The Poly and ML System: Low Level Support

David C.J. Matthews

9 December 1987

1 Introduction

This document forms part of a set describing the Poly[2] and Standard ML[1] systems. It is concerned with the low-level part of the system and the interface to the outside world. In particular it describes the data structures used to represent various high-level objects, and the algorithms used in the garbage collector and other low-level routines. It is intended primarily for those interested in the details of the implementation or planning to implement their own compiler for Poly, ML or a similar language.

2 Underlying Representations

The lowest level objects handled by the abstract machine are words and vectors. Before describing how the higher level objects such as strings and integers map into them the format of these objects will be described.

A word is a 32-bit value which is either a number or the address of a vector, distinguished by all numbers being less than 65536. Addresses may be either local store addresses or *persistent addresses*. Persistent addresses arise from the persistent storage system which is a kind of "demand-paging" of objects from a database into main store. The persistent storage system is described in a separate report[3] and as far as the compiler is concerned all addresses can be treated as local addresses, with transfers from backing store being done transparently. The only places where the persistent storage system has to be taken notice of is where a persistent store trap might cause a garbage-collection to take place. This requires rather more care in the code-generator than in a system where the garbage-collector can only be invoked when the storage allocator is explicitly called. It also means that testing for address equality, in ML this can happen when two reference values are compared, has to take account of the case of one address being a local store address and the other the persistent address of the same object.

Figure 1 shows the general form of a vector. The first word is the length of the segment in words together with a flag in the top byte. For a normal vector this

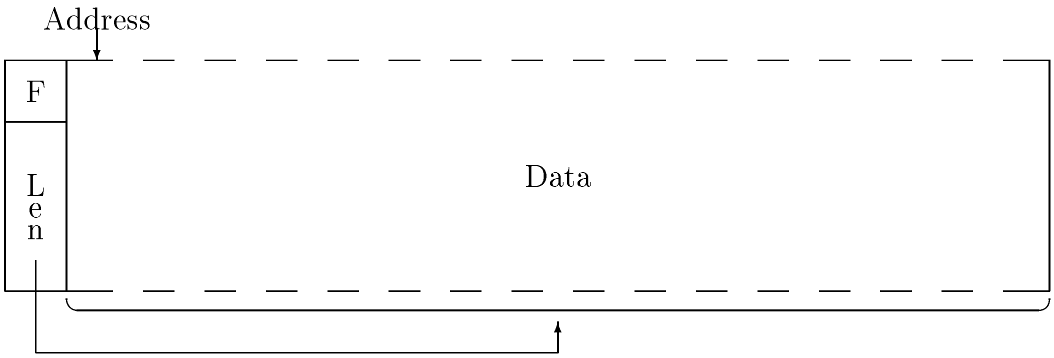

Figure 1: Vector

Bit	
0	Byte segment.
1	Code segment.
2	First word points to second.
3	Stack segment.
4	Application dependent.
5	Application dependent.
6	Segment is "mutable".
7	Used by the garbage-collector.

Table 1: Flag bit definitions

flag byte is zero. Table 2 gives the meaning of the bits in the flag byte.

Apart from the **application dependent** bits, one of which is used as a sign bit for arbitrary precision integers, the bits are only used by the store-management system. The **mutable** bit is set for segments such as variables (references) or arrays where the contents can be changed by assignment. It can be set when a segment is created and switched off later, for example to construct circular structures, but an immutable segment cannot be made mutable. It is used by both the persistent storage system and the garbage-collector where mutable and immutable segments are treated differently.

The **byte-segment** bit is used where a segment contains values which cannot be addresses of other objects. If this bit is not set the garbage-collector assumes that each word is an address of an object if it has a value greater than 65535. It is essential that the characters in a string, for instance, are not treated as though they formed addresses. **Byte-segments** are also used for long integers and reals and for some segments of executable code.

A **code-segment** is a combination of an ordinary segment and a byte-segment and is used only for segments of executable code which contain constants. Constants are either literal values, or arise from references to top-level declarations. It would be possible to load them into the closure of the procedure which uses them

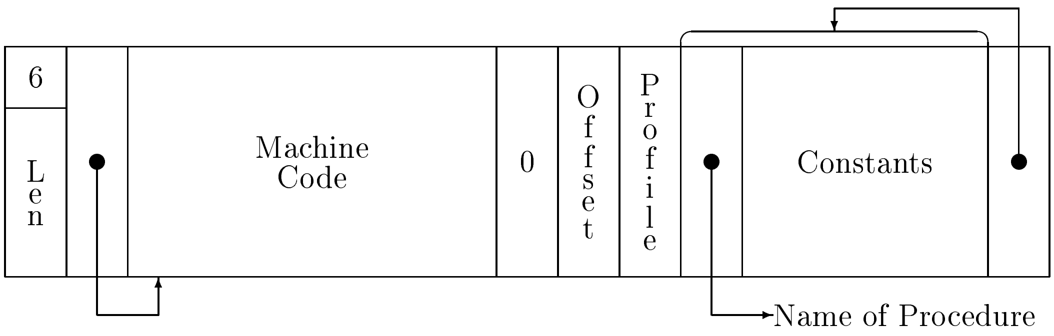

Figure 2: Format of a Code Segment.

but it is better to reduce the size of the closure by binding them directly into the code. They may be addresses so they must be inserted into the code in such a way that the garbage-collector can find them. The first part of a code-segment contains the code and this is treated by the garbage-collector as a byte segment. The constants are put at the end of the segment and the last word is the number of constants.

