BIT: A Very Compact Scheme System for Microcontrollers

Danny Dubé (danny.dube@ift.ulaval.ca)
Université Laval

Marc Feeley (feeley@iro.umontreal.ca)
Université de Montréal

Abstract. We present a compact implementation of Scheme for microcontrollers that includes a real-time garbage collector. The compiler runs on a normal workstation and produces byte-code from the source program. A smart linker links the byte-code with the runtime module. We demonstrate that with this system it is clearly possible to run realistic Scheme programs on a microcontroller with as little as 3 to 4 KB of RAM. Programs that access the whole Scheme library require only 13 KB of ROM. As a byproduct of this research, we designed a novel space-efficient real-time GC algorithm.

Keywords: Scheme language, microcontroller, embedded system, byte-code, real-time garbage collection

Abbreviations: GC – garbage collector; RAM – random-access memory; ROM – read-only memory; FLASH – non-volatile random-access memory

ACM Computing Classification System (1998): D.3.4 Programming Languages Processors

1. Introduction

Embedded applications are often implemented by microcontrollers programmed in assembly language. Indeed, this yields a high degree of control over the microcontrollers and fast and compact code for simple applications. However, this approach becomes tedious and error prone for more complex applications. For this reason, compilers for higher-level languages such as Basic, C, Forth, and recently Java have been designed for microcontrollers. The goal of our work is to show that Scheme is a viable alternative for programming microcontrollers.

To illustrate the implementation difficulties and to narrow down the contextual parameters, consider for instance the popular Motorola 68HC11 microcontroller. It typically runs at a clock speed of less than 5 MHz, it has a 64 KB address space (ROM and RAM combined), on the order of 40 I/O pins, five 16 bit registers of which only one is general purpose, and no floating-point instructions.

Clearly, coping with the very tight memory constraints is one of the main problems; it requires a compact runtime system, a compact encoding of the Scheme program, and a compact object representation. Since the main application of microcontrollers is to control or monitor other devices, our system must exhibit good *real-time* behavior, that is it must avoid unduly long or unpredictable pauses in the computation, in particular during garbage collection.

The subset of Scheme that we target is R^4RS [1] with the following exclusions. We removed port-based textual I/O operations since they are not very useful in this context. Numbers are restricted to *fixnums* because microcontrollers are not intended for numerically intensive tasks, they do not support floating points numbers, and the complete Scheme numerical library is quite big. Error checking is limited to heap overflows that halt the program's execution. We otherwise assume that the program is error free. We make this assumption in order to help attain the smallest size for a complete Scheme implementation. The addition of error checking would probably increase the size of the implementation only minimally and would extend the usefulness of the system to certain safety-critical applications.

Our subset includes first-class continuations (which are useful for implementing threads), garbage collection, and proper treatment of tail recursion. Our aim is not speed; we simply wish to obtain an implementation that has the same asymptotic complexity as that of a speed-oriented implementation.

Many of the techniques we have considered in our design exist in other implementations of functional languages or are part of the Scheme and Lisp implementation folklore. One of our contributions is to study the space usage of these techniques and select those best suited for compact systems. We cite relevant previous work for the less well known implementation techniques.

Section 2 presents our byte-code compiler, with emphasis on compactness. Section 3 discusses the representation of objects. The real-time garbage collector is described in Section 4. Section 5 describes the virtual machine. An evaluation of the system is presented in Section 6.

2. The byte-code compiler

To avoid run-time overhead, our system performs a compilation phase on a development workstation which produces an executable that is then transferred to the microcontroller. The executable is composed of a byte-code sequence and a kernel that can execute this byte-code. The byte-code is generated from the source program and selected parts of

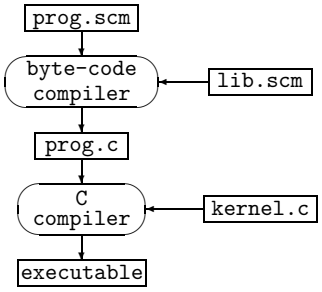


Figure 1. Compilation process of a Scheme source program (`prog.scm`).

the Scheme library. The kernel provides the garbage collector and the byte-code interpreter, which only implements the most basic Scheme functions.

This section presents the byte-code compiler which performs the compilation phase. We first give an overview of the compiler before focusing on the parts that contribute to the compactness of the resulting executable, namely the Scheme library, the processing of constants and the initial value of variables. The details of the byte-code are presented in Section 5.

2.1. Overview

Figure 1 shows the compilation process of a Scheme source program (`prog.scm`). The program can be written in normal R^4RS Scheme without constraints other than the restrictions to the language mentioned previously. The file produced by the byte-code compiler contains C code that defines three initialized arrays and their length. These arrays correspond to the byte-code, the constant descriptor, and the global-variable descriptor. Figure 2 shows the structure of the file generated for the Scheme source:

```
(display "Hello world!")
(newline)
```

Note that this program uses textual I/O functions that are not intended to be supported on microcontrollers. Simplified versions of `write`, `display`, and `newline` are defined in the library to make debugging easier on a workstation. These rely on the primitive `write-char` function. However, programs that are ready to be installed in microcontrollers restrain their use of I/O functions to microcontroller-specific ones that access the peripherals needed by the application (timers, parallel and serial ports, etc).

```
const int bytecode_len = 2594;
const unsigned char bytecode[] = {
          4,  93,   8,  94,  51,   4,  75,   8,  95,  51,   4,  74,
          8,  96,  51,   4,  55,   8,  97,  51,   4,  54,   8,  98,
                                     ...
         26,  37,  36,  52,  92,  43,  15,  37,  36,   4,  87,  52,
         92,  17};

const int const_desc_len = 27;
const unsigned char const_desc[] = {
          0,   2,  52,   0,   1,  48,  52,   0,  12,  72, 101, 108,
        108, 111,  32, 119, 111, 114, 108, 100,  33,   0,   2,   0,
          0,   0,   1};

const int nb_scm_globs = 100;
int scm_globs[] = {
           45,   -24,    50,    57,   -11,    71,   136,   150,
          -36,   -36,   212,   290,   -16,   316,   -18,   -17,
                                     ...
           -9,    -8,   -39,  2546,  2552,  2585,    -1,    -1,
           -1,    -1,    -1,    -1};
```

Figure 2. C file produced by the byte-code compiler for the "Hello world!" program.

Here is a brief description of the steps performed by the byte-code compiler:

- Reading of the program.
- Removal of the syntactic sugar.
- Transformation of the program into a node-based abstract syntax tree (AST).
- Inclusion of the required library functions.
- Traversal of the AST:
  - Gathering of the constants.
  - Identification of the variable declaration for each variable access.
  - Check for the mutability of the global variables.
  - Counting of the parameters and checking for a rest parameter.
- Assignment of the initial value of certain variables.
- A second traversal of the AST:

  - Propagation of the initial values of known variables to variable references.
  - Optimization, when possible, of call sites.

- Assignment of an index to each global variable.
- Generation of the byte-code.
- Generation of the constant descriptor.
- Generation of the global-variable descriptor.

2.2. The Scheme library

The library file `lib.scm` has a special format understood by the compiler that departs from the R^4RS syntax and semantics. It is divided into four sections.

- The first section declares the name and index of each primitive Scheme function that is provided by the runtime kernel. It helps to maintain consistency between the list of functions provided by the kernel and the one expected by the library. Each declaration is a dotted pair containing a symbol and an integer.
- The second section contains the definition of functions used internally by the library. The names introduced here are hidden to the source program.
- The last two sections both contain functions that are visible to the source program. The difference between the two is for documentation purposes only: the third section contains non-standard functions while the fourth contains R^4RS functions. In these sections, a symbol appearing alone at the top level indicates that this is a function defined in a previous section and that it is visible to the source program.

In the last three sections, the syntax is restricted to function declarations and *alias* declarations. Function declarations have the form:

`(define` ⟨*name*⟩ ⟨*λ-expression*⟩`)`

and alias declarations:

`(define` ⟨*name*⟩ ⟨*name*⟩`)`

Note that the value of each global variable of the library is either a primitive function or a closure with an empty environment. This fact is exploited to save space, as explained in Section 2.4.

Functions of the library are included with a source program according to its needs. The inclusion rule is simple: every global variable that is accessed (read or written) by the program and that is also a visible name of the library causes the inclusion of the corresponding function. Inclusion is done transitively in the library according to function dependencies.

