apply to an identifier or source expression and specify a set of values the programmer believes it may have. If a programming language has a *sound* static type system, then, if a program compiles, each static descriptor holds for any entity which can be a value of the identifier or expression to which the descriptor is applied. Static descriptors are typically compile-time entities, though they may be made available at run-time in modern programming environments.
  Static descriptors support a measure of program verification through type-checking. They also support the generation of efficient code because they allow the early resolution of some polymorphic operators.
- *Dynamic data descriptors* apply to actual data structures. They exist at run-time. If a programming language has a sound dynamic type system then the description attached to an object is bound to be correct. It is possible for the same entity to act both as a static and dynamic descriptor.

For example, C has only static data descriptors, and is unsound. POP-2 had only dynamic data descriptors and was sound. POP-11 is rendered unsound by the existence of *fast* functions. SML [MTH91] has both and is sound. Java [GJS96] has both static and dynamic data descriptors, which have a shared external existence as `.class` files.

Clearly the *holds for* relationship is many–many. In SML we may write `x:int` and `y:int`. But if we make the type definition `type num = int`, we may write `x:num`. Moreover a data descriptor may hold for a class of entities for some of which a particular operation will be illegal while being legal for others. For example, if we say `xs:int list` in SML, then the expression `hd(xs)`[22] will raise an exception if the value of `xs` happens to be the empty list. In order to raise this exception, some kind of dynamic data descriptor is required, be it only a tag field. Such a class may well be regarded as a *union* of classes of entity, though it should be noted that in SML there are no static descriptors which apply to the classes of which the union is composed. Consider, for example:

```
datatype tree = Node of int*tree*tree | Leaf;
```

While a program may determine (using a pattern) if a given `tree` is a node or a leaf, neither can be referred to in the type language.

4.2. Canonical Dynamic Descriptors

There is a case for saying that an entity created by a programming language should have a *canonical descriptor*. This needs to provide information necessary for storage management[23] and for determining whether certain operations are legal.

The key cells of POP-2 were *canonical dynamic data descriptors*, and indeed the language admitted none other. Java too has canonical dynamic descriptors. Each Java object has one unique dynamic data descriptor attached to it, specifying the unique lowest (or *base*) class to which it belongs in the hierarchy. Other dynamic descriptors which hold for a given object are accessible through the `super` operator.[24]

The dynamic descriptors of Java and POP-2 both serve to provide access to *methods* which are code-blocks,

[22] Assuming we've opened the `List` structure.

[23] How much information is required depends upon the nature of the unions that can occur in the language, as well as the strategy for storage management. A stop-and-copy garbage collector, for example, can be implemented by compiling special entry and exit code for functions which allow the state of the computation to be dumped and restored [Fer99]. This requires only as much discrimination between components of a union as is required for any other code. On the other hand, a mark-and-sweep collector may encounter a data-block without any preconceptions as to its type.

[24] Rather than through an upward cast – a point that is often misunderstood about Java is that a cast of an expression does not change the dynamic descriptor attached to the object which is the value of the expression at run-time, but only the static descriptor attached to the expression at compile-time.

specific to the basic class to which an object belongs, for performing a particular kind of function (such as printing). They also support the testing of whether an object belongs to a given basic class, and checking the validity of certain operations. The methods in POP-2 were system-created and fixed in number for a given class, whereas Java program code consists entirely of user-defined methods quite flexibly attached to classes.

In the case of Java the *only* kind of operation that can give a dynamic type error is a downward cast. This involves testing whether the canonical dynamic descriptor for a given object belongs to a set of descriptors which may hold for the class to which the cast is made.[25]

4.3. Engineers Like Objects, Mathematicians Like Functions

Mathematicians typically allow themselves to view a given entity in whatever appropriate way suits their purpose. For example, a pair of real numbers may be regarded as a complex number if that makes it easier to reason about them. In effect, mathematicians have a repertoire of entities about which they can reason, typically structuring them into sets upon which functions operate or properties hold.

This kind of approach is available in SML. Suppose we have said:

```
type point   = real*real;
type complex = real*real;
fun add_complex((x1,y1), (x2,y2)) = (x1+x2,y1+y2);
```

then we can use `add_complex` to add two points, since the types are synonymous.

Likewise the C language allows a pointer to a pair of floats to be cast (via `void*`) to a pointer to a user-defined complex number type, although there is absolutely no check on the correctness of the cast. In both C and SML we may rely on the structural equivalence of two different data descriptors to allow us to treat an entity declared with one type as having another irrespective of the original intent of the designer of the first type.