Figure 2 shows the format of the most general code segment which is a code-segment with bit 2 set. Like all vectors the first word contains the length field with the top byte used for a format code, in this case 6, and the address of the segment is the first real word of the segment. The end of a code segment has a special format which is used by the garbage-collector and other procedures. After the code there is a single word containing zero. The code-generator does not generate whole words with zero so this word acts as a marker. After the marker there is a word which contains the offset from the start of the vector. When the garbage-collector finds an address which points into the code, either a return address or the address of an exception handler, it must be able to locate the start of the vector. It searches for the marker and then subtracts the value of the next word from the address to get to the start of the vector.

The next word after the offset is used for profile counts. It is normally zero but if profiling is on this word contains the count. The profiler finds the word in the same way as the garbage-collector.

After that comes the name of the procedure. This is a string so it will normally be an address unless the name happens to be only one character long. It is used by the profiler when the profiling counts are printed out and by the tracing code when a trace is requested *via* a console interrupt.

Following the procedure name come any constants used by the code and the final word is the number of constants, including the procedure name. When the garbage-collector is tracing a format 2 or format 6 block it only looks in the constant segment for possible addresses and ignores the remainder of the vector. Of course when a format 6 block is moved during compaction the first word must be updated to the new location.

The **first-word-points-to-second** bit is only used with code-segments or byte segments containing code. This bit is set in a segment which contains the code of a procedure which has no non-local references, other than to constants. When a procedure is declared it is generally necessary to construct a closure which is represented by a vector. The first word contains the address of the code and the remaining words contain the non-local values used by the procedure. A procedure is called by loading the address of the closure into a register and doing a subroutine call indirecting through the first word. A procedure with no non-local references would have a closure which consisted of one word which was the address of the code. To save having odd one-word objects the closure and the code are combined into a single segment whose first word, instead of being the start of the code is the address of the seond word which is the real start of the code. Such a segment has bit 2 set and a separate closure is not created. The calling sequence however is exactly the same as if it had a closure so this segment can be passed around as though it were a closure.

The stack for each process is held in a **stack-segment** whose format is shown in figure 3. The format is designed to be usable on different machines with different numbers and arrangements of registers. The store-management system assumes that all the saved registers contain values which are either less than 65536 or point to the start of a vector. This allows the garbage-collector to treat them as ordinary addresses and to change the value if it moves a vector that a register points to. Three saved values are used for special purposes and are saved in specific fields. The stack pointer field, **sp**, contains the lowest extent of the stack, which grows down from the end of the segment. The program counter field, **pc**, points into the code which was being executed when the process was suspended, and the the handler field, **hr** points into the stack at the last exception handler data. These fields are used for producing stack traces and profiling information.

Normally values are either numbers or point to the start of vectors. However values on the stack are allowed to break the rules in two ways. The stack may contain pointers elsewhere on the stack, for instance for static links and for linking exception handlers. In addition it may contain pointers to the middle of pieces of code, either as return addresses or as exception handlers. To allow these to be distinguished from other addresses, addresses into code are never aligned on word boundaries. This is enforced by the code-generator.

3 Higher Level Values

All higher level objects are implemented in terms of these basic low level objects. This section describes how the basic types are implemented.

3.1 Structured Objects

Records and structures in Poly or tuples in ML are implemented as vectors of words. Unions are implemented as pairs with the first word containing a tag, and the second containing the injected value.

3.2 Integer and Long_integer

There are two representations of integers in Poly. The type `integer` is a short precision value between -32768 and 32767. It is represented as a word containing the number itself in the lower half of the word, with the top half zero. In Poly there is also the arbitrary precision integer type `long_integer` which is used for `int` in ML. The representation of numbers between -32768 and 32767 are the same as for `integer` making conversion easy and allowing operations on small values to be done rapidly. Numbers outside this range are represented by the address of a vector of bytes. A sign-and-magnitude representation is used with the sign-bit in the flag bits of the vector. Each byte represents a base-256 digit and the vector represents the number, low-order byte first.

Operations on short precision numbers are compiled into the code directly using the normal machine-code integer operations. However, the arbitrary precision package is written in C so operations on arbitrary precision numbers have to be done by procedure calls. In the Sun and Vax implementations the interface procedures which call the C from Poly or ML and are written in assembly-code, handle the special case of an operation where the arguments are small numbers and the result is small.

3.3 Real

Neither Poly nor ML are intended for applications which make intensive use of floating point numbers so the implementation is not particularly efficient. The representation used for real numbers is a double precision value held in a byte-vector. All operations have to be done by procedure calls.

3.4 String and Char

Strings are normally represented by the address of a vector of bytes. The first word is a number which is the number of characters in the rest of the string. Having a 32-bit length count means that the length of strings is constrained by storage availability rather than by any inherent limit.