Conceptually, the library has a separate name space from that of the program. That is, references to the name `cons` in the library and in the program do not resolve to the same variable. This is important to guarantee correct execution of the library functions even in the presence of mutations of global variables by the program. For example, the `map` library function must continue to work properly even if the program mutates the `cons` variable, which normally contains a library function that `map` uses. When the program mutates `cons`, the variable in the program name space changes but not the one in the library name space. Nevertheless, modifications of the variables containing library functions are rare. So if we detect that the program does not modify one of its variables, we unify it with its library counterpart. For example, if the program does not mutate `cons`, then the name `cons` in both name spaces resolves to the same memory location.

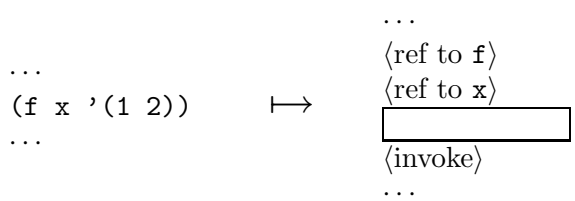
Since a large portion of the library can get included with programs, its size is important. The library is written in a style that favors conciseness over speed. For example, the functions `memq`, `memv`, and `member`, when called, simply call the parameterized function `general-member` with the same parameters plus an appropriate comparison operator. Similarly, many n-ary functions, such as `+`, are implemented as a *list folding* using an appropriate binary operation. A simple experiment in which `memq` is rewritten in a direct style (direct calls to `eq?`) reveals a 28% speed improvement.

2.3. Literal constants

Our implementation manipulates two categories of Scheme objects: *immediate* and *allocated*. Immediate objects do not have to be allocated in the heap and there are byte-code instructions that create them directly. Numbers and Booleans are immediate objects. Allocated objects reside in the heap and their creation involves calling a memory allocation function. Pairs and vectors are allocated objects. We concentrate here on the allocated ones.

Constants present in the program that are allocated objects have to be made available in the executable so that the evaluation of a constant expression at run time consists in nothing more than fetching the corresponding pre-built object. We considered three methods to

make the constants available at run time. We illustrate each method by considering the compilation of a program that embeds the expression `(f x '(1 2))`. The compilation of this program is illustrated symbolically by the following diagram where the right hand side represents virtual-machine instructions:



The empty box is intended to contain an instruction corresponding to the evaluation of the literal constant `'(1 2)` and the actual instruction depends on the chosen method.

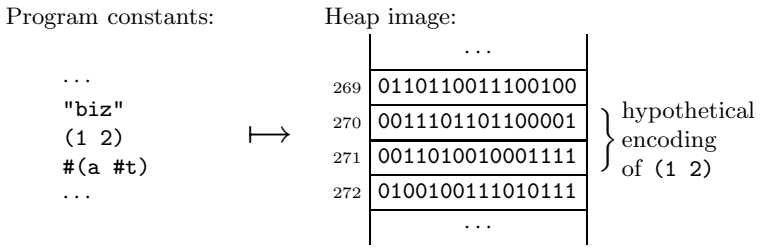
- The first method is a source to source transformation in which each constant expression is replaced by a reference to a fresh variable. Extra Scheme definitions are added at the beginning of the program to build the constants and store them in the appropriate variables. The missing instruction would be ⟨ref to `cst33`⟩.

```
; Original program
...
```

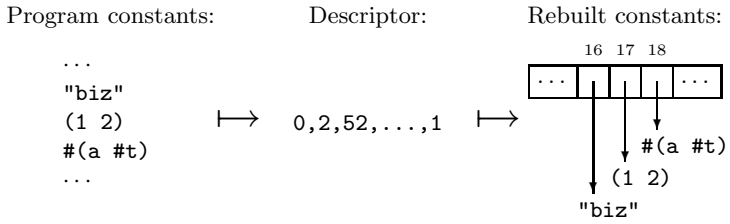
⟼

```
; Extra definition
...
(define cst32 (cons 2 '()))
(define cst33 (cons 1 cst32))
...
; Original program
...
```

- The second method consists of building at compile-time an image of the heap already containing the constants and integrating it with the executable. No run-time setup is necessary. Constant expressions are compiled as simple "load constant" instructions with a reference, ⟨load cst 270⟩ in this case.



– The third method consists of encoding the program constants into a byte-vector descriptor that is integrated with the executable. At the start of the program, an interpretation function decodes the descriptor and rebuilds the constants (essentially like `read` but with a special purpose compact encoding). Simple access instructions, such as $\boxed{\langle \text{get cst } \#17 \rangle}$, fetch the constants from a vector of rebuilt constants when necessary.



The first method has the disadvantage of making the extra construction code *and* the constants themselves coexist. This is a waste of space that the other methods avoid. The second method directly uses the image of the initial heap itself as *the* heap. So there is no construction code or descriptor that coexists along with the constants. The third method has to keep the descriptor alive until the constants have been built. But once the constants are built, the descriptor can be discarded and the space that it occupies can be coalesced with the heap to provide more free space to work with.

The second method implies that the compiler is aware of the object representation in the runtime down to the individual bits. It is more complicated to implement and maintain. The other two methods isolate the compiler from the choices of representation in the runtime kernel.

The third method requires some machinery while the second does not. Still, this machinery is relatively small. In fact, its size is constant. It does not depend on the number and size of the constants like the construction code of the first method does. This is the method that our system uses.

The encoding process is the following. First, each constant is decomposed into individual objects. Note that we make a distinction between the constants, which appear in the program as self-evaluating objects or as quoted data, and the individual objects, which form the (possibly more complex) constants. Then, each *distinct* object is given an index (this implements sharing between identical constants and sub-parts of constants); the objects are topologically ordered (children first); and information is kept to remember which objects are literal constants of the program. Finally, the descriptor is produced. It contains: the number of objects, the description of each object, the number of constants,

Const. *0*: `("biz" 2)` Const. *1*: `(#f 2)` Const. *2*: `"biz"`

(a)

Obj. *0*: `2` Obj. *3*: `"biz"` Obj. *5*: `#f`
Obj. *1*: `()` Obj. *4*: `(3 . 2)` Obj. *6*: `(5 . 2)`
Obj. *2*: `(0 . 1)`

(b)

Const. *0*: Obj. *4* Const. *1*: Obj. *6* Const. *2*: Obj. *3*

(c)

Figure 3. Steps in the encoding of a set of constants.

and the indices of the objects that are program constants. Given this encoding, it is easy to see that the construction process done at run time is very simple.

Figure 3 illustrates the encoding process of the set of constants appearing in some hypothetical program. Figure 3(a) presents the program constants. They are all allocated constants. Figure 3(b) shows the individual objects into which the constants are decomposed. There are only 7 individual objects instead of 11 because in Scheme sharing is allowed for identical constants (`"biz"` and `(2)` in this case). The contents of the pairs are denoted using object indices. Note that some of the individual objects are immediate ones. They need to be listed here because they appear inside of allocated objects. Then a binary encoding of each object and the total number of objects is produced (this is not illustrated). Finally, Figure 3(c) indicates which objects happen to be constants in the program. A binary encoding of the indices of these objects and the total number of constants is produced (also not illustrated). The concatenation of the encodings produced in this way forms the constant descriptor.

2.4. Initial value of variables

Our compiler tries to statically determine the initial value of some variables. This allows various optimizations to be performed.

The compiler only tries to statically determine the value of the global variables introduced by the library. A reason why it restricts its efforts to these variables is because their values are especially easy to determine. Also, determining the value of these variables provides an important gain in space while it may not necessarily be the case with the other variables, as we explain in the next paragraph.

The first benefit comes from a special compilation of the library code. Note that, because of the special syntax used in the library, it contains only definitions, and the expressions contained in these definitions can only be variable references or simple lambda-expressions. The result of *evaluating* the library code is simply to have a number of variables defined. Since it is possible to statically determine what function is contained in each variable, we can eliminate the code performing the evaluation of each definition's expression. Moreover, the code initializing each definition's variable can also be omitted because we can arrange for each global variable to contain the proper initial value. So our byte-code compiler produces byte-code only for the body of the closures and, when it outputs the global variables as a C array, it specifies the initial value of each variable. This is in fact a *description* of the initial value: a small negative integer for a primitive function, a positive integer which is the entry point of a closure's body, or -1 for `#f`. We arbitrarily chose `#f` as the default initial value of the variables.

The second benefit comes from the optimization of certain calls. If a call, either in the library or in the program, uses a known library function, then the operator expression no longer needs to be evaluated and a direct call to the function is made. Certain more aggressive optimizations are performed when some conditions are met. For example, the operator in the expression `(+ x y)` is optimized if the variable `+` is not mutated. The call becomes a direct invocation of the primitive function that adds exactly *two* numbers. It speeds up the execution and shortens the byte-code.