In Java, on the other hand, whether a `Point` object can be regarded as being a `Number` depends not on structural identity but on the relationship between the two descriptors `Point` and `Number` in the class hierarchy or interface mechanism. The writer of the `Point` definition in Java has to take into account the possibility of it being regarded as a complex number; this is not the case in C.

We may see object orientation as arising from the impact of a view of the form of engineering knowledge in general upon language design. *Objects* (real, physical objects) are a major topic of engineering discourse. They differ from the entities of mathematics in the concept of *identity*, and in having *state*. In mathematics, two complex numbers which have the same real and imaginary part are *equal*. But in the real world, two screws which have the same parameters are *different*. Moreover objects in the real world have *state*. A switch, for example, has states, on and off, say. Objects in Java have these attributes of real-world objects – two objects constructed with the same parameters are different according to the == operator, and characteristically they have *state*.

It is not unnatural if we are thinking about trying to create well-engineered software to think in terms of ‘objects’ which will behave in a manner more-or-less analogous to those in the real world. Indeed, much of the early impetus towards developing this approach arose from the need to *simulate* objects existing in the real world, in particular in the design of the Simula language [DaN67].

Engineering knowledge puts objects into *classes*, which have a *hierarchy*. A given electronic component may belong to the class of *diodes*. Diodes may be further classified into Zener diodes, light-emitting diodes, signal diodes, power diodes.

Such a class has standard attributes. The class of diodes has at least the *maximum current* attribute and the *peak inverse voltage*. Subclasses will have additional attributes. Light-emitting diodes have a *wavelength* attribute and an *efficiency* attribute. This kind of knowledge structure is mirrored in O-O languages, in which a `Diode` class might be extended to a `LightEmittingDiode` exhibiting the extra attributes of that class.

Thus, arguably, object orientation in programming languages attempts to structure software systems along the lines of standard engineering knowledge.

[25] The existence of the `null` in a Java pointer violates a simple concept of soundness under which if a static data descriptor holds for a field-access expression then a run-time error cannot occur. In the author’s opinion this is a grave flaw in the design of Java.

4.4. Theories and Object Classes

However, objects are not the only topics of engineering discourse. Mathematics and the sciences play a major role. A decision as to whether a particular artefact is well engineered will depend upon analyses that take place in the language of these disciplines. We may say that this aspect of engineering knowledge is structured into *theories*. Certainly aspects of theory are embedded into object-centred engineering knowledge – there are diode equations, for example. However, should *all* theory be embedded into a universe of knowledge structured into an object hierarchy?

Programming languages usually offer, using types, a very weak theory-construction language; weak as it is, it helps us think about where theories should be hung in a language.

We have seen above that the usual conventions of O-O languages with respect to type, state and identity do not match well mathematical concepts. And there is yet another mismatch. A mathematical theory, Group Theory, say, is defined by a set of axioms. It has many distinct implementations, for example the cyclic group of order 4, C_4 and the 3-D special orthogonal group SO(3). Superficially one might try defining an abstract class `GroupTheory` in Java, and coding up various groups each as an extension of `GroupTheory`. But the view presented by Java is that the instances of each such extension form a subset of the instances of `GroupTheory`. From the mathematical point of view `GroupTheory` is like one algebra with multiplication being a partial function over it, and not a way of characterising a class of algebras for each of which multiplication is a total function.

Not surprisingly, SML provides a framework much closer to mathematics. *Theories* are represented by the *signature* concept; they are realised using the *structure* concept. Structures may be built using *functors*. The concept of type is distinct from that of a structure or signature so that it's possible to distinguish between two sets of entities, each involved in a different implementation of a theory.

4.5. A Language with Descriptors as First-Class Citizens

Given that object orientation hierarchies capture an important aspect of engineering knowledge, but typically fail to represent mathematical knowledge well, let's consider how, from an O-O base, one could create a language that could do both. The key to this is to extend the concept of data descriptor to encompass mathematically oriented classes as well as object-oriented classes. The formalism I envisage is C-based, and I'll illustrate the ideas by examples.

