A Runtime System
(DRAFT)

Andrew W. Appel*
Princeton University

February 1989

Abstract

The runtime data structures of the Standard ML of New Jersey compiler are simple yet general. As a result, code generators are easy to implement, programs execute quickly, garbage collectors are easy to implement and work efficiently, and a variety of runtime facilities can be provided with ease.

1 Introduction

Some languages, like Lisp, Smalltalk, ML, Prolog, etc. rely heavily on a *runtime system* to provide essential services. The most important such service is the management of dynamically-allocated storage (e.g. **garbage collection**), but the runtime system may also provide facilities like

- **Stream input/output:** on operating systems (like Unix) that do not have a buffered input/output facility, the process must provide its own; this might be handled in the runtime system.
- **Structured input/output:** the ability to automatically write a large linked data structure to a file, and read it back in with all links adjusted, is a great convenience that can be implemented efficiently in the runtime system.
- **Process suspension:** a snapshot of an executing process may be "preserved" in a file, so that the execution of that file causes a new process to start exactly where the saved one left off.

*Supported in part by NSF Grants DCR-8603453 and CCR-880-6121

- **Operating system calls:** operating system services needed by a program may be conveniently packaged by the runtime system.
- **Handling of interrupts and asynchronous events:** if the programming language has a mechanism to handle asynchronous events, it relies on the runtime system for its implementation.
- **Handling of arithmetic exceptions:** the programming language's exception-handling mechanism must be implemented in cooperation with the runtime system.
- **Assembly-language implementation of language primitives:** it may be inconvenient for the compiler to generate code for some features of the programming language; these functions can be implemented as calls to runtime system routines.
- **Foreign-language procedure calls:** calls to subroutines written in other languages may be mediated by the runtime system.
- **Fun with continuations:** Languages with features like *call with current continuation*, which allows the explicit manipulation of threads of control, require runtime-system cooperation.
- **Execution profiling:** Automatic measurement of the time spent in different parts of the user program can be accomplished with the help of the runtime system.
- **Debugging:** starting, stopping, and displaying the execution state of user programs can be accomplished only by low-level routines.

Since this list is rather long, and several of these features may interact, it is evident that runtime systems can become nasty and complicated. The proliferation of data types may make the implementation of the garbage collector (and other programs that must traverse the data) inefficient.

We particularly wanted a runtime data layout that provided fast allocation of records and fast garbage collection, since ML makes very heavy use of dynamically-allocated storage. To this end, we eliminated the runtime stack and use a very simple data format. The only runtime data types are *integers*, *pointers*, *records*, and *strings*. Then we found that this very simple structure allowed the easy implementation of many of the services described above.

Our runtime system has relatively few ML-specific features; it could be used equally well for other languages. This paper describes the design and implementation of the SML-NJ runtime system. It should be read in conjunction with an earlier paper on the SML-NJ generational garbage collector[3].

Text in smaller type is of interest only to those who might have to read, modify, or maintain the source code to the runtime system.

2 Standard ML of New Jersey

The ML language originated as part of the Edinburgh LCF proof system [14], and was soon implemented as a stand-alone compiler [10][11]. The language was "standardized" [21][20] [22][16], and Standard ML has been implemented at Edinburgh, Cambridge, and New Jersey[8]; the New Jersey implementation is a joint effort between researchers at AT&T Bell Laboratories and Princeton University.

Though ML was first implemented as the meta-language of a theorem-proving system, Standard ML is a general-purpose programming language with several advantages over more conventional languages. Its important characteristics are:

- **Automatic garbage collection:** this is a great convenience in writing correct and readable programs.
- **Static, polymorphic types:** like Pascal, types are checked at compile-time and not at runtime; but like Lisp, there is great flexibility and re-usability of code.
- **Safety:** there are no runtime insecurities (i.e. "core is never dumped"); this is unlike the C language, where unsafe pointers run rampant, and like Lisp (except when Lisp programmers turn off the runtime type checking for efficiency). In ML, there is no run-time type checking, but safety is guarateed by compile-time type checking.
- **Higher-order functions:** like Scheme (and lambda-calculus); this can lead to a desirable conciseness of expression.
- **Typechecked Modules:** like Ada and Modula.
- **Exception handling:** a dynamically-scoped exception mechanism allows both hardware- and software-generated exceptions to be caught and handled by user programs.

One important feature of a systems programming language is not prescribed by the list above, and in fact cannot be specified by a language definition: an efficient and robust implementation. Standard ML of New Jersey[8] is meant to be a complete, efficient, robust, and cleanly written compiler for the language. It has an optimizing code generator based on continuation-passing style [2].

A knowledge of ML is not necessary to understand its runtime system.

3 Tagging schemes

Almost all the pieces of the runtime system must deal with the data structures of the executing program. Therefore, it is helpful to keep the format of this data as simple and straightforward as possible.

ML, like Lisp, allows polymorphic functions: a function that reverses a list of objects (for example) need not know the type of the objects. In order that the same piece of executable code can operate on objects of arbitrary type, it is necessary that every object be represented in the same amount of space. As in Lisp, this is achieved by making everything be the size of a pointer; if an object's natural representation is larger (as for a record of n objects), then it is represented by a pointer to storage on the heap.

The garbage collector must be able to determine the size and layout of each object it traverses. This can be accomplished in several ways:

- **By encoding a type-tag inside each pointer;** we chose not to do this because it reduces the number of bits left for actual addressing. In these days of 32-bit pointers and 100-megabyte memories, it is easy to imagine the need for every bit of addressability we can muster.

- **By reserving different areas of memory for objects of different types.** BIBOP (Big Bag Of Pages) schemes require that each page hold objects of a single type, and a global table maps pages to types. This is relatively efficient, and doesn't reduce the number of addressable words. But it complicates the process of allocating and copying objects, since several free regions are simultaneously required.

- **Statically-typed languages don't require any runtime-tagging** at all; instead, the compiler can tell the garbage collectector about the type system of the program. This has worked well in Pascal[9]. Even though it is also theoretically possible in ML, in practice the polymorphic type system introduces overhead and complexity that make this method unattractive[4].

- **Each record can have a descriptor at the beginning** that explains which fields are integers and which are pointers. This method works well in non-polymorphic languages like Modula-2+ and Mesa, where descriptors can be computed at compile-time and just inserted in records as they are created. However, in polymorphic languages like Lisp and ML this descriptor would have to be laboriously constructed each time a record is created, introducing unacceptable overhead.

- **Each record can have a descriptor at the beginning** that tells the length of the record, and **each field can have a tag bit** that tells whether it is a pointer or a non-pointer. This is what we have done.

To have a tag word on each record, and a tag bit on each field, is not clever at all; but we are not always striving for cleverness, we want simplicity and efficiency. If all our records were *cons* cells, then one-third of memory would be devoted to tag words; but memories are large and cheap. And in Standard ML, records are of many different sizes, and two-word records are not particularly predominant.

By having only one free region, we are able to do fast allocation and also manage the total amount of virtual memory used[3]; this would be much more difficult with a BIBOP scheme.

4 Data formats in our runtime system

Standard ML of New Jersey has records, strings, procedures (machine code), closures, constructors, arrays, byte-arrays, floating-point numbers, references (modifiable one-word records), modules, and integers. This large set of language primitives and user-defined datatypes are represented by just two runtime data formats: one for objects that contain pointers (records, closures, constructors, arrays, references, modules), and the other for objects containing no pointers (strings, procedures, byte-arrays, floating point). The garbage collector (and other parts of the runtime system) need to understand only these two formats, not the many kinds of ML objects.

A *field* is either a *pointer* or an *integer*. Pointers have a low-order bit of 0; integers have a low-order bit of 1. It is necessary to distinguish pointers from non-pointers in order that the garbage collector will know what structures to traverse. On a byte-addressable machine, all pointers to aligned (4-byte) words are multiples of 4 anyway, so pointers have a low-order bit of 0 in their natural representation. The high-order 31 bits of an integer field can represent an integer in the range $[-2^{30}, +2^{30} - 1]$.