3. Scheme object representation

Even if it has little influence on the size of the executable, the object representation is of great importance due to the tight RAM constraints. A more compact representation can fit more objects in the heap and so allows our system to run a broader range of programs.

We consider the representation of the objects and their type, that of the symbols, that of the continuations, and that of the environments. In each case, we present different options and conclude with our choice.

3.1. The objects and their type

There are many approaches to represent the type and value of objects [14]. We only consider four different "pure" (as opposed to hybrid) representations.

The uniform representation. All objects are heap-allocated. The reference to an object is the address where it is allocated. Every object has an extra field that indicates its type. An advantage is that basic operations (readings, writings, type tests and GC operations) on the objects are very simple and uniform from type to type. Their implementation can be shared by all types and parameterized by the type of the objects.

The tagged pointer representation. Tagging information is written in specific bits of the pointers to the allocated objects. This is possible when, for memory partitioning or alignment reasons, some bits in the pointers always contain the same value. For instance, when the whole heap lies in some part of the memory, some of the most significant bits may be constant. Moreover, when objects are always allocated starting at the boundary of machine words, some of the least significant bits are constant. Instead of containing known (and thus useless) information, these bits can be used to encode type information. Certain bit patterns may indicate that the object reference is in fact an immediate value. This way, not all types need to be heap-allocated and heap space can be saved. Sometimes, however, there are not enough available bits to tag all the types and some allocated objects need an extra field to encode a sub-type. Tagging strategies are often complex and basic operations are implemented differently for most types.

Representation of types by zones. The heap is divided into zones with one zone per object type. Individual objects do not have to carry type information with them. The type is recovered from the address of the object by identifying the zone in which it is located. We estimate that this representation can be very compact: almost all the heap space can serve as "useful" fields. Unfortunately, it seems to be very difficult to integrate this representation with a real-time garbage collector without a very complex management that would cause an unacceptable slowdown.

Representation of types by pages. The heap is divided into pages of equal size. All the objects in a given page are of the same type. Consequently, the type needs to be indicated only once per page (in the page header) and the type of an object is recovered by rounding the address of the object down to a page boundary to read the page's type. This representation has the same advantages and disadvantages as the representation by zones. Additionally, we have to deal with the presence of long objects, such as strings and vectors, that are longer than a page.

Type	Representation
Integers	`NNNNNNNNNNNNNNN1`
Pairs	`00AAAAAAAAAAAAA0`
Closures	`01AAAAAAAAAAAAA0`
Other heap-allocated types	`10AAAAAAAAAAAAA0`
Symbols	`11NNNNNNNNNNNN10`
Characters	`11XXNNNNNNNN0000`
Kernel functions	`11NNNNNNNNNN0100`
Booleans	`11XXXXXXXXXN1000`
Empty list	`11XXXXXXXXXX1100`

Sub-type	First field
Continuations	`RRRRRRRRRRRRRRR1`
Vectors	`LLLLLLLLLLLLLL00`
Strings	`LLLLLLLLLLLLLL10`

Figure 4. Tagging scheme used in our implementation.

We consider that the tagged pointer representation is better than the uniform representation. This is because of immediate objects. After a few hundred objects are created, the gain in space due to immediate objects is likely to compensate for the more complex implementation of the operators. We did not find any satisfactory solution using one of the last two representations. So our implementation uses a tagged pointer representation.

Figure 4 shows the actual tagging scheme used in our implementation. A `0` or `1` bit is part of a tag. A `N` bit represents immediate information, that is, a part of a number or index. An `A` bit represents a part of an address (they encode the index of the object's handle, see Section 4). An `X` bit indicates that the value is not important. It is set to 1 in our implementation. Three of the types cannot be encoded directly in the reference. They need sub-typing information. So, some bits of the first field of those objects are tagged. The `R` bits encode a return address in the byte-code. The `L` bits indicate the length of a variable-sized object.

The domain of the integers is −16384 to 16383. This is more restrictive than what one would expect on a 16 bit microcontroller but it is the best we can do without allocating the integers in the heap. Because 13 bits are used to encode the address of allocated objects, there can exist at most 8192 of these. Given the maximum size of the heap, this is more than enough in normal circumstances. However, in the worst case, i.e. when the heap is almost full with small objects such as pairs, the restriction on the number of references could be a limiting factor. 4096 symbols can be represented, which is a large limit. The other immediate types are completely covered. The encoding of the first field of the continuations indirectly places a limit of 32768 on the size of the byte-code. As we show later, this limit is reasonable since the byte-code is very compact. Vectors and strings are limited to a length of less than 16384 elements.

3.2. Symbols

Symbols present some interesting possibilities. First, it is not clear whether we should represent symbols as allocated objects having a field for a name. Second, if we want to be able to compare symbols efficiently, we have to maintain their uniqueness. This requires some kind of table with the names of all the symbols. Third, symbols are not removed from this table. Knowing that, we consider the following representations:

- A symbol is a two-field object: one reference to its name, which is a string, and one link to the next symbol in the table. The whole table is a kind of list of strings but its skeleton is made of symbols instead of pairs.
- A symbol is a variable-sized object that directly contains its name and a link to the next symbol.
- A symbol is an index into a table of names. This way, the symbol becomes a non-allocated object and the table of names can be represented compactly as a vector of strings.

The second option is the least interesting because variable-sized objects are expensive to implement. It is better to avoid creating such a new type. The third option saves a field per symbol compared to the first one and is as compact as the second. Also, it introduces no new allocated type. So we adopt that representation for the symbols.

There is a small problem with the third representation as presented. In order for it to be as compact as the second representation, the table

of names has to be full. Otherwise, it is less compact. The problem with a full table is that each time a new symbol is to be created, the table has to be extended to contain the new name. Creating a longer vector and copying its content each time a new symbol appears is quite inefficient. So, in practice, each time the vector is full, we replace it by a vector that is 4/3 times the current length. This strategy makes our representation a little bit less space-efficient than the second, but the loss can be reduced by changing the ratio.

Few Scheme programs explicitly ask for the creation of new symbols at run time. As explained in Section 2.3, allocated constants have to be reconstructed during the initialization of the program environment. Consequently, a Scheme program may cause the creation of many symbols "at run time" even if it does not explicitly ask for it, simply because of the fact that it contains many symbolic constants. A sensible way of managing the table of names consists in trimming the table just after the constants are reconstructed. In this way, programs that do not create symbols at run time benefit from a maximally compact table. This optimization is not used in the current implementation of BIT.

3.3. Continuations

We consider three representations of continuations. First, a continuation can be represented using a stack. When `call/cc` is called, a copy of the stack is created in the heap. Second, the source can be CPS-converted [23]. The reification of the current continuation using `call/cc` comes for free and there are no concrete continuation types to implement. Third, a continuation can be an *ad hoc* structure that saves the current state of computation.

The stack implementation does not allow the sharing of common parts between different continuations, at least not in a simple implementation, and invoking a continuation requires an arbitrary time. Since we decided to keep continuations mostly to allow multi-threading, the representation should be compact and invoking a continuation should be a constant-time operation. The CPS-conversion has a tendency to increase the size of programs, which is not desirable. So we use an *ad hoc* structure. It is a fixed-sized object that is able to save the registers of the virtual machine that executes the byte-code (see Section 5). Among the registers that are saved, there is one that contains the current continuation. So, conceptually, the continuation is a chain of these fixed-sized *ad hoc* objects. Programs are left in direct style.

3.4. Environments

Due to their central role, environments need to be represented efficiently. Here we only consider environments for non-global lexical variables because global variables are stored separately in a statically allocated global C array. Here are the representations we consider.

Associative lists. This simple representation is not space efficient because it carries the identifiers unnecessarily. In a compiled system like ours, identifiers can be discarded completely.

Lists. This is another simple representation. It takes one pair per variable. Each access to a variable is made using a relative position in the list.

Blocks of bindings. It is possible to have a more efficient representation and still keep it very simple. We can take advantage of simultaneous bindings like those of a `let` expression to group the bound variables together in a block. Access to variables is made using a pair of coordinates: the number of binding levels (or blocks) and the position in the block. Single-variable bindings can still be represented using pairs while multi-variable bindings can be represented using vectors. The vector-based representation is more compact than a sequence of pairs in the case of multi-variable bindings.