Let's consider a declaration:

```
type Point = class{public float x; public float y;
                    public Point mid(Point p1, Point p2);
                      {Point(0.5*(p1.x+p2.x), 0.5*(p1.y+p2.y));}
                   }
```

This has the following effects, assuming it is not made within another type definition:

- It introduces a new name `Point` whose value is a descriptor. Once this name is given a value by the definition it cannot be assigned to. However, this is not an ordinary definition, because the name being defined *occurs* in the right-hand side of the definition. It is *recursive*.
- `Point` holds for an expression `e` if the value of `e` always consists of a sequence of two floating-point numbers. In other words we use structural equivalence of types, at least for types with public fields.
- Every descriptor has a unique *super-descriptor* except for the descriptor which is the value of `Entity`. By default, a descriptor occurring at the top level of program text has the `Entity` descriptor as super-descriptor. The exact meaning of `Point` depends on the definition of the types `float` and `int` which are defined in `Entity`.[26]
- The definition creates a *default constructor* called `Point`. We avoid the existence of a separate `new` construct.

[26] This avoids the problem that arises in C of the meaning of a program being machine-dependent, while avoiding the rigidities of Java, where `float` and `int` have fixed meanings that may well not match a given architecture or suit a given application.

- The definition includes a *method* called `mid`. This is an instance method, but is dispatched on the first argument which is of the right type.[27]
- A `return` statement is not required when the body of a function consists of an expression statement.

4.6. Extending Descriptors

A given descriptor may be extended. Suppose we have said

```
type Person = class{String name; int age; boolean ismale;
                    String toString(Person p)
                        {"Person: " * toString(p.name) * toString(p.age);}
                    };
```

then making the definition

```
type Student = Person * class{String courses[ ];}
```

will create a descriptor which is a person descriptor with an additional field.[28] This descriptor will *inherit* the `toString` method. However, we can *override* the `toString` method:

```
type Student = Person * class
                  {String courses[ ];
                   String toString(Student s)
                      {"Student: " *  toString(s.name)
                             * toString(s.courses);}
                      }
```

4.7. Theorems Say Things about Classes

Type language is of limited expressive power. It has long been recognised that much stronger statements can be made by using *theorems* [BoM75, KST97].[29] To understand our concept of theorem we first need to introduce the `boolean` type consisting of two entities `true` and `false`, together with its subtype `|-` (said 'turnstyle') which consists of just the `true` entity.

In this framework, a *theorem* is a function whose return type is `|-`.[30] For example, consider the following:

```
|- thm1(Point p1, Point p2) {mid(p1,p2) == mid(p2,p1);}
```

This says in effect that `mid(`e_1, e_2`) == mid(`e_2, e_1`)` always returns the result `true` provided the evaluations of expressions e_1, e_2 terminate normally. However, a user can only be allowed to write a theorem in a descriptor in the very special case in which there's no chance of him being wrong. Usually we would have to use a *proof method* to create a theorem. For example, we might say:

```
type Point =
      class
        {float x; float y;
         Point mid(Point p1, Point p2);
            {Point(0.5*(p1.x+p2.x), 0.5*(p1.y+p2.y));}
        } *
```

[27] This makes explicit a fact about Java, namely that an instance method for an object has an implicit first argument, namely the object itself. Our convention avoids the necessity for a `this` construct.

[28] We are not following the Java convention of using '+' for string concatenation since mathematically this is thought of as a product.

[29] Indeed, to see types as the major mode of verification is to put the cart before the horse. Typed lambda calculi were developed to make sound the illative use of the lambda calculus.

[30] I am indebted to Dr Sheard [She99] for this idea, which arose in conversations about proof in his MetaSML. It avoids many of the problems we foresaw in creating theorems about a language with call-by-value semantics. A theorem is a method guaranteed to terminate without exception, yielding `true`. For example `|- eq(Entity x){x==x;}` is valid, whereas with free variables `x==x` is not valid, for `x` would not terminate if a non-terminating expression were substituted for it.

```
        Provers.prove1(
             '|- thm1(Point p1, Point p2) {mid(p1,p2) == mid(p2,p1);}');
```

This, if it succeeds, will have the effect of creating a descriptor for objects of type `Point` which contains the desired theorem. Here `Provers.prove1` is a *proof method* that will attempt to convert the quoted structure into a theorem valid in the `Point` class, using *system axioms* about the commutativity (but not associativity!) of floating point arithmetic, and basic proof methods encapsulating knowledge about method invocation.[31] We are assuming that `Provers.prove1` is a high-level prover built on basic proof methods. It takes a quoted expression which is the theorem required and attempts to find a proof.[32]

When a class is extended its theorems are not, in general, inherited, but must be replaced verbatim. For when the class is extended, instances of the extension are instances of the original class, so the theorems are required to be valid, but the original proof will cease to be sound if methods it requires are overridden in the extension. A descriptor that is extended without restoring its theorems is said to be an invalid descriptor, and may not be instantiated.