Some Lisp implementations use a similar representation except that pointers have a low-order 1 and integers have a low-order 0. This makes arithmetic on tagged integers somewhat easier, and takes advantage of the fact that pointers are usually used with a displacement addressing mode; the displacement can be adjusted by 1 with no penalty in efficiency. This is probably a better arrangement, but either version of this scheme will work; and a simple analysis[4] shows that the efficiency trade-offs are negligible, and that doing arithmetic around the low-order integer tag is not very costly.

An *object on the heap* may be either a *record* (containing fields, i.e. pointers and tagged integers) or a *string* (containing bytes of an arbitrary bit-pattern, but no pointers).

A *record* is a sequence of $n > 0$ fields numbered $0, 1, ..., n-1$. Each record has a *descriptor* at position -1. The low-order bit of a descriptor is 1 (making it look like an integer), the next three bits identify the object as a record, and the high-order 28 bits give the number of fields. Thus, each record is limited to one gigabyte, which should not be a significant limitation (seriously, though, it is important to avoid miserly restrictions on the sizes of objects).

In the implementation of closures, it is useful to be able to point at the interior of records. However, it is always necessary for the garbage collector, given a pointer, to find the descriptor of an object. If the fields 0 through $k-1$ of a record are all pointers, then a pointer can point at the k^{th} field of the record; since the descriptor is unboxed (has a low-order bit of 1) and the fields 0 through $k-1$ will all be boxed, the garbage collector can easily search backward for the descriptor of the record. So, we allow pointers to the interior of a record, providing all the previous fields are boxed; this is sufficiently flexible for our needs in implementing closures, as will be described. Otherwise, no pointer may address the interior of a record.

A *string* contains $n > 0$ bytes (numbered from 0 to $n-1$), padded with trailing zero bytes to a multiple of four. Immediately preceding the 0^{th} byte is a one-word descriptor whose low-order bit is 1, whose next three bits describe the type of string (and distinguish strings from records), and whose upper 28 bits give the length in bytes.

Strings are used for a variety of purposes; they can hold printable characters, real numbers, machine code, or any other kind of data that doesn't contain pointers. In fact, a string object can hold several embedded string objects. A pointer may point at the 0^{th} byte of a string, and may point to an embedded string at the k^{th} byte provided that $k > 0$ is a multiple of 4 and that the bytes $k-4, k-3, k-2, k-1$ contain a *back-pointer* descriptor that contains the number k (the offset to the beginning of the record). This allows the beginning of a string object to be found quickly, given a pointer into it. Note that a back-pointer couldn't be distinguished from data, except for the fact that it is found just before the pointed-to byte of the string.

The three tag bits of a descriptor can denote any of these kinds of records, strings, etc:

0 record

1 forwarding-pointer (for the garbage collector)

2 back-pointer (that precedes an referenceable location in a string)

3 embedded string length (described below)

4 array (just like a record).

5 byte-array (just like a string).

6 this tag value is unused.

7 string

Because of ML's static type system, it is not necessary to put type information in the descriptors of objects. Therefore, tags are necessary only for the garbage collector's benefit, since it must distinguish objects of different formats.

The only exception to this rule is that mutable objects (arrays, byte-arrays) must be distinguishable from immutable objects (records, strings) in order to implement the polymorphic equality feature. For all purposes of the runtime system, arrays are identical to records, and byte-arrays are identical to strings.

A dynamically-typed language like Lisp would require more than three bits to distinguish types of objects; this would not pose a serious problem. It would still be possible in Lisp to have just two formats (pointer-containing and pointer-free), as in our runtime system.

5 Forwarding pointers

The Standard ML of New Jersey runtime system has a generational garbage collector that takes advantage of object lifetime and referencing patterns[3]. But at the heart of any generational garbage collector is a simple copying garbage collector as originally described by Cheney[12]. Objects are copied from *fromspace* to *tospace* in a breadth-first order, with the *tospace* itself serving as the queue for the breadth-first traversal. The *fromspace* versions of objects are overwritten with *forwarding pointers*, so that when other references to them are found, it is easy to find the *tospace* copies of them.

The fundamental operation in Cheney's algorithm is to *forward* a pointer. This means taking a pointer into *fromspace* and making it point to *tospace*. If the *fromspace* object it points to has already been copied, then its forwarding pointer is taken as the new value; otherwise, the object must be copied to *tospace* and a forwarding pointer installed.

Forwarding is relatively easy using our runtime data format. We have a special kind of descriptor **forwarding-pointer**, that indicates that a *fromspace* object has already been copied. If an object has this descriptor, then the first word (after the descriptor) is to be interpreted as the address of the copy. The only complications in forwarding are that pointers may point into the middle of records and strings.

Our runtime system (and garbage collector) is implemented in C. In figures 5, 5, 6 we show the entire core of our copying garbage collector. We could use pseudo-code and describe the algorithm abstractly, but we want to emphasize that our simple runtime data format does permit an efficient and easy-to-implement garbage collector.

We pretend that all ML values are integers, and make liberal use of casts to maintain this pretense. Our `forward` function is shown in figure 5; it takes an ML value (by reference) and modifies it (if a pointer into *fromspace*) to point into *tospace*. Line 2 establishes m as a copy of the original value to be forwarded. Line 3 considers only the case that m is a pointer, and points into *fromspace*. (Any pointers not into *fromspace* are treated as constants.) Line 4 adjusts m to point at the descriptor of the *fromspace* object.

Lines 5–38 loop until the beginning of the object is found. If [line 8] m points to a back-pointer, the appropriate offset is subtracted and we start again; similarly, if [lines 35,36] m points to a pointer (which can happen if we point into the middle of a closure record) or an embedded descriptor, then m is decremented and we try again.

If m is a string or byte array [line 13], then we adjust its *len* to count in words rather than bytes (i.e. we divide by 4, rounding up). (Lines 14–18 are explained below.)

Now at line 22 we have either a string or record in *fromspace* that needs to be copied. We verify that there's enough space remaining in *tospace*; if not, we call the function `gmore` that will either get more space or die trying. Then we copy all the words of the object. Finally [lines 27 and 30] we install a forwarding pointer into the *fromspace* object and mark its descriptor as forwarded.

For an already forwarded object (line 33), whether it was forwarded in a previous call or just now, adjust the reference `*refloc` to point at the new object. Since the reference might have pointed at the middle of the old object, we must take care to make it point to the corresponding location in the new object; this is accomplished

```
forward(int *refloc)
{register int *m = *((int**)(refloc)), len;
 if( (m&1)==0 && (m >= (int*)lowest && m < (int*)highest))
 { m--;  /* make m point at the descriptor */
   while(1)
     {len = (*m)>>4;
      switch(m&15)
       {case tag_backptr:
                m -= len;
                continue;
        case tag_string:
        case tag_bytearray:
            len = (len+3)/4;

            /* fall through */
        case tag_record:
        case tag_array:
            {int **i=(int**)m, **j=to_ptr;
                 while (j+len >= to_lim)
                   do to_lim=gmore();
                 while (len-- >= 0)
                   do {*j++ = *i++;}
                 ((int**)m)[1]= 1+(int*)to_ptr;
                 to_ptr = j;
            }
            (*m) = tag_forwarded;
            /* fall through */
        case tag_forwarded:
            *(int*)(refloc) += ((int*)m)[1] - ((int)(m+1));
            return;
        case tag_embedded:
        default:
                m--; continue;
}}   }  }
```

Figure 1: The `forward` function

by the computation:

$$\text{old pointer} + \text{beginning of new object} - \text{beginning of old object}.$$

Then the `forward` function returns.