Blocks of bindings with display. Instead of having only a link to the next block, we can use a display, and thus have a direct link to every surrounding binding block. Access to variables can always be done in constant time, independent of the lexical distance. Still, this representation, compared to simple blocks of bindings, only improves the speed. In space requirements, it can only be worse.

Flat representation of closures. Here, closures are variable-size objects that capture lexical variables. Closures themselves can be seen as special environment blocks. An advantage of this representation is the ability to select the variables to retain in the environment at closure-creation time [11]. Also, accesses to the variables are constant-time operations. On the other hand, the cost of creating a closure increases with the number of variables to capture. Finally, the flat representation by itself is not able to handle general environments since it can only represent the definition environments of closures. A representation for invoke-time bindings still has to be chosen.

```
(define make-thunk1
  (let ((a (f1 1))
        (b (f2 2))
        (c (f3 3)))
    (lambda (d)
      (lambda ()
        (list a b c d)))))
```

```
(define make-thunk2
  (lambda (a)
    (let* ((b (f1 a))
           (c (f2 b))
           (d (f3 c)))
      (lambda () (g d)))))
```

Figure 5. Two functions that create thunks with different environments.

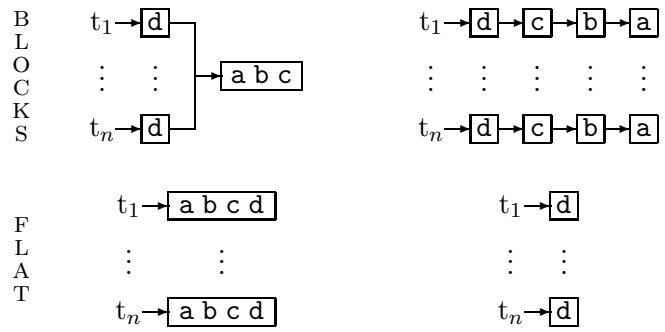


Figure 6. A comparison of the environment representation by blocks of bindings and the flat representation.

Of the first four representations, the one using simple binding blocks is clearly the best. The flat closure representation, however, is hard to compare with the others. Figure 5 shows two functions that create thunks. The environments produced by `make-thunk1` have a more compact representation using blocks and the environments produced by `make-thunk2`, using the flat representation. In the first case, it is the sharing of the blocks between environments that is advantageous. In the second, it is the ability to select the variables. Figure 6 sketches the layout of the environment of multiple thunks created using both representations for both programs.

The *safe for space complexity* rule introduced by Shao and Appel [22] states that *"any local variable binding must be unreachable after its last use within its scope"*. Flat closures have the advantage of being safe-for-space but we believe that this issue has little importance in our context because the programs are relatively small and the programmer can avoid this problem with some testing, analysis and manual program transformation.

We choose the representation with blocks because it is simpler, complete and does not require a new data type.

4. Garbage collection

Implementing a real-time garbage collector is quite a challenge and on a microcontroller especially so. We will first discuss the requirements on the memory manager. We then give an overview of the memory management technique we designed.

4.1. Requirements

The fact that the microcontroller does not have much memory means that the heap is quite small. It is tempting to assume that a blocking GC on such a small heap would be fast enough. However, the microcontrollers we target are not very fast so a complete GC cycle may cause pauses that are too long for many control tasks. Consequently, we need a real-time GC in order to provide a truly useful system.

Our GC must compact live data in some way. We cannot afford to let fragmentation ruin the possibility of allocating long objects. For example, it only takes 40 badly positioned small objects in a non-compacted heap of 4 KB to prevent the allocation of a string of only 100 characters. Because the degree of fragmentation is hard to predict in advance and depends on run-time conditions that vary over time, a non-compacted heap is not suitable for microcontroller applications that must be robust throughout their execution (that can last years).

Many real-time GC algorithms use two semi-spaces, that is, the heap is separated in two halves. During the GC cycle, live objects are transfered from one semi-space to the other. The transfer has the effect of compacting the objects. This process prevents fragmentation. Still, the use of semi-spaces represents a serious waste of space.

We did not find a real-time GC technique in the literature that tries to minimize the waste of space. The GC technique we designed addresses exactly this problem.

We first give our definition of a real-time memory manager (not just of a real-time GC). It is best presented by comparing the behaviors of three memory managers: an ordinary blocking one, an idealized one, and a real-time one. The blocking memory manager is a conventional one that offers no guarantees on the time required to perform any single operation. The idealized one has an infinitely large non-initialized memory at its disposal and takes advantage of it. That is, it does not have to free dead objects. Still, it has to provide read and write access to the fields of the objects and it has to allocate and initialize new objects. Under these circumstances, we expect constant-time access to the fields of the objects and linear-time allocation of new ones. The real-time memory manager has to deal with a finite memory but has to provide

operations with costs comparable to those offered by the idealized one. By "comparable", we mean that each operation performed by the real-time manager can be slower than those performed by the idealized one but by at most a constant factor.

Let us formalize this concept. Let *op* denote some Scheme operation that is related to memory management. It could be a read operation, such as `car`, a write operation, such as `string-set!`, or an object creation operation, such as `make-vector`. Let $\mathrm{T}(op)$ denote the time that is needed to perform *op* on a system using the idealized memory manager. Let $\mathrm{RT}(op)$ denote the worst-case time that is needed to perform *op* on some (presumably) real-time memory manager. Then the latter is real-time if there exists a constant $c \geq 1$ such that, for all operations *op*, $\mathrm{RT}(op) \leq c * \mathrm{T}(op)$ holds.

Note that our definition of a real-time memory manager does not imply that the manager should be able to execute *every* operation in constant time. The idealized manager cannot execute every operation in constant time either. Some operations have a cost that is intrinsically higher than constant time. For example, a reasonable Scheme implementation based on an idealized memory manager would be expected to allow executions of `(car x)`, `(make-string` n `#\c)`, and `(list-ref lst` i`)` in $O(1)$, $O(n)$, and $O(i)$, respectively. So we expect the same complexity to hold on the execution times in a real-time system.

This definition of real-time allows the Scheme programmer to reason easily about the time consumption of his program: each Scheme operation has a guaranteed *natural* duration. In time critical parts of his program, the programmer has to take care not to require the execution of costly operations (at the Scheme level). The system guarantees that it will not introduce unexpected pauses during the execution of the operations.

There is a side condition that must be satisfied in order to ensure that the real-time memory manager is able to meet the requirements. The program must not try to hold on to too many live objects. This condition is stated in most real-time garbage collectors. Indeed, the performance of any GC degrades when the heap is too full [28].

4.2. Overview of the GC

Our GC technique, which is described in depth in a separate paper [9], is basically an adaptation of a mark and compact blocking GC using ideas from Brooks [5]. The first phase consists in incrementally marking all the live objects of the heap. The second one compacts the marked ob-

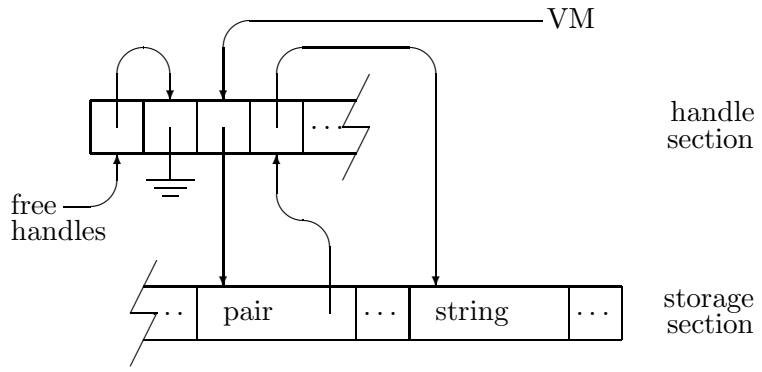


Figure 7. Sketch of the heap with handles.

jects by *sliding* them to the bottom of the heap. The program continues to run while the GC does its work.

One of the major difficulties in garbage-collecting while the program continues to run is to update pointers to objects that are moved by the GC. Since an object may have an arbitrary number of references to it, it is impossible to update them all at the moment the object is moved without causing an important pause in the execution of the program. A solution to this problem is to use *handles*.

A handle is a pointer that is unique to each object and that always points to the current position of the object. All *references* to an object go through its handle. The virtual machine and the objects themselves do not possess the address of allocated objects, they simply have the address of their handle. This implies that read and write operations now require two memory accesses instead of one. On the other hand, the handles allow the GC to move an object and instantaneously update all the references to it simply by changing the value of its handle.