4.7.1. The Logic of the Functional Subset

A method is said to be *functional* if, when applied to equal arguments, it produces equal results. A class is functional if all its methods are functional. Because we want *e==e* to evaluate to true if *e* terminates[33] it is necessary for the default definition of == for a functional class to be structural equality as in SML.[34]

4.7.2. Halting and Implication

Because a theorem is guaranteed to terminate without exception we can use it to express the fact that a computation terminates. The standard method `halts` is defined as follows:

```
    boolean halts(Entity x){true;}
```

The utility of this definition is that `halts` can be put in system axioms; for example:

```
    |- thm_div0(int x, int y){y==0 || halts(x/y);}
```

says in effect that integer division by a non-zero quantity always terminates without exception *because of the interpretation put upon theorems*, namely that they always return `true` without exception. Note that the validity of `thm_div0` depends on the 'short-circuit' rule for evaluation of the `||` operator.

To assist in writing theorems we introduce an implication operator =>. To evaluate e_1=>e_2 we first evaluate e_1. If it evaluates to `false`, then the result is `true`, otherwise evaluate e_2 to obtain the result.

4.8. The Semantics of Type Definitions

Since a type definition may be recursive, it is not possible to use the usual rule for evaluating an initialised declaration. In the above example `Point` would need to have a meaning before the right-hand side is evaluated.[35] The definition must be understood as a recursion equation. The name `Point` is to be bound to a descriptor which is equivalent to taking the structure definition and adding the theorem to it. A solution for the equation can be constructed by creating the descriptor as a circular data structure that is progressively updated.

[31] Here we are following the HOL system [Gor87], which exploits Milner's idea that a theorem is a data type whose only constructors are proof methods.

[32] Following Sheard [She98], quoted expressions come with an associated context of declaration bound in, so the prover has the declaration context of a quoted expression available to it.

[33] That is, we treat reflexivity of equality as a proof rule and not an axiom.

[34] Featherweight Java provides a functional approach to object orientation [IPW99].

[35] This is of course analogous to the problem of defining a recursive function.

4.9. Abstract Classes

A method may be declared rather than defined, in which case it is called an abstract method. A descriptor that contains an abstract method is called an abstract descriptor. It may not contain any fields (including inherited fields). However, an abstract descriptor may contain non-abstract method definitions.

```
type Ordered = class
    {boolean <= (Ordered o1, Ordered o2);
     boolean <  (Ordered o1, Ordered o2) {o1<=o2 && o1!=o2;}
    }
```

Abstract descriptors may contain theorems (called *axioms*) and, in exception to the usual case, the user may write these theorems. This is permissible because while any such theorem must be valid a counter example for a theorem requires an instance of the class, and the only way we can create an instance of an abstract class is to extend it. If we do this, we must reprove the theorems for the extended class.

So we could have written Ordered thus:[36]

```
type Ordered = class{
    boolean <= (Ordered x, Ordered y);
    boolean <  (Ordered x, Ordered y){x<=y && x!=y;}
    |- reflexive(Ordered x){x<=x;}
    |- antisymmetric(Ordered x, Ordered y){x<=y && y<=x => x==y;}
    |- transitive(Ordered x, Ordered y, Ordered z)
        {x<=y && y<=z => x<=z;}
    }
```

Some abstract classes may never have instances. They are *unrealisable*.[37] The most important such class is the class Real of real numbers, because it allows a theoretical basis of operations involving representations of reals to be created [Har94]. We might, for example, say:

```
type RPoint = class
    {Real x; Real y; }

type RLine = Geom * class
    {Real a, Real b, Real c,

     static boolean on(Point p, Line l)
        {p.x*l.a + p.y*l.b + l.c == 0;}

     static RLine join(Point p1, Point p2) /*line between points*/
        {Real x1 = p1.x, y1 = p1.y, x2 = p2.x, y2 = p2.y;
         RLine(y1-y2,x2-x1,x1*y2 - x2*y1);}
    }
```

Now, consider the theorem:

```
|- thm1(RPoint p1, RPoint p2) {on(p1,join(p1,p2));}
```

Let us observe that this theorem can be proved for points defined over the reals, because addition and multiplication are associative.[38] On the other hand, over the floats, the theorem would not hold, because floating point addition is not associative.