Almost without exception, all pointers to objects point to places where an immediately preceding descriptor explains the format of the object. The exception is an artifact of the very fast allocation mechanism that can create and initialize a k-word object in $k+2$ instructions[3]. This mechanism relies on a page fault trap to invoke the garbage collector when the free space is exhausted; when this trap occurs, the program counter will point into the middle of a string object that might be moved by the garbage collector. This is the only reference that points to a place in a string that lacks a backpointer. It can be handled relatively simply: as each string object is moved, we check to see if the saved program counter points into the middle of it; if so, we adjust the saved program counter. Since the number of string objects is typically less than one percent of the number of record objects, very costly overall. The test occurs at line 14, as shown in figure 5.

```
14              if (!trap_pc_done
15                   && m < trap_pc && m+len >= trap_pc)
16                 {trap_pc_done=1;
17                  trap_pc += to_ptr - (int**)m;
18                 }
```

Figure 2: Testing the (saved) program counter

6 Garbage collection

Once the `forward`ing procedure is written, a simple copying garbage collector is trivial (figure 6). The function `gc` is parametrized by the lower and upper bounds of *fromspace* and the lower and upper bounds of *tospace*. The `roots` parameter is a vector of pointers to the roots of accessible objects; in the Standard ML system this is little more than the addresses of saved machine registers. The last two parameters are the address of the saved program counter (necessary as explained in section 5) and the address of a function which can be called to expand the *tospace* if necessary.

The first step [lines 6–9] is to put various quantities into global variables to make them accessible to the `forward` procedure, since C does not have nested procedures.

```
gc(int *from_low, int *from_high,
   int *to_low, int *to_high,
   int **roots,
   int *trap_pcx, int *(*get_more)()
  )
{ gmore=get_more; trap_pc = *trap_pcx; to_ptr = to_low;
  trap_pc_done = !(trap_pc>=(int*)from_low
                && trap_pc<(int*)from_high);
  lowest=from_low; highest=from_high;

  while (*roots) forward(*roots++);

  { int *x = to_low;
  while (x<to_ptr)
  { int *p = x+1, descr = (*x), len=descr>>4;
    if (string_or_bytearray(descr))
          x += (len+3)/4 + 1;
    else {x += len+1;
          do {forward(p++);} while (p<x);
  }      }

  *trap_pcx = trap_pc;
}
```

Figure 3: The gc function

The next step is to forward all the root variables (line 10).

Finally [lines 11-18], we forward each word of each record in *tospace*. Since the `forward` procedure may increment the `to_ptr` variable that denotes the end of the filled portion of *tospace*, we have to keep iterating until `x` catches up with it. In effect, the *tospace* between `x` and `to_ptr` is the queue of the breadth-first search. The queue must eventually become empty, since there is a finite amount of accessible data to be copied. In this phase, integers (and strings) are just skipped, since they contain no pointers that need forwarding.

This garbage collector is relatively simple, and therefore it's not difficult to make it fast. Even the fanciest of generational or concurrent collectors relies on a program like this as its inner loop; the very simple layout of data in the Standard ML runtime system makes efficiency easy.

7 Fast allocation

Copying garbage collection, because it takes time proportional only to the live data and not to the amount of garbage, can be arbitrarily efficient[1]. That is, there is no lower bound on the amortized cost of garbage collection for each cell allocated. Standard ML's generational copying garbage collector[3] expends on the order of one (amortized) machine instruction for every cell allocated; the precise amortized cost is proportional to the ratio of live data to heap size.

Since garbage collection is so fast, it makes sense to make allocation fast too. Standard ML of New Jersey allocates a new record every 40 to 80 machine instructions, so we want a very low overhead on the creation of an object.

Since copying garbage collectors compact the live objects into consecutive memory cells, the free region is all contiguous. This means that to create an n-word object, we can just grab the next n words of the free space. Since objects are created so often, it makes sense to make allocation in-line (with no procedure call) and to reserve a register `fr` to point to the beginning of the free region. Then it becomes very simple to allocate a new object: `A = CONS(B,C)` is implement as

1. mem[fr+2] := C
2. mem[fr+1] := B
3. mem[fr] := descriptor(2,tag_record)
4. A := fr+1
5. fr := fr+3

Each of these lines is one machine instruction. Thus, in five instructions (plus one instruction of amortized garbage collection overhead) we have made a new *cons* cell; it takes only twice as long to create a data structure as it does to read it! This fast allocation encourages a simple and clean programming style; no longer do programmers have to stand on their heads to avoid *cons*ing.

Of course, this won't work if the free region is exhausted. We can insert an explicit test to make sure that `fr` is not near the end of the free region. But a more clever trick is to make the virtual memory page at the end of the free region inaccessible, so that we will get a page fault when the free region is exhausted. That's why we store the last field (`mem[fr+2]`) first in the example above; it's simpler for the garbage collector if the fault occurs at the very beginning of creating the new cell. (On the MC68020, the state of the machine at a page fault is complicated, and it's not easy to restart the faulting instruction; so on that machine we use an explicit comparison with a limit register.)

The page fault will be caught by the hardware and handed to the operating system, which can then pass control to the user process. The user process then has to find all the registers of the faulting procedure; these registers are the roots of the accessible data. Appel[3] and Cormack[13] both describe schemes for finding these registers; Cormack's is simpler and more reliable.

The ML program and the garbage collector behave like two processes (threads) running in the same address space: while one thread executes, the other's registers are saved. When ML suspends itself (either because of a page fault indicating end of free space, or voluntarily), it saves its registers into a static area that looks just like a process control block, so that the garbage collector (and other parts of the runtime system) can access them.

> Here's how Cormack's scheme works: A page fault arrives, causing the operating system to invoke the C function `ghandle`. This function is passed (as an argument) a structure containing the address of the faulting instruction. Other saved registers are at undocumented locations on the stack. `ghandle` saves the faulting pc in `saved_pc`, and modifies its argument structure to point to the assembly-language function `saveregs`. Then it returns; the operating system restores the pc from the argument structure, restores other registers (from those undocumented locations), and resumes. But of course, we have fooled the operating system into resuming in `saveregs`, which stores all registers into known global variables and returns to the C thread (i.e. in the function `runML`).
>
> The C thread (typically) does garbage collection, then calls the assembly-language function `restoreregs`, which loads registers (and program counter) from their global variables and resumes execution of the ML thread.
>
> Sometimes it's useful to invoke the C thread without a page fault, e.g. to export the process state into a file or to do a structured write. In this case,

`saveregs` can be called directly from assembly code in the ML thread.

8 Heap allocation of procedure call frames

Since heap allocation is so cheap, we put procedure call frames on the heap instead of on the stack. This has many advantages[2], but the one of interest here is that it greatly simplifies the runtime system. Many of the facilities described in this paper would be much more complicated to implement if runtime stacks had to be dealt with.

Any record on the stack must be initialized as it is created; otherwise it will contain garbage data that could be interpreted by the garbage collector as spurious pointers. However, typical code generators will allocate a call frame on entry to a procedure and spill registers into it as needed. This must be avoided. One way to solve this problem is to delay allocation of the frame; values can be accumulated in registers until spilling is necessary, then all the registers can be spilled at once into a newly-created frame, which is just a record object in runtime data format. Thus, the frame is completely initialized as it is created.

The Standard ML of New Jersey compiler doesn't use "procedure call frames." Since it uses continuation-passing style[24][18][2], what an ordinary mortal might call a "frame" is really just the closure of a continuation. It is easy to create closures as ordinary records that are completely initialized when they are created. This simplifies both the code generator and the runtime system, though the trick described in the previous paragraph will work for more conventional code generators.

Since there's no runtime stack, the *call-with-current-continuation*[23] primitive can be implemented very efficiently. In implementations with a runtime stack, the entire stack must be copied when *call/cc* is evaluated (or else there must be a lot of extra complexity in stack management); without a stack, the execution of *call/cc*, and the execution of saved continuations, take just a few instructions each. This makes *call/cc* a practical programming tool, just as fast allocation makes *cons* practical.

9 Representing ML structures in records and strings

Section 4 describes just two kinds of objects—records and strings—referenceable at their beginning and (in a limited way) at interior points. All of the kinds of ML data can be represented in records and strings.