Our implementation of handles is closely related to the way the *object table* is managed in Smalltalk-80 [13]. Figure 7 presents a sketch of the heap when our GC is used. Handles are kept in a separate section. The true content of the objects is located in the *storage section*. When an object is created, sufficient space is reserved in the storage section and a free handle is assigned to point to this space. This handle remains the same as long as the object exists, no matter how many times the object is moved. When an object is collected, its handle is linked back into the chain of free handles.

The handle section has a fixed size which depends on the size of the smallest objects. In our implementation, 1/4 of the heap is occupied by this section. This ratio is an improvement over the ratio of 1/3 that

would normally be used if we truly considered the smallest objects. Indeed, the smallest objects are the empty strings and the empty vectors. They have only 1 useful field: the length/sub-type field. However, we artificially extend these with one dummy field so that they become as long as the pairs. This is not a big waste as empty strings and vectors are relatively rare. Pairs, on the other hand, are very frequent. Consequently, our ratio of 1/4 is based on the fact that allocated objects require 1 field for the handle, 1 field for the back-pointer, and at least 2 fields for the useful contents. Even though 1/4 of the heap is reserved for handles, the space efficiency compares favorably to a two semi-space heap.

The use of handles eliminates the need for a *read barrier* for short objects because the handles always point to completely coherent data in the storage section (in other words the moving of small objects and the update of the handle is performed atomically by the GC). However, a *write barrier* is still needed to avoid collecting a live object whose reference is stored in an object that has been marked. This is a classic problem with real-time garbage collectors. We solve this problem with a Dijkstra barrier, that is when a reference to object X is stored in the object Y, the GC will immediately proceed with the marking of X if the GC is in the marking phase and Y has been marked.

The handling of long objects is more complicated and both read and write barriers must be used. This is because the GC cannot move long objects without exceeding the time allotted for a chunk of GC work. The object is conceptually split in two parts while the GC is moving it. During this time, access to one of its fields by the program is done either in the new (moved) part or in the old (not yet moved) part. Each time the GC is given control, it moves a bounded size chunk of the object, increasing the size of the new part and decreasing the size of the old part. This continues until the whole object is moved. To allow the program to access the right location during the movement of the long object, the GC maintains a pointer to the object and the size of the new part.

The sharing of the time between the program and the GC is ruled by a *time bank*. It is a counter that indicates how much work the GC can do before it has to give control back to the program. The execution of the GC is tightly coupled with the allocations performed by the mutator and so each allocation adds some units to the time bank. When the time bank is positive, the GC immediately starts to work and continues to do so until the bank is empty or negative. All the work involved in a complete GC cycle is divided in small work chunks, each having a unitary cost and each executing in constant time. The allocation of an object of length l adds $R*l$ time units to the bank, R being a constant,

which ensures that the program gets control back after a pause of $O(l)$ time units. This is what makes the GC real-time.

The constant R is chosen so that, by the time the rest of the free space gets allocated, the GC completes its cycle. In the worst case, the GC provides new free space exactly when the current free space is exhausted. R is called the GC's *ratio* of work. It is a function of the maximal fraction (α) of the heap that can be occupied by live objects. If it is known that the fraction of the heap occupied by live objects is never higher than α, then R will always be sufficiently large. The actual function is $R = \frac{5+3\alpha}{2-2\alpha}$ (see the original paper [9] for the details). However, we did not try to compute α to perform the experiments presented in this paper. Instead, our implementation computes a new R at the start of each GC cycle so that it makes sure that each cycle finishes in time. The ratio for a given cycle is $R = \frac{3+\rho}{2-2\rho}$, where ρ is the fraction of the heap that is occupied at the start of the cycle.

The GC technique that is used in the BIT system is a slight variation of the original technique [9]. In order to further reduce the space requirements of the heap-allocated objects, we improved the implementation of the mark stack. In the original technique, one extra field per object is required for the mark stack. In the modified technique, we do not require this extra field anymore. Instead, a *mark chain* is maintained by linking the reached objects together using their back-pointer field (i.e. the back-pointer points to the handle of the next object in the mark chain). When a marked object is scanned, its back-pointer is restored to its original value (i.e. the address of the object's handle). This way, the back-pointer field plays a dual role: implementing a mark chain during the mark phase and pointing at the object's handle during the compact phase.

4.3. Real-time systems

The integration of a hard real-time GC in the BIT system may suggest that BIT could readily be used to implement hard real-time applications. However, our single claim is that only the memory management technique meets hard real-time requirements. Our definition of a real-time memory manager agrees with the usual "constant-time operations" requirements presented in the literature on GC techniques, even if it is slightly more general.

From the point of view of hard real-time systems practitioners, the mere presence of a hard real-time GC does not automatically qualify BIT as an adequate tool for achieving specific hard real-time constraints. Our GC technique is only presented as an algorithm (and its C code implementation) which does not specify the absolute execution

times of operations. It is only when the specifics of a target machine, C compiler, and memory size are known that the absolute execution time of each Scheme operation can be determined. Moreover, hard real-time applications require bounds on the execution times of all kinds of operations, not just the ones related to memory management. Development environments for hard real-time systems must provide tools to assist the programmer in the computation of these bounds. Although, in theory, an analysis could be done manually, in practice, the desired guarantees are too costly to obtain by hand and are often checked through testing.

In principle, the programmer who uses BIT could obtain a bound on the time required by the execution of any part of his program, although this would admittedly require a lot of effort. He would have to provide a bound on the size of the objects his program maintains live at any given time. Using this bound and a description of the speed of the microcontroller, it would be possible to obtain the pace at which the memory manager has to perform garbage collection and then the execution time of every memory operation, every C function in the runtime, every virtual machine instruction, every Scheme library function, and, ultimately, every expression of the program. Note that the cost of some operations depends on the inputs that are provided to these operations. In these cases, cost functions instead of simple costs could be obtained. Moreover, provided that the C runtime contains no recursion at the C level (which is the case with BIT), a bound on the time *and* space of each operation could be computed.

5. The virtual machine

The development of our virtual machine was done in two stages. The first machine is simple but not space efficient. The second machine is a space-optimized variant of the first machine. We will use the first machine for most of the explanations because it is simpler.

5.1. A simple virtual machine

The first virtual machine has a few specialized registers: PC is the index of the next instruction, VAL is the accumulator, ENV is the current environment, ARGS is the current list of arguments, PREV_ARGS is a list of lists of arguments, CONT is the current continuation.

Figure 8 gives a list of the virtual machine's instructions. Some instructions have a variable number of operands. This is because there are variants for local/global variables, for short/long operands, and for blocks with/without a rest parameter. Access to local variables

0 ⟨description⟩ Get immediate constant.
1 ⟨index⟩ Get allocated constant.
2–5 ⟨operand$_1$⟩ [⟨operand$_2$⟩] Read variable.
6–9 ⟨operand$_1$⟩ [⟨operand$_2$⟩] Write variable.
10 Make closure.
11 ⟨address⟩ Conditional jump.
12 ⟨address⟩ Unconditional jump.
13 ⟨address⟩ Save continuation.
14 Restore continuation.
15 Initialize argument list.
16 Push argument.
17 Apply.
18 ⟨index⟩ Apply kernel function.
19 Flush environment.
20–23 ⟨size⟩ Make binding block.
24 Stop.
25 Save argument list.
26 Restore argument list.

Figure 8. Instructions of the first virtual machine.

$\mathcal{C}^*[\![$ `(set!` ⟨var⟩ ⟨exp⟩`)` $]\!] =$

- $\mathcal{C}[\![$ ⟨exp⟩ $]\!]$
- Write variable ⟨operand$_1$⟩ [⟨operand$_2$⟩]
- Restore continuation

Figure 9. Compilation rule for `set!` in terminal position.

is specified by a "number of blocks to jump over" and "position in the block" pair of operands. The second operand is omitted in certain cases: when the designated binding block contains only one variable, the second operand is assumed to be 0.

The compilation rules are quite straightforward. The only part that is a little more sophisticated is the set of rules for calls which depend on what the compiler knows about the operator: the operator is statically unknown, it is a kernel function, it is a closure from the library, or it is a lambda-expression. Figure 9 shows one of the compilation rules. $\mathcal{C}^*$ and $\mathcal{C}$ are the compilation functions for expressions in terminal and non-terminal position respectively.