However, the class of reals is unrealisable: we might try to represent reals using Cauchy sequences, but equality of reals is undecidable for this or any other representation. So, if we want to do any computation we have to transform our definition using the reals into one we can compute with

[36] We can meet the requirement for there to be multiple independent implementations of the ordered set concept by making Ordered be a type function taking a type as parameter.

[37] This concept must be distinguished from that of *unsatisfiable* which arises from the logical inconsistency of definitions and theorems.

[38] There is of course a bug in the definition, because the 'line' $0x + 0y + 0 == 0$ can be represented, and would be created by joining two identical points. The theorem, nevertheless is valid.

```
type Point = Entity.Class.represent(RPoint,[(Real,float)],[ ]);
type Line  = Entity.Class.represent(RLine, [(Real,float)],[ ]);
```

Here the method `Entity.Class.represent` transforms the `RPoint` and `RLine` classes, rebuilding them with the `Real` type replaced by the `float` type. All theorems will be deleted from the classes during this operation.[39] This approach allows us to generate code for the computer with more assurance of its correctness than can be provided by type theory (for example, writing `y2-y1` for `y1-y2` in the above example leaves the code type-correct).

4.10. System Axioms and Implicit Class Axioms

We have already referred to *system axioms* which are a collection of theorems created at system initialisation time. They describe the behaviour of the methods built in to the system. For example:

```
|- condition_true(Entity o1, Entity o2)
       {true?o1:o2 == o1}
```

In addition, whenever a new class is created, it is automatically populated with theorems which specify how field access is related to constructor use. Suppose we say:

```
type Point = class
    { float x; float y;
    }
```

then a subclass `Point.Theorem` is automatically created, populated with the theorems:

```
|- x(float a, float b){Point(a,b).x == a;}
|- y(float a, float b){Point(a,b).y == b;}
|- halts_x(Point p){halts(p.x);}
|- halts_y(Point p){halts(p.y);}
```

in the case of a recursive class, methods supporting proof by induction over that class are generated.

4.11. Primitive Proof Rules

In our approach to proof we follow Milner in treating theorems as members of a data-type whose constructors are primitive proof rules. Our general approach to these is to adapt the primitive proof rules of HOL [Gor87] to the call-by-value semantics of our system.

We envisage the proof methods referred to above, such as `Provers.prove1`, being crafted out of these primitives. Unlike HOL [Gor87], for which β-reduction is sound, our system is call-by-value, so that β-reduction, or its equivalent, is unsound without a halting proof. So a primitive step of inference might be from:

```
|- thm1(Point p1, Point p2) {halts(mid(p1,p2));};
```

to infer by a step equivalent to β-reduction, and using the definition of `mid` from the environment:

```
|- thm2(Point p1, Point p2)
       {mid(p1,p2)== Point(0.5*(p1.x+p2.x), 0.5*(p1.y+p2.y));}
```

4.12. Theorems in Mutable Classes

We propose to treat classes of mutable objects by defining the whole language in the functional subset. Reasoning about mutable classes involves functional reasoning about the value of an imperative expression in a state, using the syntax *e* `in` *s*.

[39] Attentive readers will have noticed that the `on` method in the transformed code is of little use, since it will now depend on the equality of floating point numbers. A practical approach might be to replace equality with approximate equality.

For example, we might characterise a queue abstractly, expressing a requirement that if an object is inserted in it in a given state, then a `members` method that makes a list of the entries will contain the object as its last member in the state resulting from the insertion.

```
type AbsQueue = class{
    void insert(Entity x; AbsQueue q);
    void next(AbsQueue q);
    void dequeue(AbsQueue q);
    type LO = List(Entity);  /* apply the type-function List */
    LO members(AbsQueue q);  /* makes a list of queue members */
    |- thm1(Entity x, AbsQueue q, State s)
        {halts('insert(x,q)' in s) ==>
          LO.last('members(insert(x,q))' in s) == x;}
```

5. Discussion

In this paper I've provided an account of the development of POP-2, seen from my perspective. I've explored the question of extending the work in a direction more influenced by object orientation than the functional approach of SML. This is not without its difficulties, particularly in handling the many theories which do not (up to isomorphism) uniquely characterise their implementations and so require distinction to be made between different implementations. However, I would see the use of descriptor-valued functions as being sufficiently versatile to cope with this.