An ML value must be representable in one word. A larger value can be *boxed* by putting it in several words on the heap and keeping a (one-word) pointer to it. A small value (like a 31-bit integer) can be kept *unboxed* by representing it without indirection in a machine word. Boxed values (pointers) are distinguished from unboxed values (integers, or data represented as integers) by their low-order bit.

An ML **record** is an n-tuple of values. It has a natural representation as a record in our runtime data format. ML has a *pro forma* record of length 0; as our runtime data format does not allow objects of length 0 (since that would leave no room to put a forwarding pointer, as described in a section 5), the empty record is represented as the unboxed integer 0. This does no harm, since no fields can be selected from an empty record anyway.

An ML **array** is also an n-tuple of values. In the ML language, record-field offsets are determined at compile-time, whereas arrays may be indexed by runtime values; and arrays may be modified after they are created, whereas records are immutable. But neither of these differences matters to the runtime system. The polymorphic equality function of ML requires distinguishing between mutable and immutable objects at runtime, so arrays and records have different tags, however; this requires that they have different tags.

ML does permit arrays of length 0, but our runtime data format does not. Happily, all arrays of length 0 have the same behaviour, so a special object of length 0 (located outside the garbage-collectible region) serves to represent all the empty arrays.

References in ML are mutable objects: `val a = ref 5` declares a reference variable `a` that may be changed by a later assignment statement, unlike most variables which cannot be modified once defined. Reference variables behave just like single-element arrays, and that's how they are represented in the runtime system.

ML has **datatype** declarations to allow tagged variant types. For example, the declaration

```
datatype t = NAME of string | NUMBER of int
```

specifies a type `t` that can be either a string or an integer, depending on whether the constructor `NAME` or `NUMBER` is used to create it. The program can examine any object of type `t` and determine whether it has the `NAME` or `NUMBER` representation—that is, it is a *tagged* union. An object of type `t` is represented as a two-element record: one field contains the tag (a small integer), and the other field contains the value. The details of constructor representation may be of interest only to those knowledgeable in ML, and are described in section 15.

ML has **strings** of characters. Strings can be of any non-negative length. Since our runtime representation cannot support objects of length 0 (because one word is needed to store the forwarding pointer, as described in a section 5), the empty string must be treated specially. However, it turns out that it is never necessary to create a new empty string; the only occurrences of empty strings are as literals in the program text. String literals (and back-pointers) will be discussed in section 11.

Strings of length one are treated specially by the compiler, though this is not necessary either for the ML language or for our runtime data format. Single-character strings are treated as unboxed integers between 0 and 255, to avoid heap-allocation for this (frequent) special case. Though this technique may cause less allocation, it requires special tests on every string operation. It's not clear whether this special case saves more than it costs. However, this special case is transparent to the runtime system anyway; all conversions, etc. are handled explicitly by the compiler; the single-character strings are not a new runtime data format, but look like ordinary unboxed integers.

Byte-arrays in ML are to strings as arrays are to records: they have the same representation as strings, but their tag is different to facilitate certain language features.

Floating-point numbers are too large to fit in one word. Even if we chose to use single-precision floating point, it would be difficult to store them unboxed because there is no bit available for use as a tag bit. Thus, all floating-point numbers are stored boxed, with 8 bytes of data and one word of descriptor. The descriptor must be one that indicates that the contents of the object contain no pointers. Since ML is statically typed, the language never needs to distinguish (at runtime) floating-point numbers from strings, so we can just represent a double-precision float as an 8-byte string. This is not an ASCII string, it is the hardware representation; a "string" descriptor does not denote that the contents are printable characters, just that there is an arbitrary bit-pattern not containing pointers.

Similarly, **machine-code procedures** do not require a separate class of descriptor. From the garbage collector's point of view, machine code is just like strings: it contains bits that are to be (perhaps) moved from place to place but which never represent pointers. Thus, machine code is just kept in string objects. The mechanism by which we avoid placing pointers (or any reference to other objects) in machine code will be described in section 12.

10 Closures

In ML, as in many lexically scoped languages, functions may have free variables. That is, the body of a function may reference not only its own formal parameters (the bound variables of the function), but the formal parameters of a statically enclosing function. Consider the function

```
fun add(x) = let fun add1(y) = x+y
              in add1
             end
```

The function `add(x)` defines an internal function `add1(y)` that adds `x` and `y`, and then `add` returns `add1` as its result. The variable `x` is bound by `add`, and `y` is bound by `add1`. Therefore, if we just consider the function definition `fun add1(y) = x+y`, the variable `x` is a free variable of this internal function.

In order to evaluate `add1` applied to some argument, we must have a context in which the value `x` is defined. It will not suffice for the `add` function just to return a pointer to the machine code of `add1` as its result. Instead, it must return a *closure*: a combination of a code-pointer an some information about the values of all the free variables (in this case, just `x`).

For example, the result of evaluating `add(5)` is (abstractly) the function which will always add 5 to its argument. Figure 10 shows a possible representation as a record object, with a descriptor at the beginning for the garbage collector. The

```
     +------+
     | desc |
     +------+
p--> |  o------------> machine code for add1
     +------+
     |  5   |
     +------+
```

Figure 4: A simple closure.

pointer `p` is the runtime representation of the particular version of `add1` in which `x` is bound to 5. This pointer may be passed as an argument to some other function that doesn't know anything about the `add1` function. All the other function has to know is how to "call" a closure. The calling sequence might be:

1. Put `p` in register 0.
2. Put the return address in register 1.
3. Fetch p[0] into register 2.
4. Jump to the address contained in register 2.

Step 1 is necessary so that when add1 is entered, it can gain access to the closure record where the binding of x is stored. Step 2 is just the normal saving of a return address, which might in many cases be on a stack instead of in a register. Steps 3 and 4 are just the typical code necessary to jump to a runtime-bound address.

It is important to note that the function that calls a closure p doesn't need to access any part of it except the machine-code pointer in p[0]. This means that the free variables (the rest of the closure) may have an arbitrary arrangement, known only to the function that builds the closure (add, in this case) and the function that executes it (add1, in this case). Closures can be structured in many ways[7], and can even point to other closures for access to free variables (i.e., they can have "static links," in Algol terminology).

Clearly, closures are represented a lot like records. A closure has several fields, each of which is either a machine-code pointer or a free variable. The machine-code pointer is just a (boxed) string pointer, and the free variables are just one-word ML objects. Therefore, we use record objects to represent closures.

When several functions share a set of free variables, it is convenient to have a **multiple-function closure.** Consider the functions

```
fun test(y,z) =
    let fun even(i) = if i=0 then y else odd(i-1)
        and odd(i)  = if i=0 then z else even(i-1)
     in (even,odd)
    end
```

This function, when applied to arguments (3,7), would return two functions (let's call them even37 and odd37). from integer to integer. Even37 applied to even integers returns 3 and applied to odd integers returns 7, and odd37 works the other way around.

We could represent these functions by two different closures, as shown in figure 10. This is particularly unfortunate because even and odd are free variables of each other, and this requires the construction of a cyclic data structure of closures, which can get complicated.

A more clever trick[8][18] is to let these two functions share a closure, as shown in figure 10. Now, when even37 is called, its closure-pointer will be loaded in register 0, and it knows that it can access y at offset 2 from register 0. And when odd37 is called, its closure pointer (which is really a pointer to the second field of the record) will be loaded in register 0, and it can access z at offset 2. In some situations, the same free variable may be accessed by different functions in the closure, and the code generator will have to remember that the offsets from register 0 to a particular field

```
            +------+
            | desc |
            +------+
even37 -->  |  o------------> machine code for even
            +------+
         y  |  3   |
            +------+
       odd  |  o-------> odd37
            +------+

            +------+
            | desc |
            +------+
 odd37 -->  |  o------------> machine code for odd
            +------+
         z  |  7   |
            +------+
      even  |  o--------> even37
            +------+
```

Figure 5: Two mutually recursive closures

```
            +------+
            | desc |
            +------+
even37 -->  |  o------------> machine code for even
            +------+
 odd37 -->  |  o------------> machine code for odd
            +------+
         y  |  3   |
            +------+
         z  |  7   |
            +------+
```

Figure 6: Two functions sharing a closure

depend on which closure-pointer is loaded (i.e. which function we are generating code for).