0–1 ⟨index⟩ Get allocated constant.
2–5 ⟨operand$_1$⟩ [⟨operand$_2$⟩] Read variable.
6–9 ⟨operand$_1$⟩ [⟨operand$_2$⟩] Write variable.
10 Make closure and restore continuation.
11 ⟨address⟩ Conditional jump.
12–13, 19 ⟨address⟩ Unconditional jump.
14 Restore continuation.
15 Initialize argument list.
17 Apply.
20–23 ⟨size⟩ Make binding block.
24 Stop.
25 Save argument list and reinitialize argument list.
26 Restore argument list.
27–34 [⟨description⟩] Get immediate constant.
35 Drop binding block.
36–41 Read local variable (specialized).
42–44 Make binding block (specialized).
45 Save continuation and initialize argument list.
46 Set return address and apply.
48 ⟨address⟩ Make closure and unconditional jump.
49–50 ⟨operand⟩ Pop multiple arguments.
51 Pop one argument.
52–55 ⟨index⟩ Read global variable and apply contents.
215–255 Apply kernel function.

Figure 10. Instructions of the second virtual machine.

5.2. The final machine

While experimenting with the first virtual machine, we discovered, as expected, several ways in which the compactness of the code could be improved by modifying the virtual machine. These modifications exploit common patterns of instructions that are generated by the compiler. New instructions are added to perform the same operations as the patterns but more compactly. Sometimes these new instructions eliminate the need for some of the instructions of the first virtual machine. The instruction set of the final virtual machine is summarized in Figure 10. We do not present every detail of the evolution leading to the final virtual machine, only the main classes of modifications.

Specialized instructions. Some instructions are almost always used with the same operands. In these cases, we created new instructions that are specialized for those operands. For instance, we determined with a set of sample programs that 90% of the local variables that are read are located in one of these pairs of coordinates: `(0,0)`, `(0,1)`, `(0,2)`, `(1,0)`, `(1,1)`, and `(2,0)`. Also, the operand of the "Apply kernel function" instruction has been eliminated by creating a separate instruction for each kernel function.

Merged instructions. Some instructions always occur next to some other instructions. For example, the instruction "Save continuation" always precedes the instruction "Initialize argument list". So, an instruction that does both operations was created.

Automatic push. The instruction "Push argument" is so frequent that we made it implicit. All instructions that produce a value directly add it to the argument list. An explicit "Pop argument" has to be done when the pushed value is not desired.

New instructions. For example, the instruction "Pop the first block from the environment" was added.

This new virtual machine allows the byte-code to be considerably more compact. A comparison of the two machines was done with two programs: the first one is a program that forces the inclusion of all of the library and the second one is a parser generator. When these programs are compiled for the first virtual machine, about 10500 bytes of byte-code are produced for each program. When they are compiled for the second machine, about 5500 bytes of byte-code are produced for each program. This demonstrates that our R^4RS Scheme library fits in 5.5 KB of byte-code.

6. Evaluation

The goal of this section is to evaluate the practicality of the BIT system for implementing space-constrained embedded real-time applications. It is difficult to characterize these applications because there is a wide range of performance requirements and available embedded computing platforms. Some embedded applications are based on small single-chip microcontrollers with a slow clock, and very little RAM and ROM (for example, the PIC12C508 8-pin 8-bit CMOS microcontroller has 25 bytes of RAM, 512 words of ROM, a one microsecond instruction cycle time, and currently costs less than one dollar in bulk).

Typical applications include controlling a car's ignition and anti-lock braking systems, controlling household appliances, and "intelligent" toys. At the other extreme, where processing power is critical, there are platforms with specialized signal processing hardware and several megabytes of memory.

6.1. Hitachi H8

The target applications of the BIT system are those that require low to moderate computing power and where a few kilobytes of memory are required. A representative application is hobby robotics, as exemplified by the LEGO MINDSTORMS robot kit. The computing element of this kit can control up to 3 motors and read up to 3 sensors. It is based on a 16 MHz Hitachi H8/3292 microcontroller with an external 32 KB RAM. The firmware is stored in the microcontroller's 16 KB ROM. With an infrared link it is possible to upload machine code programs into the first 28416 bytes of RAM.

For this application, the BIT system was extended with 8 primitives for controlling the motors, sensors, LCD display and speaker. These primitives call low-level routines in ROM that access the microcontroller's I/O ports. Code was also added to the byte-code interpreter's main loop to show an activity status while the Scheme program is running and to properly respond to the on/off pushbutton.

This system could be used by hobbyists and researchers to quickly experiment with various high-level robot control and navigation algorithms. It also is appropriate for an academic setting to teach Scheme programming and robotics to students, whether they are beginners or advanced. Development and debugging could be done on a workstation using a full featured Scheme system augmented with a simple robot simulator, and then the program would be uploaded to the robot for testing after compiling it with BIT.

6.2. Zilog Z8 Encore!

The BIT system was also ported to the Zilog Z8 Encore! family of 8-bit microcontrollers. The target platform is powered by a 20 MHz Z8F6401 microcontroller which internally has 64 KB of FLASH memory (for program code) and 3840 bytes of RAM. It consists only of the microcontroller, an infrared transceiver (for uploading Scheme programs and I/O), three light-emitting diodes, a crystal, two capacitors, one resistor, and a 3 volt battery. The total cost of the parts is below 10 dollars and it fits in a volume of roughly 1 cm^3 and weighs 1.5 grams (see Figure 11). The current consumption while a program is running is roughly 30 milliamperes making it possible to replace the battery by

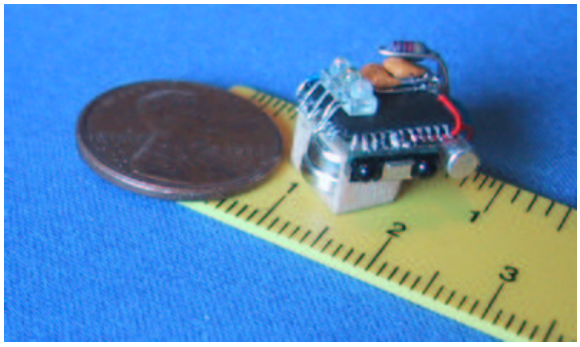


Figure 11. This picture of the target Z8 Encore! platform next to a penny shows its small size.

a small solar cell. This platform could be used as the brain of a very compact robot or remote sensor.

For this application, the BIT system was extended with a primitive to control the light-emitting diodes. The infrared port is accessed through the standard `read-char` and `write-char` functions.

6.3. Performance

Five programs were used in evaluating performance.

`empty`: Empty program.

`thread`: Small multi-threaded program that manages three concurrent threads with `call/cc`. The threads perform a tail-recursive loop which calls on each iteration a function that forces a context switch to another thread.

`photovore`: Program which controls a mobile robot to guide it towards a source of light (using a light sensor and 2 motors). The source code is given in Figure 14.

`all`: Program which references each Scheme library function once. The implementation of the Scheme library is 894 lines of Scheme code.

`earley`: Earley's parser, using an ambiguous grammar.

The `photovore` program is a realistic robotics program with real-time requirements. The other programs are useful to determine the minimal space requirements (`empty`), the space requirements for the complete Scheme library (`all`), the space requirements for a large

Program	Lines of code	Byte-code	H8/3292 Read only	H8/3292 Read write	Z8 Encore! Read only	Z8 Encore! Read write
`empty`	0	1296	8894	2196	16326	2603
`photovore`	38	1552	9226	3272	16808	3661
`thread`	44	1744	9386	2840	16820	3243
`all`	173	5479	13396	2404	20824	2799
`earley`	653	6253	13976	7244	21404	–

Figure 12. Space requirements in bytes for each platform and program.

program (`earley`), and to check if multi-threading implemented with `call/cc` is feasible (`thread`).

6.3.1. *Space Requirements*

For each of these programs, we used the smallest Scheme heap at which the program could execute without causing a heap overflow. Because an incremental collector is used, this heap size is larger than the maximal amount of space occupied by live objects during execution. But it is so by at most a constant factor, which is related to the GC's ratio of work (R). Although program execution speed can be increased by using a larger heap it is interesting to determine what is the absolute minimum amount of memory required.

Figure 12 shows for each program the memory (in bytes) required for read-only data (which includes the byte-code interpreter, the program's byte-code and constants) and for read-write data (which includes the Scheme heap and global variables). The size of the source code and that of the byte-code also appear in this figure. The `gcc` C compiler version 3.3 was used to cross-compile the system to the H8/3292 with the following compilation options: `-O2 -fno-builtin -fomit-frame-pointer`. For the Z8 Encore! the ZDS II C compiler version 4.2.0 was used with no optimization. These tests were performed on Zilog's Z8 Encore! development kit which uses a 18.432 MHz clock.

