BIT: A Very Compact Scheme System for Embedded Applications

Danny Dubé
DIRO
Université de Montréal
dube@iro.umontreal.ca

Abstract

We present an implementation of Scheme for microcontroller that is very compact and includes a real-time garbage collector. The compiler, which runs on a normal workstation, produces byte-code from the source program and the byte-code is linked with a runtime module. With this system, we demonstrate that it's clearly possible to run Scheme programs on a microcontroller with 64 KB of memory such as the Motorola 68HC11. Executables that include the whole library can be as little as 22 KB. As a secondary result, the research on memory management for this system brought us to create a space-efficient real-time GC algorithm.

1 Introduction

Embedded applications are often implemented by programming microcontrollers in assembly language. Indeed this provides a high degree of control on the microcontroller 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 and Forth have been designed for microcontrollers. The goal of our work is to show that Scheme is also a viable alternative for programming microcontrollers.

To better understand the implementation difficulties and narrow down the contextual parameters, we will pick a popular microcontroller family as a target: the Motorola 68HC11. This controller typically runs at a clock speed of less than 5 MHz, it has a 64 KB address space (ROM and RAM combined), up to 40 I/O pins, five 16 bit registers of which only one is general purpose, and no floating-point operations.

Clearly, coping with the very tight memory constraint is one of the main problems we face; it implies compact run-time system, encoding of the Scheme program, and object representation. Also, since the main application of microcontrollers is to control or monitor other devices, our system must have good *real-time* behavior, that is it must not make unduly long or unpredictable pauses in the computation, in particular when garbage collecting.

The subset of Scheme that we target is R^4RS (see [2]) with the following exclusions. Port-based textual I/O operations are removed since they don't make sense anymore. Numbers are restricted to *fixnums* because the chip itself isn't intended for numerically intensive tasks, it doesn't support floating points numbers, and the complete Scheme numerical library is quite big. Error checking isn't performed. Indeed, our system is not interactive and there is normally no way to report the error. So we assume the program is error free. Our subset does include first-class continuations (which are useful for multi-threading), garbage collection, and proper treatment of tail recursion. We don't aim to have a very fast implementation; we simply wish to obtain an implementation that has the same asymptotic complexity as those of a good implementation.

Numerous issues must be considered in the design of a very compact Scheme implementation. We begin in Section 2 by presenting our byte-compiler with emphasis on points related to compactness. In Section 3 we discuss the representation of objects. Our real-time garbage collector is described in Section 4. We describe the virtual machine in Section 5. Finally, Section 6 presents some experimental results.

2 The byte-compiler

To avoid run-time overhead, our system performs a compilation phase on a normal workstation which produces an executable that is then transferred to the microcontroller. The executable is composed of a byte-code sequence and a kernel which can execute this byte-code. The byte-code is generated from the source program and selected parts of the Scheme library. The kernel provides the garbage collector and the byte-code interpreter, which only includes the most basic Scheme functions.

This section presents the byte-compiler which performs the compilation phase. We first give an overview of the compiler and then give more details on the parts that address 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 independently in the section on the virtual machine (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 other constraints other than the restrictions to the language that we mentioned in the introduction. The file produced by the byte-compiler is actually a C file containing three declarations of array and their length. Those contain: the byte-code, the constant descriptors, and the global variable descriptors. Figure 2 shows an abstract of the file generated for the Scheme source[1]:

[1]This program uses functions that are not supposed to be supported by our implementation. Still, we included simplified versions