Finally, if `even` wants to call `odd`, it can just generate the appropriate closure value in register 0 by adding an offset of 1 to its own closure pointer. In general, all mutually recursive functions can be handled this way, and cyclic closure structures are never required. But even non-mutually-recursive sets of functions can save storage by sharing closures.

It should now be clear why (in section 4) we arranged for pointers into the middle of record objects. Given the pointer `odd37`, the garbage collector can easily search backward until it finds the descriptor of the record; the descriptor is unboxed, while all of the previous machine-code pointers are boxed fields.

11 Function entry points

A typical compilation unit will contain many functions, and it would be unwieldy to make a different string object for each function: since the functions in a compilation unit often call each other, these calls will go much faster as pc-relative jumps than if we had to fetch addresses from closures. But to achieve pc-relative jumps, the relative distance between two functions cannot change; and the garbage collector moves objects around. So to achieve this constant distance, we must put several functions in the same string object, and have several entry points.

For example, the `even`, `odd`, and `test` functions of section 10 could all be placed a single string, with appropriate back-pointer descriptors (figure 11). The numbers `[5]` and `[10]` are back-pointers, formatted as described in section 4. They tell the garbage collector the offset to the beginning of the object, so it can find the true descriptor.

The advantage to putting several functions in the same string is that they can refer to each other directly, without having to access each other through closures. Of course, it will be necessary to adjust the closure pointer in register 0. Here, `even` could call `odd` by this (simpler) calling sequence:

1. Add 1 to register 0 (to adjust p).
2. Put the return address in register 1.
3. Jump to `odd`.

The garbage collector may move a code string from one place to another in memory, so it is necessary that the jump in step 3 be a pc-relative jump.

```
          +------+
          | desc |
          +------+
test -->  |  t   |
          +  e   +
          |  s   |
          +  t   +
          |      |
          +------+
          | [5]  |
          +------+
even -->  |  e   |
          +  v   +
          |  e   |
          +  n   +
          |      |
          +------+
          | [10] |
          +------+
 odd -->  |  o   |
          +  d   +
          |  d   |
          +------+
```

Figure 7: Three code strings embedded in a string

ML programs may contain literal strings and floating point constants. These are embedded within string objects amongst the machine-code functions. For example,

```
fun show(b) = if b then "true" else "false"
```

could almost have the representation shown in figure 11. The function `show` returns
.

```
          +------+
          | desc |
          +------+
 show --> |  s   |
          +  h   +
          |  o   |
          +  w   +
          |      |
          +------+
          | [5]  |
          +------+
 true --> | true |
          +------+
          | [7]  |
          +------+
false --> | fals |
          +      +
          | e000 |
          +------+
```

Figure 8: String literals, oversimplified

a pointer to `true` or `false`; this pointer points into the middle of the large string object at one of the embedded strings. Because the embedded string is preceded by a back-pointer descriptor, the garbage collector won't be confused.

There's a slight problem with this. Although the garbage collector doesn't need to know the length of the string `"true"`, the ML program might apply the `length` function to it. String lengths are kept in the descriptor word, and are accessed by fetching field -1 and shifting off the tag bits. The problem here is that the descriptor for `true` is a back-pointer, not a string descriptor. The solution—simple and inelegant—is to introduce a new kind of descriptor, an *embedded string*, which contains the length of the string and always is preceded by a backpointer. Thus, the actual representation of the `show` function is more like figure 11.

```
          +------+
          | desc |
          +------+
 show --> |  s   |
          +  h   +
          |  o   |
          +  w   +
          |      |
          +------+
          | [6]  |
          +------+
          | 4emb |
          +------+
 true --> | true |
          +------+
          | [9]  |
          +------+
          | 5emb |
          +------+
false --> | fals |
          +      +
          | e000 |
          +------+
```

Figure 9: String literals embedded in a code string

The treatment of embedded floating point constants is just like that of embedded strings. However, the ML program will never need to take the length of a floating point constant, so the embedded-string descriptor can be omitted and a simple backpointer will suffice. Note that embedded strings of length 0 are permitted, since there is no need for a word to install a forwarding pointer (see section 5, and note that forwarding pointers go only at the beginning of the entire string object, not at embedded strings).

The address of string and floating constants may appear in machine code, but only in a pc-relative way, since code objects may be moved in memory by the garbage collector.

12 Modules and compilation units

Standard ML has a powerful module system, allowing nested modules, modules as parameters to other modules, thinning of a richer module to form a smaller module, etc. The module system can be represented entirely as records and functions in the runtime system[8]; no new machinery is needed.

A compilation unit is just a sequence of function and value declarations that will be compiled together into one string object. But one compilation unit may refer to values defined in another, and a linkage mechanism is required.

We can use the power of higher-order functions to implement the linkage in a way that is transparent to the runtime system. If unit `B` refers to a value in unit `A`, then the compiler will treat `A` as an implicit parameter of `B`. For example, suppose we have two compilation units

```
fun f(x,y) = x+y
```

and

```
fun g(z) = f(z,z)
```

clearly `f` is a free variable of `g`. But the compiler can parametrize the second unit, as if it were written

```
fun g0(f) = let fun g(z) = f(z,z)
             in g
            end
```

This compilation unit has no free variables; the ML system will simply apply the second unit to the first one, yielding the desired function `g`. The compiler must keep track of inter-module references in order to do this, but at least the runtime mechanism for linkage is simple.

So, each compilation unit is a closed function with no free variables. No special linkage mechanism is built into the runtime system; the closure mechanism handles linkage very elegantly.

Eliminating free variables from compilation units also simplifies the code generator. Whenever the code generator must analyze local free variables, generate pc-relative references, etc., its job is much simpler because there are no global references to other objects. The interface between the front end of the compiler and the code generator are much cleaner as a result.

13 Linkage to assembly language

Some primitives of a programming language are best implemented in a different language (typically assembly language). Standard ML of New Jersey makes use of 8 functions written in assembly language, and 8 functions written in C. (For comparison, more than 100 standard-library functions are written in ML.) The assembly-language functions are:

1. **array(n,x)** creates an array of length n, each element initialized to x.
2. **callc(f,a)** calls a C-language function f with argument a.
3. **create_b(n)** creates a byte-array of length n.
4. **create_s(n)** creates a string of length n.
5. **floor(x)** returns the smallest integer less than or equal to x.
6. **logb(x)** returns the exponent part of (the floating-point) x.
7. **scalb(x)** inserts a new exponent into (floating-point) x.
8. **syscall(i,args,k)** does operating system kernel-call i with k arguments.

The assembly-language functions are located at (constant) addresses within the runtime system. But it's a good idea to follow the rules about free variables in module-linkages: even the references to assembly-language primitives should not be "hard-wired" addresses in code objects. If this were done, then all ML code would have to be re-compiled whenever the runtime system was re-compiled.

Instead, we just treat the assembly-language functions as a special record object containing nine elements. Other modules are parametrized by this `Assembly` module just as if it were an ordinary one. All the assembly-language functions follow the ML calling conventions.

Most of the C functions are to provide access to system calls with hard-to-manage interfaces, like `fork`, etc. There is an added complication that the C and ML calling conventions don't match. The assembly-language function `callc` arranges the arguments for an ML function to call a C function. The details are uninteresting, but the point is that absolute references are again avoided.