Note that the read-write memory requirements of `earley` exceed the 3840 bytes of RAM available on the Z8 Encore!, so it could not be executed on that platform.

The space required for the byte-code interpreter's machine code for the Z8 Encore! (14423 bytes) is almost twice that of the H8/3292 (7416 bytes). This can be explained by the difference in C compilers,

processor architectures and machine instruction encoding. The size of the program's byte-code, even for large programs, is considerably smaller. The total read-only data required is the sum of the interpreter's size, the size of the program's byte-code, and a few hundred bytes for various tables used to initialize Scheme constants and global variables. Note that the minimal byte-code size is 1296 bytes. This accounts for the part of the Scheme library that initializes the table of Scheme constants (even though the linker could remove this part of the library when it is useless, it does not do so because only atypical programs do not use Scheme constants).

The amount of read-write memory required is proportional to the peak amount of data held by the Scheme program and the number of global variables. It is noteworthy that some of the Scheme programs, in particular `photovore`, fit in less than 4 KB of RAM.

We experimented with `thread` to measure how much heap space is required per thread. The smallest heap is 39 KB when the number of threads is increased to 200, which corresponds to about 190 bytes per thread. Of course, more space would be required per thread when the context switches are performed at moments when the thread's continuations is larger (e.g. during a deep recursion). By using continuations, a better usage of memory is possible than the prevalent implementation of threads which allocates a fixed-size block of memory to hold the stack of each thread.

6.3.2. *Execution Speed*

As might be expected the speed of execution on these platforms is rather low in absolute terms. The number of byte-code instructions executed per second for a simple tail-recursive loop is roughly 8000 on both platforms. This low speed is due to the low computing power of these 8-bit processors, the use of a real-time collector and little RAM, and the space-conscious coding style of the byte-code interpreter and library. Nevertheless, we get adequate performance for the `photovore` application which requires a certain degree of promptness to properly control the motors as a result of the light sensor readings.

6.4. Related Work

Other implementations of Scheme have been designed to be compact but none to our knowledge share the specially tight constraints imposed by microcontroller applications. Most implementations, but not BIT, implement a read-eval-print loop and `eval`.

Some implementations are small but principally because they leave out important features of R^4RS, such as `call/cc`, proper tail-recursion,

and in some cases even recursion. For example, the LEGO/Scheme system [27], which compiles programs into the byte-codes understood by the LEGO MINDSTORMS robot's built-in interpreter, is limited by that interpreter's capabilities: it cannot allocate memory (e.g. pairs, closures), handle more than 31 variables, and perform procedure calls except tail-calls and calls to a very limited set of predefined procedures. LEGO/Scheme is so crippled that programming is as tedious as when using assembler with none of the benefits. XS [31] is another system for the LEGO MINDSTORMS. By reprogramming the robot's firmware it is able to support a more complete subset of Scheme which nevertheless does not include `call/cc`. By means of an infrared communication link with a user-interface program running on a workstation, the Scheme program can execute `load`, textual console I/O (`read`, `write`, etc) and provide the user with a traditional read-eval-print loop. The system uses a mark and sweep blocking collector and only 3 KB of the 32 KB RAM memory are left for the heap. The small heap imposes a severe limit on the size of programs because the heap contains both the data and the program represented as a S-expression.

Yet other implementations target specific applications and consequently provide extensions to R^4RS. A fair comparison would have to take into account the complete set of features of each implementation. The goal of this section is less lofty. It only aims to give a rough feel of the size of the implementations by measuring the size of the executables.

We obtained the source code of several Scheme implementations which appeared to be compact and compiled them on an Athlon-based GNU/Linux workstation. The makefiles of the systems were used when available. Dynamic linking was used when possible and the executables were then stripped to remove debugging information. In the case of BIT, we compiled it using `gcc` without any options and then we stripped the executable.

Figure 13 shows the results. The only implementation whose size comes close to BIT is Mini-Scheme. This implementation is far from being R^4RS compliant and part of the Scheme library is in an initialization file loaded at startup which is not accounted for in our size measure.

When it comes to time efficiency our implementation is comparable to other systems, but somewhat on the slow side. We measured the execution time of `photovore` modified so that it exits after 200 sweeps and with dummy function definitions for the robot specific primitives. The execution when using BIT and a large heap is 2.0 times slower than when using the Gambit interpreter [10] and 8.6 times slower than when using the SCM interpreter which is one of the fastest Scheme

Implementation		Size of interpreter
`QScheme 0.5.1`	[8]	198 KB
`SCM 5.7`	[15]	168 KB
`SIOD 3.2`	[6]	160 KB
`LispMe 3.11`	[4]	151 KB[1]
`Pocket Scheme 1.1.0`	[12]	124 KB[2]
`fools 1.3.2`	[20]	75 KB
`Vx-Scheme 0.3`	[25]	66 KB
`TinyScheme 1.33`	[26]	45 KB
`Mini-Scheme 0.85`	[21]	32 KB
`BIT`	(byte-code interpreter with full library)	22 KB

Figure 13. Size of different small Scheme implementations.

interpreters available. While we took great care with space-efficiency, we essentially ignored execution speed as long as it stayed reasonably (asymptotically) efficient.

The main sources of inefficiency come from the memory management and the virtual machine. First, even in the best conditions, our GC is not the fastest incremental GC (see the work of Larose and Feeley [16] for a comparison). Second, we do not try to reduce the GC overhead by grouping the collection phases into coarser, less frequent phases. So the GC is called during most of the allocations. Third, since our virtual machine does not use a stack, it keeps the arguments of each call in a list. It means that a pair must be allocated for each argument. Given that memory management is slow, this process is rather heavy. Finally, the concise style in which the library is written adds to the time inefficiency. Higher-order functions are extensively used, even in many apparently basic operations such as `+` and `<`.

Although there is extensive literature on real-time garbage collection, we did not find another GC technique that tries to minimize the heap space that is occupied by administrative structures. Ours provides hard real-time guarantees, eliminates fragmentation, and allows objects

[1] LispMe is intended to run on a Palm Pilot. We did not have a version that ran on our workstation. The size that is given is that of the image file (`LispMe.prc`) that goes directly on the Palm.

[2] The size of the `Pocket Scheme` interpreter is that of the Windows executable along with its companion DLL.

of arbitrary length [9, 16]. Other techniques either use two semi-spaces or some heap organization that is costly in space [5], do not eliminate fragmentation [29, 30], do not provide hard real-time guarantees [2], cannot accommodate large objects [3, 24], or a combination of these and so we do not consider them to fit our needs. Interestingly, the technique presented by Bacon *et al.* [2] regulates garbage collection effort on a time basis, while most other techniques work on an allocation basis. The pace of the execution of the program is very steady as the program is not directly accountable for its allocations. However, it is the responsibility of the programmer to guarantee that his program does not allocate too much data during any period of time. Unless the program allocates very little, this is a condition that is quite hard to verify.

Microcontroller implementations of high-level languages such as Basic, C, and Forth have existed for some time now. More recently, efforts have been invested to adapt Java to this task too. In particular, JavaCards with an 8-bit processor and a few KB RAM can be programmed to perform a variety of tasks. However, the subset of Java supported does not include some important features such as garbage collection, multidimensional arrays, strings and threads [7] which lowers the expressive power of the language.

6.5. Future work

We can think of many ways to extend our work.

- The unnecessary machinery that rebuilds the allocated constants could be dropped. If no constant of a certain type has to be rebuilt, the construction code specific to this type is useless.
- The symbol names should be dropped, when possible. Often, only the identities of the symbols are required, not their names.
- The runtime could be given the ability to drop the parts of the byte-code that become useless and turn them into additional heap space. Indeed, it is quite common to have parts of Scheme programs intended only for the initialization.
- The compiler should provide the user with flags to control the inclusion of features and declare properties about the program.
- The time efficiency could be improved.
- We should consider the execution of compressed byte-code. Latendresse *et al.* [18, 17, 19] have demonstrated that byte-code

compressed using Huffman encoding could be executed directly with a negligible loss of speed. A Huffman encoding of the byte-code and a customized virtual machine can be generated on a per-program basis, leading to very compact representations of Scheme programs. However, the decoding virtual machine, which is not compressed, can become relatively large if good execution speed is desired.

- A better implementation of environments could be provided. Environment representations that are tailored to the local needs of the Scheme expressions would be preferable (see Figures 5 and 6).
- Various analyses that are well known in the speed optimization areas could be put to use in space optimization areas too. Such analyses include flow analyses [23], dead code detection, representation analyses, useless-variable detection, and storage use analyses.