Null strings are represented by a string with a zero length field. For efficiency a word in the interface vector is used for this and all the string operations return a pointer to this if they need to return a null string. Single character strings, including the string containing the ASCII NUL character, are represented by the character value itself. This makes some operations rather more complicated but saves a considerable amount of store, since otherwise each single character would be represented by two words. This means that a single character string has exactly the same representation as a character.

3.5 Streams

Input and output streams work through the normal Unix standard I/O mechanism however the Poly program does not operate directly on the streams. Instead the value returned by opening a stream is the address of a token which contains an index into the **stream table** which is part of the run-time system data structures outside the Poly heap. Each entry in the table is a pair of values, one of which is the address of the token and the other the address of the Unix stream. A new token is allocated on the heap when a stream is opened (figure 4), and when a stream is closed the token address in the entry is cleared (figure 5). Every time the read or write functions are called the run-time system checks that the token address part of the entry corresponds to the token that was provided. If it fails to match the function raises an exception. This allows an entry in the table to be reused if a new stream is opened but in such a way that a previous stream that has been closed will not appear to have been reopened. In figure 6 S1 has been closed and S2 reopened using the same entry. Because the token address in the entry does not point to S1 any operations using S1 will raise exceptions.

This is also useful if the database has been written out with streams open. The stream tokens will be written out and if they are read in to a new session will fail to match. The exception is the standard input and output streams, whose tokens are entries in the interface map. They are linked up by the persistent storage system so they will always match.

Another use for this is to try to ensure that unreferenced streams are closed automatically. The stream tokens are held on the heap and may be moved by the garbage-collector if the heap is compacted. The corresponding addresses in the stream table must be updated so that they will still match the tokens. It is possible that a stream may be become no longer referenced before it has been closed. In that case there can be no further activity on it and in particular it will not be explicitly closed. We can detect that the stream has become unreferenced by the garbage-collector finding that there are no references to the token, apart from the entry in the stream table. In that case the stream handler closes the stream. All this is really only necessary if there is a limit on the number of streams that can be open at once.

A similar mechanism has been used for a window system. There, however, an attempt to do an operation on a window whose token does not match can cause the window to be reopened. The token is more complicated than for a stream and holds information about the window's size and position. This is used to reactivate windows when a new session is started, using information from the persistent store. If a window has been explicitly deleted that is marked in the token, and operations on it will raise an exception.

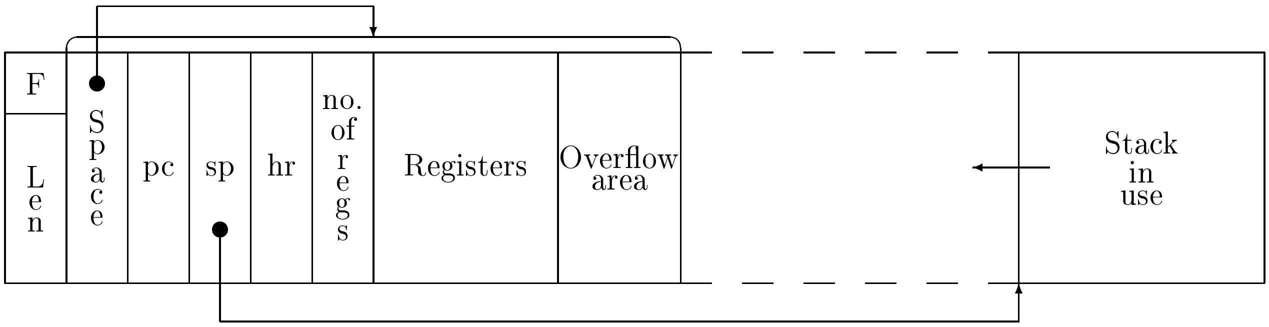

Figure 3: Format of a Stack Segment.

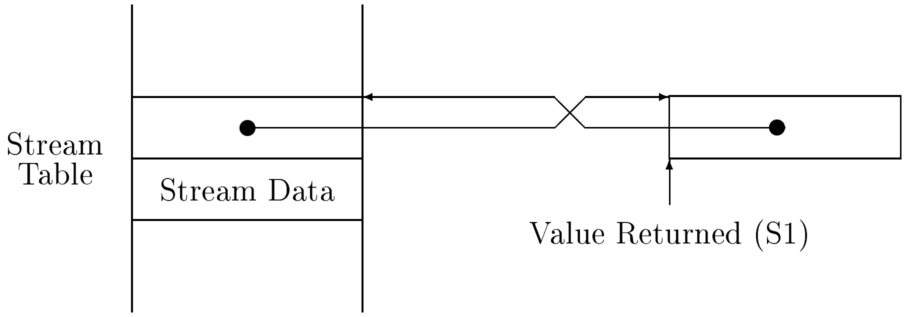

Figure 4: After Opening a Stream.