> The file `boot/assembly.sig` contains the signature `Assembly`, approximately as in figure 13. The substructure `A` is the machine-dependent part (implemented in assembly language), and the other components (exceptions,

```
signature ASSEMBLY =
  sig
      datatype datalist = DATANIL | DATACONS of (string * string * datalist)
      type func
      datatype funclist = FUNC of (func * string * funclist)
      type object
      structure A : sig
        val array : int * 'a -> 'a array
        val callc : 'b (* func*)  * 'a -> 'c
        val create_b : int -> string
        val create_s : int -> string
        val floor : real -> int
        val logb : real -> int
        val scalb : real * int -> real
        val syscall : int * string list * int -> int
      end
      exception Div
      exception Float of string
      exception Interrupt
      exception Overflow
      exception SystemCall of string
      exception UnboundTable
      val array0 : 'a array
      val bytearray0 : string
      val collected : int ref
      val collectedfrom : int ref
      val current : string ref
      val datalist : datalist
      val external : funclist
      val gcmessages : int ref
      val gcprof : string ref
      val majorcollections : int ref
      val minorcollections : int ref
      val opsys : int   (* 1 = vax bsd ultrix, 4.2, 4.3
                         2 = sunos 3.0, 4.0
                         3 = vax v9 (bell labs) *)
      val pstruct : object ref
      val ratio : int ref
end
```

Figure 10: The ASSEMBLY signature

constants, etc.) are just data structures that can be described machine-independently in C. `A` is arranged as an ML record at the label `runvec` in the file `VAX.prim.s`, `M68.prim.s`, etc. The machine-independent part is in `cstruct.c`.

The exceptions in this structure are just those that can be raised from C or assembly language. Exceptions as elaborated by the compiler look like references to strings, and in `cstruct.c` are implemented as initialized structures to mimic this with all the appropriate descriptors.

The values `array0` and `bytearray0` are just the array and byte-array of length 0; this is necessary because objects of length 0 are not permitted in the garbage-collectible region.

The references `collected`, `collectedfrom`, `majorcollections`, and `minorcollections` are updated by the garbage collector with performance information. The `ratio` variable can be set from ML to tell the garbage collector what ratio to maintain between heap size and amount of live data.

The variable `current` is used in execution profiling[5]; it is a pointer to an array of length 2 of ML integers. When profiling is enabled, a timer interrupt will periodically cause current[1] to be incremented.

The `pstruct` is a pointer to the `Initial` (pervasive) structure. The ML loader (`boot/loader.sml`) needs this because other modules may reference the structure `Initial`.

The `datalist` is used for linkage to sharable compiled ML code. In vanilla Unix, such code must be link-loaded into the text segment of the sml executable file, and the `datalist` provides the ML loader a way to get to it. Each element of the data list contains two strings (the name of the module, and the executable code), along with a link to the next element.

The `funclist` is similarly used for linkage to functions implemented in C. These functions could be described in the Assembly signature but this requires recompilation whenever a new function is added. The funclist is terminated by a function whose name is "xxxx". All functions in the list must have a name of exactly 4 characters. All C-functions must take exactly one argument in ML format and return a result in ML format. The functions implemented in C at this writing are:

- **fion** tells how many characters may be read from an open file-descriptor without blocking.
- **fork** does a Unix process fork.
- **prof** is obsolete.
- **syst** executes a shell command as a sub-process.
- **time** gets execution-time information from the operating system and the garbage collector.
- **argv** gets the command-line arguments of the executing program (as a list of strings).

- **envi** gets the Unix environment string (as a list of strings).
- **blas** does a structured write (as described section 17).
- **salb** does a structured read.

These functions should be careful not to allocate memory using `malloc`.

All of these functions are called from the ML "thread" (see section 7). Some C functions may require the (temporary) suspension of the ML thread, e.g. to do a garbage collection for a structured write. By setting the global variable `cause` to a nonzero value and then returning, a function may cause control to be grabbed immediately thereafter in the thread controller.

14 Access to operating system services

A typical operating system provides many services, each with one or more operations ("system calls"). A very early version of Standard ML of New Jersey made little use of these services, so it sufficed to write an assembly-language interface to each desired system call, and then call the assembly-language functions from ML programs. As the ML environment grew more sophisticated, it needed more operating system services, so the number of assembly-language interface functions grew. This eventually became intolerable.

Now there is just one assembly-language interface function, `syscall`, that takes a list of arguments and a system-call number. This function pushes the arguments onto the stack and makes the system call. The ML standard library hides the `syscall` function behind a variety of typechecked functions. Implementing these functions in ML rather than assembly language is better for two reasons: ML is safer and easier to program in, and (more important) the ML code doesn't need to be rewritten for each target architecture.

All Unix system calls return integer values. These are converted into ML integers (by shifting and incrementing) and returned as the result of `syscall`. Many system calls take parameters by reference that they stuff results into. These results are (of course) not appropriately tagged for ML. A byte-array (of the appropriate length) is used as the actual parameter for such an argument, and then the results can be extracted from the byte-array after the system call.

Integer arguments to system calls are shifted right by `syscall` to convert them into their (untagged) representation.

String and byte-array arguments are left alone, yielding their natural representation. However, strings in C (and Unix) are terminated by a null character, whereas strings in ML have a length prefix. The prefix is ignored by the kernel, since it occurs at offset -4 (bytes). To achieve null termination of strings, you may take the actual ML argument and concatenate the string `"\000\000"` of two zero characters (two characters are required because of the unboxed representation of single-character strings, and the possibility that the actual argument is an empty string).

15 Constructor representations

Those unfamiliar with ML might wish to skip this section.

Some of the constructors in an ML datatype may not carry values. For example, in the type

```
datatype 'a option = NONE | SOME of 'a
```

the `SOME` constructor carries value of type `'a` and the `NONE` constructor carries no value. Constructors that carry no value are represented not as two-element records, but as (unboxed) small integers. In this case, `NONE` is represented by the integer 0 (with a low-order 1 tag bit), and `SOME(x)` is represented by a two-element record containing the value `x` and a small integer representing the `SOME` constructor. The ML program can distinguish which constructor has been used by testing for "boxity:" `NONE` is an unboxed value, `SOME(x)` is a boxed value, and the low-order bit will distinguish them.

Finally, in the case that there is only one value-carrying constructor, and the value carried always has a boxed representation, the extra indirection record can be eliminated. Thus, for the **list** datatype

```
datatype 'a list = NIL | CONS of ('a * 'a list)
```

the constructor `NIL` can be represented as the unboxed integer 0, the constructor `CONS(a,b)` can be represented as a two-element record containing `a` and `b`. Again, the low-order "boxity" bit distinguishes the constructors. If there had been several value-carrying constructors, then the boxity bit alone could not distinguish them, and an indirection record (containing the carried value and a small integer denoting the constructor) would be required.

It is tempting to play more elaborate tricks with the representation of constructors. For example, the *option* datatype described above might not seem to need an extra indirection; surely the boxity bit should be enough to distinguish between `NONE` and `SOME(x)`? The problem is that the value `x` might itself be either boxed or unboxed; the indirection guarantees that `SOME(x)` will indeed be boxed.

Similarly, it is tempting to use a clever representation for the datatype

```
type t = a * b
datatype atom = NUMBER of int | RECORD of t
```

Here, the values carried by the NUMBER constructor are always unboxed, and the values carried by the RECORD constructor are always boxed; the boxity can serve to distinguish them.

The problem with these schemes is caused by the abstraction provided by ML *functors*: a functor might take the atom datatype as a parameter with some of its structure hidden:

```
functor F(sig type t
              datatype atom = NUMBER of int | RECORD of t
          end)
```

and here it is not at all clear that `t` is always boxed. (Unfortunately, this problem applies to the "clever" representation of the *list* datatype as well, but we have chosen to ignore it there—we can detect the (rare) problems with partially-abstracted lists as functor parameters and give error messages, but we didn't want to use an extra indirection in the representation of lists.) The problem of functor/datatype interaction could be considered a defect in the language design.

> Note that the constructor tag is element 1 of the record and the value is element 0. There's no particular reason for this; it's probably a bad idea. For exception constructors, the constructor tag is not a small integer, it is a **string ref**. A ref cell is used instead of just a string so that (generative) exceptions may be compared for equality in pattern-matching exception constructors.

16 Suspending a process

Having designed a simple layout of runtime objects, we can now implement easily a variety of runtime services.

A convenient feature of a programming environment is the ability to save the current state of the computation in a file, so that by executing that file on a later date the computation can be resumed. In an interactive system, one might wish to compile and load some programs, and then "save the world" so that in the next session these programs don't have to be re-loaded. This feature could also be useful for program checkpoints.