Many of the tools for an implementation of the proposed language exist, for I have already implemented an interactive version of C in POPLOG. I am now developing the syntactic rules for the new language and designing the required abstract syntax.

References

[And89] Anderson, J. A. D. W., editor: *POP-11 Comes of Age: The Advancement of an AI Programming Language.* Ellis Horwood, Chichester, 1989.

[Bar68] Barnes, J. G. P.: *System 4 POP-2 Users Guide.* Department of Machine Intelligence and Perception, Edinburgh, 1968.

[BaP68] Barnes, J. G. P. and Popplestone, R. J.: *POP-2 in POP-2.* Property of the University of Edinburgh Round Table, Department of Machine Intelligence and Perception, 1968.

[BBH63] Barron, D. W., Buxton, J. N., Hartey, D. F., Nixon, F. and Strachey, C. S.: The main features of CPL. *Computer Journal*, 6:134–143, 1963.

[BoM75] Boyer, R. S. and Moore, J. S.: Proving theorems about LISP functions. *JACM*, 22(1):129–144, 1975.

[Bur69] Burstall, R. M. Proving properties of programs by structural induction. *Computer Journal*, 12:41–48, 1969.

[BCP68] Burstall, R. M., Collins, J. S. and Popplestone, R. J.: *POP-2 Papers.* Edinburgh University Press, Edinburgh, 1968.

[BCP71] Burstall, R. M., Collins, J. S. and Popplestone, R. J.: *Programming in POP-2.* Edinburgh University Press, Edinburgh, 1971.

[BuP68] Burstall, R. M. and Popplestone, R. J.: POP-2 reference manual. In E. Dale and D. Michie, editors, *Machine Intelligence 2.* Edinburgh University Press, Edinburgh, pages 207–246, 1968.

[DaN67] Dahl, O.-J. and Nygaard, K.: *SIMULA: A language for Programming and Description of Discrete Event Systems*, 5th edition. Norwegian Computing Center, Oslo, 1967.

[DoL98] Dollin, C. and Leach, S.: Evolving ECMAScript into Spice: a rationale. `http://www.w3.org/People/Raggett/Spice/ECMA/rationale.html`, 1998.

[Dun70] Dunn, R. D.: *POP-2/4100 Users' Manual.* Department of Machine Intelligence and Perception, Edinburgh 1970.

[EFG71] Elcock, E. W., Foster, J. M., Gray, P. M. D., McGregor, J. J. and Murray, A. M.: ABSET: a programming language based on sets; motivation and examples. In B. Meltzer and D. Michie, editors, *Machine Intelligence 6.* Edinburgh University Press, Edinburgh, pages 467–490, 1971.

[Fer99] Ferrari, A.: Personal communication, 1999.

[Gor87] Gordon, M.: HOL: a proof generating system for higher-order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis.* Kluwer, Dordrecht, 1987.

[GJS96] Gosling, J., Joy, W. and Steele, G.: *The Java Language Specification.* Addison-Wesley, Reading, MA, 1996.

[Har94] Harrison, J.: Constructing the real numbers in HOL. *Formal Methods in System Design*, 5(1/2):35–59, 1994.

[Hew69] Hewitt, C. PLANNER: A language for proving theorems in robots. *Proceedings of IJCAI*, Washington, DC, 1969.

[Hoa61] Hoare, C. A. R.: Algorithms 63 and 64. *CACM*, 4:321, 1961.

[IPW99] Igarashi, A., Pierce, B. and Wadler, P.: Featherweight Java: a minimal core calculus for Java and GJ. In *Proceedings of the ACM Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA). ACM SIG-PLAN Notices*, 34(10):132–146, 1999.

[KST97] Kahrs, S., Sannella, D. and Tarlecki, A.: The semantics of Extended ML: a gentle introduction. *Theoretical Computer Science*, 173:445–484, 1997.

[Lan64] Landin, P. J.: The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, 1964.

[Lan65] Landin, P. J.: The correspondence between ALGOL 60 and Church's lambda notation. *CACM*, 8:89–101, 158–165, 1965.

[Lan66] Landin, P. J.: The next 700 programming languages. *CACM*, 9:157–66, 1966.