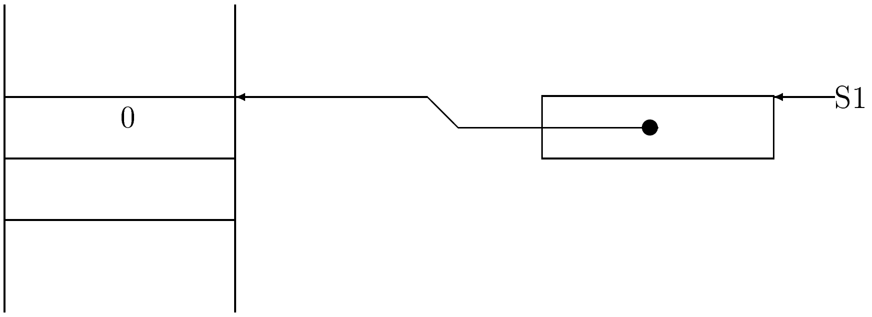

Figure 5: After Closing the Stream.

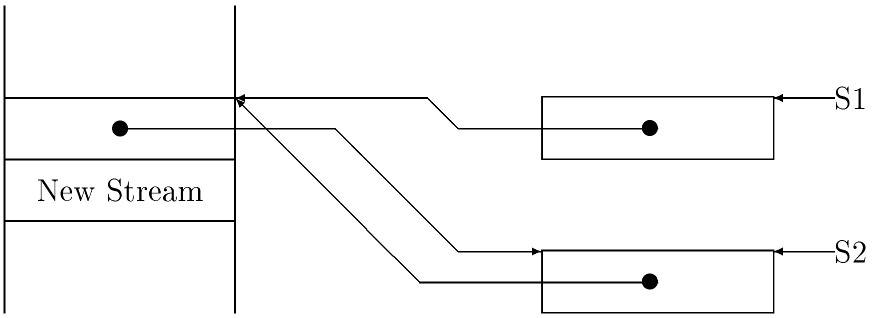

Figure 6: After Opening another Stream, Reusing the Entry.

4 Processes

The process mechanism in Poly is mostly implemented in the run-time system. Each process has a process block and a stack which is initially quite small, but it can grow if necessary by allocating a new stack segment and copying the data over into it. A register points to the lower end of the stack so that a simple comparison with the stack pointer will normally suffice to check for overflow. A larger stack can then be made, unless the size has become too large when it is assumed that the process is in an infinite recursion and an exception is raised.

Stack overflow checking also serves another purpose. It is sometimes necessary to interrupt a process when, for example a process has exceeded its time-slice. However a few pieces of code have to be treated as single units, a particularly common example being the instructions which allocate store off the heap. So, instead of interrupting the code immediately, the pointer to the end of the stack is changed so that the next time the code checks for stack overflow it will trap. The code that would normally handle the overflow trap can recognise this case and call the process scheduler.

The processes communicate on blocking channels. A channel is represented by a pair of words one of which is used as the head of a chain of processes waiting to send on the channel and the other the chain of processes waiting to receive. It is possible, but unusual, for both the chains to have processes in them.

Processes are created by two different run-time system calls. The simpler is **fork**. This creates a process that runs in parallel with its caller. It takes two arguments, one which is a function to run as the process, and the other which is a switch which determines whether the process is to receive console interrupts as **interrupt** exceptions. Runnable processes are put into a doubly-linked list. Changing the current process simply involves changing the pointer into this chain. When a process wants to send or receive on a channel and there is no process waiting to do a corresponding transfer the process is blocked. It is taken out of the runnable chain and added to the appropriate list for the channel. Note that it is possible for such a process to be garbage-collected if the channel it is waiting for is no longer reachable from a runnable process.

The other way of creating a process is with **choice**. This takes two functions as arguments and runs them as processes. Unlike processes created with **fork** these *choice* processes are mutually exclusive as far as sending and receiving messages are concerned. If one successfully sends or receives a message the other will not be allowed to. The interlocking between the processes is done using **synchroniser** values.

A process can also be blocked if it attempts to read from a stream when there is no data available. The process is suspended in such a way that when it is restarted it will try reading again, either succeeding if data is now available, or being suspended again. There is a timer which interrupts the current process and starts another one. This ensures that processes get a reasonably fair share of the machine and that processes waiting for input will run when the input arrives.

5 Addresses and the Persistent Store

The persistent storage system is designed to be transparent as far as the machine-code is concerned. The code can operate on addresses without needing to know whether it is using real or persistent addresses and the persistent store handler will take care of this. There are however two places where the use of the persistent store affects the code-generator, in comparing addresses and in the effect of garbage-collection.

Equality of addresses is used for equality of structures in Poly and for equality of references in ML. In most cases this could be done with a simple compare instruction. However it is possible to have a persistent address and a local address which refer to the same object. These should clearly be considered as equal even though the addresses are represented in different ways. To get this right address comparison is done by code which calls a routine in the persistent storage system if it finds two addresses one of which is a persistent address and the other local.

The other point about the persistent store is that garbage-collections can happen at many more places than in a system without a persistent store, where garbage-collections only take place when the store-allocator is called explicitly. This is important because the garbage-collector must be able to find and update all the addresses of objects currently in use which means that the state of all registers in particular must be well-defined. The code-generator must be aware of the potential garbage-collections and ensure that the state is well-defined at each of them. Between these places it can use the registers as it likes. The garbage-collector for the Poly/ML system assumes that every word in a register or on the stack is the address of an object if its value lies in the range of possible heap addresses. This requires all addresses to point to the start of an object and not inside one. Normally this is quite easy to achieve but a few cases have to be treated specially, usually by clearing a register after it has held a value which would break the rule.