7. Conclusion

Our goal was to determine whether it is possible to program microcontrollers in Scheme. The two major constraints concern space and real-time-ness of the implementation. In order to obtain a small implementation, we took advantage of the static nature of microcontroller applications and separated the implementation in a byte-code compiler and a runtime kernel. The compiler is designed to run on a normal workstation. It produces byte-code which, added to the runtime kernel, provides a small executable code to transfer to the microcontroller.

We took great care in our design to favor space efficiency. The principal choices concern: run-time representation of Scheme objects such as type information and environments; memory management, which has to be real-time; the virtual machine embedded in the runtime kernel and its associated byte-code. In general, we selected the most compact approaches as long as they stayed reasonably simple and that they did not compromise the asymptotic complexity of Scheme programs.

Our results clearly demonstrate that it is feasible to program microcontrollers in Scheme. Scheme sources, once compiled, become byte-codes several times smaller. Interesting programs can be executed with as little as 9 KB ROM and between 3 KB and 4 KB RAM. The main weakness of our system is the low speed of execution, which is about 10 times slower than the fastest Scheme interpreters. However, the system delivers adequate performance for realistic applications including hobby robotics.

Acknowledgements

We wish to thank the anonymous reviewers and the editors for their helpful comments. This work was supported by grants from the Natural Sciences and Engineering Research Council of Canada and Université Laval.

References

1. Abelson, H., N. I. Adams, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steele, G. J. Sussman, and M. Wand: 1991, 'Revised^4 Report on the Algorithmic Language Scheme'.
2. Bacon, D. F., P. Cheng, and V. Rajan: 2003, 'A Real-time Garbage Collector with Low Overhead and Consistent Utilization'. In: *Proceedings of the Symposium on Principles of Programming Language*. pp. 285–298.
3. Baker, H. G.: 1978, 'List Processing in Real-Time on a Serial Computer'. *Communications of the ACM* **21**(4), 280–294.
4. Bayer, F., '`LispMe` implementation'. `http://www.lispme.de/lispme/index.html`.
5. Brooks, R. A.: 1984, 'Trading data space for reduced time and code space in real-time collection on stock hardware'. In: *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*. pp. 108–113.
6. Carrette, G., '`SIOD` implementation'. `http://www.cs.indiana.edu/scheme-repository/imp/siod.html`.
7. Chen, Z.: 2000, *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Addison-Wesley.
8. Crettol, D., '`QScheme` implementation'. `http://www.sof.ch/dan/qscheme/index-e.html`.
9. Dubé, D., M. Feeley, and M. Serrano: 1996, 'Un GC temps réel semi-compactant'. In: *Actes des Journées Francophones des Langages Applicatifs 1996*.
10. Feeley, M., 'Gambit implementation'. `http://www.iro.umontreal.ca/~gambit/`.
11. Feeley, M. and G. Lapalme: 1992, 'Closure Generation Based on Viewing Lambda as Epsilon plus Compile'. *Journal of Computer Languages* **17**(4), 251–267.
12. Goetter, B., '`Pocket Scheme` implementation'. `http://www.mazama.net/scheme/pscheme.htm`.
13. Goldberg, A. and D. Robson: 1983, *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc.
14. Gudeman, D.: 1993, 'Representing Type Information in Dynamically Typed Languages'. Technical Report TR 93-27, Department of Computer Science, The University of Arizona.
15. Jaffer, A., '`SCM` implementation'. `http://swissnet.ai.mit.edu/~jaffer/SCM.html`.
16. Larose, M. and M. Feeley: 1999, 'A Compacting Incremental Collector and its Performance in a Production Quality Compiler'. *ACM SIGPLAN Notices* **34**(3), 1–9.

17. Latendresse, M.: 2000a, 'Automatic Generation of Compact Programs and Virtual Machines for Scheme'. In: M. Felleisen (ed.): *Proceedings of the Workshop on Scheme and Functional Programming*. pp. 45–52.
18. Latendresse, M.: 2000b, 'Génération de machines virtuelles pour l'exécution de programmes compressés'. Ph.D. thesis, Université de Montréal, DIRO.
19. Latendresse, M. and M. Feeley: 2003, 'Generation of Fast Interpreters for Huffman Compressed Bytecode'. In: *Proceedings of the ACM SIGPLAN Workshop on Interpreters, Virtual Machines and Emulators*.
20. Lee, J., '`fools` implementation'. `ftp://ftp.cs.indiana.edu/pub/scheme-repository/imp/fools.1.3.2.tar.gz`.
21. Moriwaki, A., '`Mini-Scheme` implementation'. `ftp://ftp.cs.indiana.edu/pub/scheme-repository/imp/minischeme.tar.gz`.
22. Shao, Z. and A. W. Appel: 2000, 'Efficient and safe-for-space closure conversion'. *ACM Transactions on Programming Languages and Systems* **22**(1), 129–161.
23. Shivers, O.: 1991, 'The Semantics of Scheme Control-Flow Analysis'. In: *Proceedings of the Symposium on Partial Evaluation and Semantics-based Program Manipulation*. pp. 190–198.
24. Siebert, F.: 1999, 'Hard Real-Time Garbage Collection in the Jamaica Virtual Machine'. In: *Proceedings of the Sixth International Conference on Real-Time Computing Systems and Application*. Hong Kong, China, pp. 96–102.
25. Smith, C., '`Vx-Scheme` implementation'. `http://colin-smith.net/vx-scheme/`.
26. Souflis, D., '`TinyScheme` implementation'. `http://tinyscheme.sourceforge.net/`.
27. Wick, A. C., M. Wagner, and K. Klipsch, '`LEGO/Scheme` implementation'. `http://www.cs.indiana.edu/~mtwagner/legoscheme/`.
28. Wilson, P. R.: 1992, 'Uniprocessor garbage collection techniques'. *Lecture Notes in Computer Science* **637**, 1–42.
29. Wilson, P. R. and M. S. Johnstone: 1993a, 'Real-Time Non-Copying Garbage Collection'. In: *ACM OOPSLA Workshop on Memory Management and Garbage Collection*. Washington D.C.
30. Wilson, P. R. and M. S. Johnstone: 1993b, 'Truly Real-Time Non-Copying Garbage Collection'. In: E. Moss, P. R. Wilson, and B. Zorn (eds.): *OOPSLA/ECOOP Workshop on Garbage Collection in Object-Oriented Systems*.
31. Yuasa, T.: 2003, 'XS: Lisp on Lego MindStorms'. In: *International Lisp Conference 2003*.

```
; This program controls a LEGO MINDSTORMS robot so that it will find a
; source of light on the floor (flashlight, candle, white paper, etc).
; The robot is made of 2 motors (A and C) and a light detector (at
; position 2).  Each motor controls one of the wheels.  Only one motor
; is active at any moment, so the robot zigzags towards its target.
; It sweeps on one side, and then the other, and so on.  On each sweep
; it determines at which heading the reading of the light sensor was
; greatest and this heading becomes the nominal heading of the next
; sweep.  Once in a while a wide sweep is performed.

(define narrow-sweep 20) ; width of a narrow "sweep"
(define full-sweep 70) ; width of a full "sweep"
(define light-sensor 1) ; light sensor is at position 2
(define motor1 0) ; motor 1 is at position A
(define motor2 2) ; motor 2 is at position C

(define (start-sweep sweeps limit heading turn)
  (if (> turn 0) ; start to turn right or left
      (begin (motor-stop motor1) (motor-fwd motor2))
      (begin (motor-stop motor2) (motor-fwd motor1)))
  (sweep sweeps limit heading turn (get-reading) heading))

(define (sweep sweeps limit heading turn best-r best-h)
  (write-to-lcd heading) ; show where we are going
  (if (= heading 0) (beep)) ; mark the nominal heading
  (if (= heading limit)

      (let ((new-turn (- turn))
            (new-heading (- heading best-h) ))
        (if (< sweeps 20)
            (start-sweep (+ sweeps 1)
                         (* new-turn narrow-sweep)
                         new-heading
                         new-turn)
            (start-sweep 0
                         (* new-turn full-sweep)
                         new-heading
                         new-turn)))

      (let ((reading (get-reading)))
        (if (> reading best-r) ; high value means lots of light
            (sweep sweeps limit (+ heading turn) turn reading heading)
            (sweep sweeps limit (+ heading turn) turn best-r best-h)))))

(define (get-reading)
  (- (read-active-sensor light-sensor))) ; read light sensor

(start-sweep 0 full-sweep 0 1)
```

Figure 14. The source code of the `photovore` program.