[McC60] McCarthy, J.: Recursive functions for symbolic expressions. *CACM* 3:184–95.

[MAE62] McCarthy, J., Abrahams, P. W., Edwards, D. J., Hart, T. P. and Levin, M. I.: *LISP 1.5 Programming Manual.* MIT Press, Cambridge, MA, 1962.

[Mic67] Michie, D.: Memo functions: a language facility with 'rote-learning' properties. *Research Memorandum MIP-R29.* Department of Machine Intelligence and Perception, Edinburgh, 1967.

[Mic68] Michie, D.: Memo functions and machine learning. *Nature*, 218:19–22, 1968.

[MTH91] Milner, R., Tofte, M. and Harper, R.: *The Definition of Standard ML.* MIT Press, Cambridge, MA, 1991.

[MoL70] Moore, C. H., Leach, G. C.: *FORTH: A Language for Interactive Computing*. Mohasco Industries, New York, 1970.

[Pop67] Popplestone, R. J.: Memo functions and the POP-2 language. *Research Memorandum MIP-R-30.* Department of Machine Intelligence and Perception, Edinburgh, 1967.

[Pop68a] Popplestone, R. J. In E. Dale & D. Michie, editors, POP-1: an on-line language. *Machine Intelligence 3.* Edinburgh University Press, Edinburgh, pages 185–194, 1968.

[Pop68b] Popplestone, R. J.: In E. Dale & D. Michie, editors, The design philosophy of POP-1. *Machine Intelligence 3.* Edinburgh University Press, Edinburgh, pages 393–402, 1968.

[PDR74] Popplestone, R. J., Dunn, R. D., Rae, R. and Pullin, D. J. S.: *The Multipop System.* A print-out in the possession of Robert Rae, 1974.

[Pul67] Pullin, D. J. S.: *A Plain Man's Guide to Multi-POP Implementation.* Mini-MAC Report No. 2. Department of Machine Intelligence and Perception, Edinburgh, 1967.

[Rey70] Reynolds, J. C.: GEDANKEN: a simple typeless language based on the principle of completeness and the reference concept. *CACM*, 13:308–319, 1970.

[Rob65] Robinson, J. A.: A machine-oriented logic based on the resolution principle. *JACM* 12:23–41, 1965.

[Sco68] Scott, J.: *A Supplementary Manual for the Lancaster POP-2 System.* University of Lancaster, 1968.

[Sha64] Shaw, J. C.: Joss: a designer's view of an experimental on-line computing system. In *AFIPS Conference Proceedings (FJCC 1964)*, volume 26, pages 455–464, 1964.

[She98] Sheard, T.: Using MetaML: a staged programming language. *Advanced Functional Programming*, 207–239, 1998.

[She99] Sheard, T. Personal communication, 1999.

[SSG92] Smith, R., Sloman, A. and Gibson, J.: POPLOG's two-level virtual machine support for interactive languages. In D. Sleeman and N. Bernsen, editors, *Research Directions in Cognitive Science Volume 5: Artificial Intelligence*, Earlbaum, Hillsdale, NJ, 1992.

[Str67] Strachey, C., editor: *CPL Reference Manual.* Privately circulated. Programming Research Unit, Oxford University, 1967.

[SuW70] Sussman, G. J. and Winograd, T.: *Micro-planner Reference Manual.* AI Memo No. 203, Project MAC, MIT, Cambridge, MA, 1970.

[Tow68] Townsend, H.: *The Basic POP/903.* Regional Clinical Neurophysiology Service, Western General Hospital, Edinburgh, 1968.

[WiH66] Wirth, N. and Hoare, C. A. R.: A contribution to the development of Algol. *CACM*, 9:413–432, 1966.

Rod was the first of a rather small number of collaborators who have been central to my working life. When I met him, I was a refugee from mathematics who had discovered the obsessive pleasures of building software. Rod was consistently interested in exploiting mathematical ideas for achieving a more formally predictable understanding of computation. This difference both enriched and ultimately limited our working relationship: mathematics was a common language which did much to enable us to agree on our goals in designing POP-2, ensuring that it was a language which, for all its idiosyncrasies, could be used in a functional style; but I was reluctant to go along with Rod in developing the formal approach in depth. By the time an active use of mathematics had re-entered my working life, our interests had diverged. Some thirty years later, the years working with Rod remain a magic time, in which all things were possible.

Received October 2000
Accepted in revised form January 2002