6 Linkage Conventions and Register Usage

A similar calling sequence is used in each of the versions of the system. This has advantages because the format of a stack frame is then the same and so code in the garbage-collector and for producing trace information can be independent of the machine.

The standard calling conventions used on the VAX and MC68020 are not suitable for Poly for a number of reasons. On the VAX this is because the standard call instruction is too slow, and on the 68020 because it is possible to make use of a special return instruction which is not normally used.

The conventions differ slightly according to whether the function is being called with a static link or as a full closure. In the case of static link call a register contains the static link and the subroutine jump is to the start of the code. For a closure call the register contains the address of the closure and an indirect jump is made using the address in the first word of the closure as the destination. The arguments to the function are passed on the stack so after the call the stack contains the return address, pushed by the subroutine call, and the arguments. No other values are explicitly put on the stack. The register with either the static link or the closure address is marked on the pseudo-stack and will be pushed onto the real stack if the register is required, in particular for another function call, in same way as any other register. Small functions which do not do any function calls themselves, never need to push this register and any references to variables in the closure can be done by indirecting through the register.

The code-generator keeps a note of the number of items on the stack and addresses all the values on it using the stack pointer register. This avoids the need for a frame pointer register but makes reading the assembly code listing somewhat difficult because the offsets of local values and arguments change according to the number of items currently on the stack.

6.1 Exceptions

Raising an exception is similar to calling a procedure except that instead of returning to the point of call execution continues with the code following the handler. Exceptions are set up using a *handler register* which is either a real machine register, as on the 68020, or a location in store, as on the VAX. When a handler is set up a number of words are pushed onto the stack. The first is current value of the handler register, and then follow a number of pairs of words. Each pair consists of an exception identifier and the address of the code to handle the exception. The handler register is then set to the value of the stack pointer, in other words to the address of these words. In this way exceptions are chained together.

If an exception is raised the chain is searched for a handler with an identifier which matches the identifier of the exception. When a match is found the stack is reset to the handler, the old handler register value is reinstated and the handler code is entered. If control reaches the end of the scope of an exception handler without the handler itself being executed the previous handler is reinstated and the handler address is removed from the stack.

Zero is a reserved exception identifier and is used for a handler which will catch all exceptions. Zero as the address of the handler code is also reserved and is used by the exception tracing mechanism. The **exception_trace** function pushes a special handler on the stack with exception identifier and handler addresses both zero. It then calls its argument, a function. If an exception is raised inside that function or somewhere called from it the normal exception mechanism will involve a search for a handler. If there is no handler inside the function the special handler will match. Instead of jumping to the code, however, the exception mechanism prints a stack trace. It then reraises the exception.

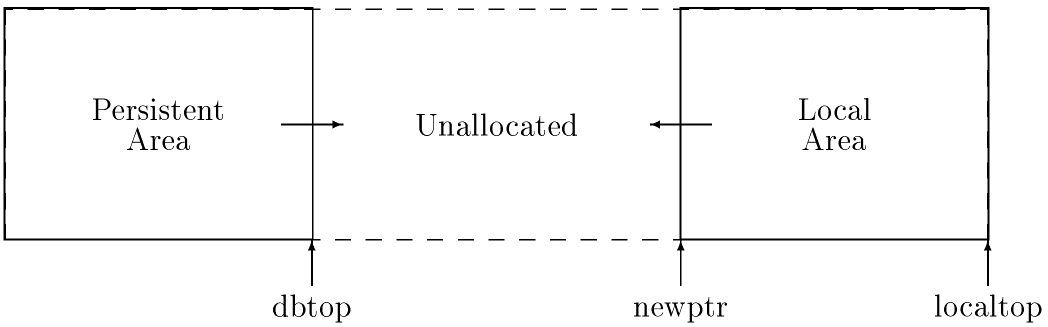

Figure 7: Overall Heap Layout

7 Storage Allocation

Storage allocation is a major problem when trying to implement languages like Poly and ML efficiently. There are several competing requirements which complicate the problem.

Store may be needed for a number of purposes. Poly or ML code needs space for local declarations and for objects such as tuples or closures. The persistent storage system needs store for blocks that are brought into store and for storage maps. The input/output system must create buffers in store and some other system calls need workspace. An ideal situation would be to have several different regions in virtual memory each able to expand independently so that each requirement could be dealt with in a separate region. Unfortunately the Unix operating system allows there only to be two segments, a *stack* segment that expands automatically as space is required, and a *heap* segment which can be extended by calls to the operating system.

The names of the segments reflect the way that C programs make use of them but there is no particular reason why Poly and ML should follow that usage. It is perfectly possible, for example, to use the heap area for the stack. Indeed if a multi-process system were implemented for Poly the stacks would probably be allocated there. In practice it is convenient to follow the C usage to simplify calls from Poly/ML to the run-time system written in C. In the interpreted system the stack is a vector allocated in the C stack, but in the Vax and 68020 versions the stack is shared between Poly and C with special pieces of assembly code handling the conversion between Poly and C calling conventions.