It's not too hard to suspend a process in this way. It's even possible to make the resulting file an ordinary executable file. An executable file contains a header, a text segment, and a data segment. The (read-only) text segment of the saved file will be identical to the text segment of the currently executing process, so a single `write` system call will serve to write it to the file. The data segment of the file will consist of the original data segment of the process plus any newly allocated heap memory; again, one or two `write`s will put it neatly into the file. (It is useful to do a garbage collection immediately before saving, to minimize the size of the file.) Finally, the header can be synthesized appropriately.

An early version of Standard ML of New Jersey used a runtime stack, and we had no end of trouble getting the stack saved correctly. A Unix executable file doesn't have a stack segment, so the stack had to be copied into the heap before saving. And after restoring and copying back to the stack segment, we found that we had creamed the (new process's) command-line arguments, etc. When we eliminated the runtime stack, these annoyances ceased.

> The resulting executable file has a ZMAGIC magic number on Berkeley Unix, and NMAGIC on SunOs 4.0. This is because ZMAGIC implies dynamic loading of shared libraries on SunOs 4.0, which complicates the address map. Creating executable files is annoyingly difficult in operating systems like SunOs and Mach, whose notion of a process address map is not as simple as in Berkeley Unix.

17 Structured writes and reads

It's often necessary to take a data structure (with many records, pointers, strings, etc.) and write it to a file in binary form so it can be read back in quickly. It's always possible to do this in an *ad hoc* way for each different kind of data structure that has to be written, but it's sometimes difficult to get the pointer-sharing relations right. Since the runtime data format has enough information for the garbage-collector to traverse the structure, then a writer/reader must be able to traverse it as well.

In fact, we can view the saving of such a structure as just a garbage collection. The argument to the `structured-write` function is just a root pointer, and `structured-write` must traverse all the data accessible from this root. In doing so, it must copy the data to a file, being careful to make only one copy of each record.

We can, in fact, just invoke the `gc` procedure, giving as the `roots` argument just a singleton vector—the pointer that was handed to `structured-write`. Then, the resulting *tospace* will be exactly the desired contents of the file, which can be written out in a single operation.

At this point, however, memory is in a bit of a mess; some objects have been forwarded and others have not. One solution is just to finish the garbage collection: each of the "normal" roots of the data is forwarded, and then (again) all the remaining unforwarded pointers in *tospace* are forwarded. Then memory is again consistent, and execution may continue.

This is unattractive because the time required to do a structured write is now proportional to the amount of all live data, not proportional to the amount written. So a different solution is better: keep a separate list of repair information about the *fromspace* objects that were overwritten with forwarding pointers. Whenever a *fromspace* object is forwarded, two words of information must be kept about it: the location of the object itself, and the previous value of its first word (that was overwritten). Then, repair is easy; the list of repair information can be traversed, and each *fromspace* object is restored to its original state. The descriptor can be recovered by copying the descriptor of the *tospace* copy (accessed via the forwarding pointer), and the first word (which had been overwritten by the forwarding pointer) can then be recovered from the repair information. Saving the repair information is fast, and performing the repairs is also fast. The time to do a structured write is now proportional (with a small constant of proportionality!) to the amount written.

There's one last problem. Where do we keep the repair information, and what if it grows too large? The repair information can be kept at the far end of *tospace*. Then, as long as the amount written is fairly small, we'll never run into it. And if we do run into it, that means that the amount written must be at least half the size of *tospace* (where *tospace* is always at least the size of the total accessible data). What we can do in this case is just to toss away the repair information, and go ahead with a garbage collection!

> In a multi-generation system (as in SML-NJ's runtime system), we do a (cheap) minor collection before the structured write; the collection referred to in the paragraphs above is a major collection.

18 Buffered input/output

Some operating systems (like Unix) don't provide buffered input and output as a primitive, and even in operating systems that do, the overhead of one system call per character may be too high. What the runtime system can do is provide this service for user programs.

The Unix/C standard library[17] demonstrates that the runtime system need not provide this service; instead, library routines can implement buffered I/O. We have taken the same approach in our system; the buffered I/O functions are implemented in ML as library functions, and the runtime system is completely ignorant of buffered I/O.

19 Execution profiling

In large software systems it can become difficult to tell where the inefficiencies are. Execution profilers can help with this process. A typical profiler might provided information about the amount of time spent in each procedure and the number of times it was called.

Standard ML of New Jersey has an execution profiler[5], which will only briefly be described here. Call counts are handled completely independently of the runtime system; the compiler inserts program variables that are incremented on entry to procedures, and the profiler examines these variables to generate call-count information.

Execution-time estimates are more difficult. The Unix `prof` system call starts a timer interrupt in the operating system that periodically samples the program's program counter. An affine function is applied to the PC and the corresponding element of a "histogram" array is incremented. Then the histogram can be compared against the object file to see how many interrupts occurred in each procedure, which gives a good estimate of actual time spend in each procedure.

With a garbage collector that moves procedures around in memory, this method becomes unwieldy. Instead, the runtime system has a variable `current` that points at a sample-count cell for the currently executing procedure. Whenever (profiled) code begins executing a procedure (or returns from a call), it assigns the address of the new procedure's sample-count variable into `current` (the compiler inserts code to do this). The timer interrupt, instead of examining the PC, just increments the cell pointed to by `current`.

This very simple mechanism does not greatly complicate the runtime system, but provides profiling information with a relatively low overhead.

20 Debugging

We are currently developing a high-level debugger based on the same principles as our profiler—namely, minimal interaction with the runtime system. The work is still in its early stages, but we believe that it is possible to make a debugger this way, and that the runtime system won't have to grow much in size to support it. A detailed description of our plans is beyond the scope of this paper.

21 Handling of interrupts and exceptions

The ML language has an exception-handling mechanism with dynamically-nested handlers. Some exceptions are raised by the program itself, some are related to synchronous hardware exceptions (like floating point errors), and some are generated asynchronously (like the `Interrupt` exception raised when the user presses the interrupt key).

The ML program has a "current-exception-handler" register that just contains a pointer to a continuation (i.e. a closure). Raising an exception just corresponds to invoking this continuation with the exception-object as an argument. Dynamic nesting of handlers is handled completely by the compiler: it generates code that saves the previous handler (in the new handler's closure) and restores it upon exit from the scope of the new handler.

This makes it particularly easy for the programming language's exception mechanism to be attached to the hardware's notion of fault or interrupt. When a fault or interrupt occurs, the operating system calls a special (assembly-language) function in the runtime system. This function then determines what caused the fault, clears it if necessary, then *does not* return to the operating system but just invokes the current-exception-handler continuation on the appropriate exception object.

22 Fancier garbage collectors and object boundaries

Our current system uses a good, simple generational garbage collector[3]. Generation garbage collection[19][25] is based on the observations that newer records tend to become garbage more quickly, so that the collector should concentrate its effort on the newer records; and that newer records tend to point to older records, but not vice versa, so that the older records need not be searched for roots into the newer area.

Sometimes, however, older records are modified (by a `rplaca` operation in Lisp or an assignment operation in ML) to point to newer ones. Generational garbage collectors require the maintenance of a list of all modified records. To ensure portability, we use a very simple scheme to maintain this list: the compiler inserts instructions after each `update` operation to put the address of the updated cell on a special list for the garbage collector to use.

A clever trick is to use the virtual memory system in maintaining this list. It is only updates to records in older generations that must be remembered; we can just make all the memory occupied by older generations read-only, and an update will cause a page fault. The fault-handler can then put the page on a list of updated pages, and mark the page as writeable. During garbage collection, the entire page can be searched for roots into the newer region.

A different virtual-memory trick can be used to achieve concurrent garbage collection[6]. By making the pages of the *tospace* inaccessible until they have been scanned and forwarded, we can allow the ML program to continue executing even while the garbage collector is running. An access to an unscanned page of *tospace* causes a page fault, and the fault handler scans that page and makes it accessible. Then the latency is bounded by the product of the page size and the record size (in practice, to just a few milliseconds). This is very desirable in a real-time or interactive system.

Both of these algorithms require that pages be scanned (and forwarded) out of order. When objects cross page boundaries, it is difficult to know where the first object on a page begins, so that its descriptor may be found. There are solutions to this problem that involve keeping track of objects that cross page boundaries[6], but the SML-NJ runtime system has a runtime data format that allows a very simple solution.

Suppose we need to scan a particular page (to forward all its pointers). When we start at the beginning of the page, we don't know whether we are in the middle of a record or of a string; and we don't know where the descriptor for the object is. But if we knew that the page contained only records, then we wouldn't need to find the descriptor; the important thing about records is that their boundaries can be ignored. If a page consists of the last part of one record, followed by several complete records, followed by the first part of the last record, the pointers in all of those records can be scanned and forwarded without regard to the boundaries. Since the descriptors are unboxed (they look like integers), they won't be touched; and all the other fields are tagged pointers or integers which will be forwarded (or not) appropriately. (See figure 22 for an illustration.)

And if we knew that the page contained only strings, then it wouldn't need to be scanned at all, since strings contain no pointers.

Thus, a simple arrangement that's useful when pages need to be scanned in arbitrary order is to segregate pointer-containing objects (records and arrays) from non-pointer-containing objects (strings and byte-arrays). This isn't necessary for the newly-allocated objects; the compiler can continue to maintain just one register that points at the beginning of the free space. But the copying garbage collector (the functions `gc` and `forward`) must copy records and strings into different parts of memory. This greatly simplifies the implementation of page-based garbage collection algorithms.

```
     +------+--object boundary
     | desc |
     +------+
     |  o------------>
-----+------+-----page boundary
     |  5   |
     +------+--object boundary
     | desc |
     +------+
     |  o------------>
     +------+
     |  o------------>
     +------+--object boundary
     | desc |
     +------+
     |  7   |
     +------+
     |  o------------>
-----+------+-----page boundary
     |  6   |
     +------+
     |  o------------>
     +------+--object boundary
```

Figure 11: Object boundaries may be ignored by the forwarding algorithm

23 Size of the SML-NJ runtime system

Our runtime system is implemented in 1548 lines of C and 232 lines of assembly language (the assembly language must be duplicated for each target architecture). This is small indeed, considering the functionality of our Standard ML system. We have made it small by moving as much as possible out of the runtime system, and by having a simple format for ML data.

This 1780 lines of code can be divided approximately as follows:

291 lines of C to initialize and link the Standard ML library and loader, which then loads the rest of the system. (run.c)

299 lines to implement the copying garbage collector with repairs. (gc.c)

55 lines of C to implement the simulated process-switching required for garbage-collector access to ML registers. (callgc.c)

400 lines for the management of generational garbage collection, and for deciding when to ask the operating system for more memory. (callgc.c)

304 lines to implement C-language functions and data structures used as primitives by the ML program. (cstruct.c)

117 lines of C to implement the suspending of process states into Unix executable files. (export.c)

72 lines of utility functions for C functions that manipulate ML objects. (objects.c)

38 lines of assembly language to handle simulated process switching. (*.prim.s)

195 lines of assembly language to implement primitive functions callable by ML programs. (*.prim.s)

For comparison, the Icon runtime system[15] (excluding its interpreter) is 18,000 lines of C; the T3[18] runtime system is 1,900(?) lines of C and assembly language; the FranzLisp runtime system (including interpreter) is 19,500 lines of C; the Poly/ML runtime system is ??? lines of ???.

24 Conclusion

It's easy to implement a hi-tech runtime system if you do it cleanly.

25 Acknowlegements

David MacQueen, Trevor Jim, and Bruce Duba have all worked on the runtime system, and their assistance has been valuable. Norman Ramsey provided many useful suggestions about the organization of this paper.

References

[1] Andrew W. Appel.
Garbage collection can be faster than stack allocation.
Information Processing Letters, 25(4):275–279, 1987.

[2] Andrew W. Appel.
Continuation-passing, closure-passing style.
In *Sixteenth ACM Symp. on Principles of Programming Languages*, pages 293–302, 1989.

[3] Andrew W. Appel.
Simple generational garbage collection and fast allocation.
Software—Practice/Experience, 1989.

[4] Andrew W. Appel.
Tag bits aren't necessary.
Lisp and Symbolic Computation, 1989.

[5] Andrew W. Appel, Bruce F. Duba, and David B. MacQueen.
Profiling in the presence of optimization and garbage collection.
Technical Report CS-TR-197-88, Princeton University Dept. Comp. Sci., Princeton, NJ, 1987.

[6] Andrew W. Appel, John R. Ellis, and Kai Li.
Real-time concurrent collection on stock multiprocessors.
SIGPLAN Notices (Proc. SIGPLAN '88 Conf. on Prog. Lang. Design and Implementation), 23(7):11–20, 1988.

[7] Andrew W. Appel and Trevor T. Y. Jim.
Optimizing closure environment representations.
Technical Report 168, Dept. of Computer Science, Princeton University, 1988.

[8] Andrew W. Appel and David B. MacQueen.
A Standard ML compiler.
In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture (LNCS 274)*, pages 301–324. Springer–Verlag, 1987.

[9] Dianne E. Britton.
Heap storage management for the programming language Pascal.
Master's thesis, University of Arizona, 1975.

[10] Luca Cardelli.
ML under Unix.
Polymorphism, 1(3), December 1983.

[11] Luca Cardelli.
Compiling a functional language.
In *1984 Symp. on LISP and Functional Programming*, pages 208–217, 1984.

[12] C. J. Cheney.
A nonrecursive list compacting algorithm.
Communications of the ACM, 13(11):677–678, 1970.

[13] G. V. Cormack.
A micro-kernel for concurrency in C.
Software—Practice/Experience, 18(5):485–492, 1988.

[14] M. J. C. Gordon, A. J. R. G. Milner, L. Morris, M. C. Newey, and C. P. Wadsworth.
A metalanguage for interactive proof in LCF.
In *Fifth ACM Symp. on Principles of Programming Languages*, 1978.

[15] Ralph E. Griswold and Madge T. Griswold.
The Implementation of the Icon Programming Language.
Princeton University Press, Princeton, NJ, 1986.

[16] Robert Harper, Robin Milner, and Mads Tofte.
The definition of Standard ML, version 2.
Technical Report ECS-LFCS-88-62, Univ. of Edinburgh, 1988.

[17] Brian W. Kernighan and Dennis M. Ritchie.
The C Programming Language.
Prentice-Hall, Englewood Cliffs, NJ, 1978.

[18] D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams.
ORBIT: An optimizing compiler for Scheme.
SIGPLAN Notices (Proc. Sigplan '86 Symp. on Compiler Construction), 21(7):219–233, July 1986.

[19] Henry Lieberman and Carl Hewitt.
A real-time garbage collector based on the lifetimes of objects.
Communications of the ACM, 23(6):419–429, 1983.

[20] David MacQueen.
Modules for Standard ML.
In *Proc. 1984 ACM Conf. on LISP and Functional Programming*, pages 198–207, 1984.

[21] Robin Milner.
A proposal for Standard ML.
In *ACM Symposium on LISP and Functional Programming*, pages 184–197, 1984.

[22] Robin Milner.
The Standard ML core language.
Polymorphism, 2(2), October 1985.

[23] J. Rees and W. Clinger (eds.).
Revised report on the algorithmic language Scheme.
SIGPLAN Notices, 21(12):37–79, 1986.

[24] Guy L. Steele.
Rabbit: a compiler for Scheme.
Technical Report AI-TR-474, MIT, 1978.

[25] David Ungar.
Generation scavenging: a non-disruptive high performance storage reclamation algorithm.
SIGPLAN Notices (Proc. ACM SIGSOFT/SIGPLAN Software Eng. Symp. on Practical Software Development Environments), 19(5):157–167, 1984.