Dividing up the heap segment is more difficult. After some experimentation the present system was arrived at where the heap is divided in two between the *persistent* area at the bottom of the heap growing upwards, and the *local* area at the top growing downwards 7. When the areas meet the garbage-collector is called, but only the *local* area is collected.

The *persistent* area contains data read from the persistent storage system and objects in it are arranged as blocks of a fixed size. Rather than garbage-collecting this area it would be better to write the blocks back to disc if they were not being actively used, but in practice this has not been found necessary.

This area is also used for buffers and other objects created by operating system calls. If these were allocated in an area which could be compacted by the garbage-collector their internal structure would have to be known so that addresses in them could be traced. These objects are managed by replacing the standard storage manager calls *malloc* and *free* by versions which maintain a free list of areas among the persistent blocks.

The *local* area is used for objects created by the Poly or ML code or for strings or arbitrary precision numbers created by the run-time system. Allocating store for tuples or closures is a frequent operation and must be made efficient. If micro-code support is available it is possible to have quite complex allocators without reducing the speed significantly, but if allocation is done by macro-instructions then the only efficient method is to have a contiguous area of memory. The allocator then merely subtracts the size required from the current heap pointer, compares with a limit and takes a trap if it has overflowed. This can be done in very few instructions if a trap instruction rather than a subroutine call is used to enter the garbage-collector. On the 68020, for example, this is three instructions and can be compiled in-line rather than using a call to an allocator.

8 Garbage Collector

In a language like ML objects such as lists or closures are created and may then be no longer required. For the system to be efficient the store occupied by them must be recovered so it can be reused. This requires garbage-collection to sort out the objects which are still needed from those which have been thrown away. The usual process is to search for all objects which could be referred to from the registers and the stack either directly or through other objects and call these the active objects. Anything else is certainly garbage because there is no way it can be addressed.

8.1 The Basic Scheme

The garbage collection process used in the Poly/ML system is essentially a compacting scheme but some variations. The basic scheme will be described first. The garbage collector works in three phases. The first phase finds the accessible objects and therefore the garbage. A simple-minded scheme would stop there, put the garbage on a free-list and the store allocator would allocate its store from that. The main disadvantage of this is that the store would become fragmented with lots of small free objects scattered through the store. Another problem is that searching a free-list is expensive. For these and other reasons the garbage-collector compacts the store so that the active objects form a contiguous area at the top of the storage region with the free area beneath it. The second phase compacts the store by moving active objects into store which was previously garbage, and the final phase changes the addresses of objects which have been moved so that they refer to their new locations.

The objects in the heap are represented by a bit-map with a bit for each word and initially the bits are cleared to zero. The first phase of garbage-collecting starts with the addresses in the registers and on the stack and sets the bits in the bit-map for each word in an object which it finds. This is called marking the objects and is a recursive process since each object may refer to other objects. An object is marked before the objects it refers to are checked and objects which have already been marked are not scanned again. This avoids problems with loops of references. Non-recursive marking algorithms have been proposed but have not been tried in the Poly/ML garbage-collector.

At the end of the first pass the bit-map shows the words which are known to be free and those which are garbage. The second pass starts from the bottom of the area and copies each active object up to the highest free area which will hold it. When it moves an object it leaves a "tombstone" behind which gives the address it has been moved to. At some point it will not be possible to copy an object any higher because there is no free store above it and the process stops with all the active objects compacted together at the top of the store. There may be a few small areas of free store left among the active objects if the last object was a large one, but measurements have shown that they are very few. The third phase scans through all the addresses on the stack, in the registers or on the heap and changes every address which points to an object which has moved to point to the new location, using the "tombstone" which was left behind when it was moved. Finally the newly freed area is cleared to zero so that it can be reallocated.

The garbage-collector can only produce free store if there is some to be had and it is possible to have a program which keeps on creating objects and not throwing them away. The Poly/ML system starts with a relatively small amount of store for the heap and then gets more by calling the operating system if it runs out. It does this by counting the number of active words of store during the first phase of garbage-collection and then working out the amount which is free. If this has fallen below a certain amount it asks the operating system to extend the upper limit of the storage region to give some store to work with. This new area of store is treated as though it were free store so the second phase of the garbage-collection copies the active objects from the bottom of the heap up to this new area. The result is to make an area of store at the bottom of the heap available.

It might be thought better to have the active objects starting from the bottom the storage area and the free area at the top. New free store would then be added on at the end. The reason for not doing this has to do with the persistent storage system. For various reasons it is convenient for objects which have been brought in from the persistent store to be held in a separate area to the objects which have been created locally on the heap. The latter are often transient and so need to be garbage-collected while the former are longer-lived. Objects brought in from the persistent store are held in an area which extends from the bottom of the region upwards. This area grows up towards the heap as each new block brought in. When the areas meet the store must be garbage-collected.

8.2 The Full Scheme

The garbage collector as originally implemented worked as described above, but after some use it was found to have a serious drawback. Many programs, and in particular compilers, work by first building up a data structure such as a parse-tree and then processing it. While the data structure is being built up parts of it that have already been made are not actually being used. In a virtual memory system the part of the store containing them will be paged out. This works fine until the store has to be garbage collected. The scheme described must scan all the active objects at least twice, once during the marking, and again to find and update the addresses of objects that have moved. The effect is that every time the store is garbage collected all the objects in the virtual memory are paged in, and if real memory is tight this may involve a great deal of paging activity. The solution is to garbage collect only the part of the heap which is being actively used. The basic idea was used by Cardelli in his VAX ML compiler[4].

His scheme was primarily intended to reduce the cost of garbage-collection. He pointed out that objects varied in lifetime so that many newly created objects become free very quickly while others might survive several garbage-collections before becoming free. In general the longer an object survived the lower the chance that it would be recovered during any particular garbage-collection.

The heap was divided up into regions by *age*. Initially objects were created in the youngest area, and as they survived more garbage-collections they migrated to the older areas. The younger regions were garbage-collected frequently and the older areas less frequently.

When a region is garbage-collected all the addresses into it must be identified in order to mark the objects and then they must later be updated after the objects have been compacted. If this required searching the whole of the heap to find the addresses there would be no advantage in garbage-collecting only a single region, but it is possible to restrict the search to only part of the heap.

Ignoring for the moment updatable values, (*references* in ML or *variables* and *arrays* in Poly), this is not too difficult. Whenever a new object is created it can only refer to objects that already exist. This means that an object in a particular region can only contain references to objects in the same region or an older one. A particular region can be garbage-collected by looking for references to it from the younger regions only, the older regions need not be touched. Using this method the garbage collector does not need to look at the older objects and the virtual memory activity is much reduced.

So far we have ignored the problem of updatable values. If a variable can be assigned to after it has been created then it is possible for it to refer to an object newer than itself, and if the variable is in an older region and the object it refers to is in a newer then the variable would have to be changed if the compaction phase moved the object. To avoid this problem updatable objects are put into a separate area of memory, the *local mutable area*, which is scanned during the marking and updating phases to find addresses every time any region is garbage collected. Because updatable values are relatively uncommon in Poly and ML this extra cost is quite small. The *local mutable area* can only itself be garbage-collected when the whole of the store is garbage-collected.

The actual implementation divides the heap up dynamically rather than having static regions. The heap grows downwards from the top of store with the topmost area used as the *local mutable area.* Below this is the oldest region with newer regions lower down the heap. The lowest area, immediately above the persistent store region, is used for newly created objects. New objects, whether updatable or not, are created in this region, but updatable objects are marked with the *mutable* bit set. When garbage-collection is necessary the newly created objects, and possibly some of the other regions, are together garbage-collected using the scheme described before. The result is a single region containing the youngest objects. During the compaction phase any updatable objects are moved into the *local mutable area.* Periodically the whole of the store is garbage-collected, particularly when more store is obtained from the operating system, since this is allocated at the top of the store.

Figure 8 shows the local area of the heap immediately before a garbage-collection. Two regions, "Old" and "Young" are shown. Figure 9 shows the area after the newly allocated objects have been collected into a new region, "Younger". The mutable area has increased in size slightly as newly allocated mutable objects have been copied into it. Figure 10 shows an alternative partial collection in which "Young" has been included in the collection. Again the newly allocated mutable objects have been added to the mutable area. Finally Figure 10 shows the state after a complete garbage-collection. The immutable objects have been compacted into a single "Old" area, and the mutable area, which has not been garbage-collected since the last full collection, has now been reduced in size as short-lived mutable objects have been collected.

As well as *references* or *variables* it is possible for an object to refer to objects newer than itself. This can occur in the case of, for example, the closures of mutually recursive procedures, where each closure may refer to the other. These will only cause problems if the store is garbage-collected after one closure has been created but before it has been filled in with the reference to the other. To avoid these problems a newly created object is marked as updatable and then this mark is removed when the object has been filled in.

In general all the words in a newly allocated object are filled in with values immediately after it is created. However in some cases, for example with closures of mutually recursive procedures, the garbage-collector could be called before the whole of an object has been filled in. For this reason after each garbage-collection

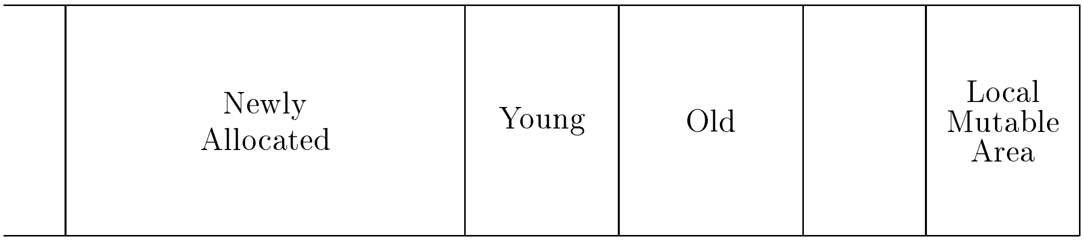

Figure 8: Before Collection

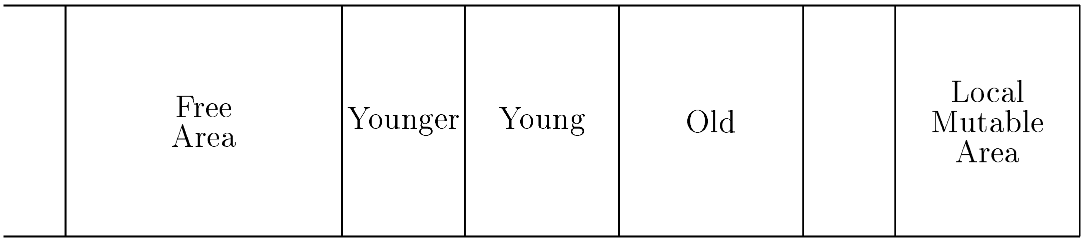

Figure 9: After Partial Garbage Collection (1)

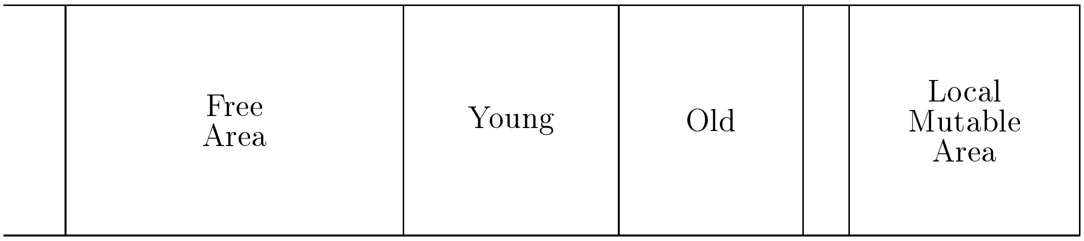

Figure 10: After Partial Garbage Collection (2)

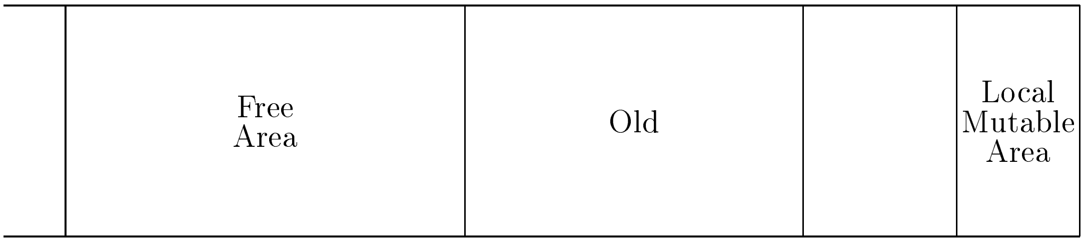

Figure 11: After Full Garbage Collection

the free store is cleared so that unset words are always zero. I found that clearing the whole of the free area immediately after a garbage-collection caused a great deal of virtual memory activity as pages were read in to be cleared and then written out again. One solution would have been to clear each object when it was allocated but this would make the allocation process more complicated, and since the code for allocating an object is compiled in-line it would increase the size of the code. Instead a compromise was found whereby only part of the free area is cleared and the garbage-collector is called when that area is used up. If there is more space available the garbage-collector simply clears some more space and returns, and only actually garbage-collects if the space is used up. This scheme reduces the paging activity since the pages that are cleared are likely to be used for new objects fairly quickly.

9 Poly/C Interface

The Poly code must be able to call functions which provide an interface to the outside world. It would be possible to insert code to do system calls directly in the Poly code using some sort of escape mechanism but it is better to isolate the operating system dependencies in a separate run-time system, leaving the Poly code independent of the operating system.

The run-time system is largely written in C, making operating system calls fairly simple. Rather than have the Poly code call these directly it indirects through the **interface map**. In this way the addresses of run-time system routines are not bound directly into the Poly code so changes can be made to the run-time system without requiring the Poly system to be reconstructed. The entries in the vector have special persistent addresses and the persistent storage system maps these onto vector entries when objects are read in. Indirecting through the vector actually works very well because the normal function call is done by an indirect call through the first word of a closure. In this way calling a function in the run-time system is exactly the same as calling any other function.

Functions written in C cannot be called directly from Poly code for a number of reasons. Instead the entries in the interface map point to pieces of assembly code which move the arguments from the stack of the current process onto the stack used by C procedures. Arguments which are addresses are put into a special area known to the garbage collector and the arguments to the C procedures are pointers into that. This allows the garbage collector to modify these addresses without having to search for Poly addresses among the variables used by the C code. If the C code needs to allocate new objects on the heap to pass back to the Poly code it stores the address in this area and uses it by means of indirect references.

10 Conclusions

The Poly run-time system has evolved significantly over a number of years and grown in size and complexity as new facilities have been added. The garbage-collector in particular is much more complicated than it was originally, but for a functional language this is a critical component of the system. The importance of paging activity on the garbage collector was unexpected. The low level object formats have changed very little since the original system, though stack segments were added recently along with the rest of the process mechanism.

References

[1] Robin Milner, *The Standard ML Core Language*, Edinburgh University.

[2] David C.J. Matthews, *The Poly Manual*, SIGPLAN Notices, September 1985.

[3] David C.J. Matthews, *A Persistent Storage System for Poly and ML*, Technical report 102, University of Cambridge Computer Laboratory, January 1987.

[4] Luca Cardelli, *The Functional Abstract Machine*, Computing Science Technical Report No. 107 AT&T Bell Laboratories.