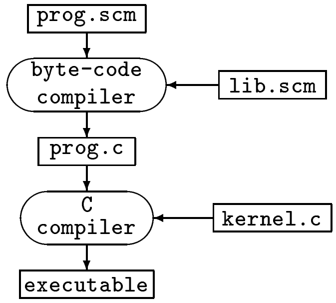

Figure 1: Compilation process of `prog.scm`.

```
int bytecode_len = 2594;
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};

int const_desc_len = 27;
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};

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-compiler for the "Hello world!" program.

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

Here's a brief description of the steps performed by the byte-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.
  - Checking 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.
- Another traversal of the AST:
  - Propagation of the initial value 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 descriptors.
- Generation of the global variable descriptors.

of `write`, `display`, and `newline` to make tests during the development.

2.2 The Scheme library

The library file `lib.scm` has a special format. It doesn't follow the syntax of Scheme programs. It's divided in four sections. Each section has a different purpose.

- The first section declares the name of each primitive Scheme function that is provided by the runtime kernel and its index. It helps to maintain the 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 contain functions that are visible to the source program. The difference between both is that the last contains only *standard* functions. In these sections, a symbol appearing alone at the top level indicates that this is a function declared in a previous section and that it's visible to the user's program.

In the last three sections, the syntax is restricted to function declarations and *alias* declarations. Function declarations have the form:
    (define $\langle name \rangle$ $\langle \lambda\text{-}expression \rangle$**)**
and alias declarations:
    (define $\langle name \rangle$ $\langle name \rangle$**)**
Note that the value of each global variable of the library is either a primitive function or a closure with an empty environment. This regularity can be exploited to save space, as we explain in Section 2.4.

Functions of the library are included with the source program according to its needs. The inclusion rule is quite 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 the program. That is, references to the name `cons` in the library and in the program don't resolve to the same variable. This is important to guarantee correct execution of the library functions even if the program modifies its global variables. Nevertheless, modifications of the variables containing library functions are rare. So if we detect that the program doesn't modify one of its variables, we fold it with its library counterpart.

Since a sizeable part of the library typically gets included with programs, its length is important. The library is written in a very concise style. 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 a binary operation.

2.3 Literal constants

Our implementation manipulates two categories of Scheme objects: *immediate* and *allocated*. Immediate objects don't 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 have to be communicated to the executable so that they are available when their corresponding expression gets evaluated.

We considered three methods to create the constants at run time.

- Each constant expression is replaced by a reference to a new variable. Extra Scheme definitions are added at the beginning of the program to build the constants and store them in the appropriate variables.

- An image of the heap already containing the constants is integrated with the executable. No building code is necessary. Constant expressions are compiled as simple "get constant" instructions with an index.

- A description of the constants, which is a string, is integrated with the executable. At the start of the program, some interpretation function decodes the string and builds the constants. Simple access instructions get the 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. The other two can *dispose* of the description of the constants either by making it the initial heap or by discarding it once the constants are built.

The second method implies that the compiler is aware of the object representation in the runtime down to the individual bits. It's 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 doesn't. Still, this machinery is relatively small and doesn't depend on the total size of the constants the way the first method does. It's the method that we use.

The encoding process is the following: each constant is decomposed into individual objects; each *distinct* object has an index (this implements sharing between identical constants); objects are ordered in topological order (children first); information is kept to remember which objects "are" program constants by themselves; finally, the description string is produced. The string contains: the number of objects, the description of each object, the number of constants, the index of the objects that are program constants. Given this encoding, it's easy to see that the construction process done at run time is extremely simple.

2.4 Initial value of variables

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

The only variables for which the compiler searches the value are the global variables introduced by the library. The first reason for this is that it's very easy with these variables. Second, we always obtain an important gain in space with these variables while it may not necessarily be the case with the other variables.

The first gain 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's possible to statically determine what function is contained in each variable, we can omit the code performing the evaluation of each definition's expression. Moreover, the code initializing each definition's variable can be omitted too because we can arrange for each global variable to contain the proper initial value. So our byte-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, -1 for `false`, or a positive integer which is the entry point of a closure's body.

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 `+` isn't mutated. The call becomes a direct invocation of the primitive function that adds *two* numbers.

3 Scheme object representation

Even if it doesn't influence the size of the executable, the object representation is of great importance due to the small memory. A more compact representation can fit more objects in the heap and so, allows our implementation to run a broader range of programs.

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

3.1 The objects and their type

The choices in the representation of the type and value of the objects are almost unlimited (see [5]). We only consider four different "pure" representations.

The uniform representation. All objects are heap-allocated. An object reference is the address where it's 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 similar from type to type. Their implementation can be shared by all types and parameterized by the type of the objects.

The tagged pointer representation. For alignment and memory partitioning reasons, the addresses of heap-allocated objects may have some bits at known values. These bits can be used to encode type information. Moreover, 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. It results in space savings. Sometimes, however, there aren't enough bit patterns for all the types and objects of certain types need an extra-field to encode a sub-type.

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 3: Tags in our implementation.

Tagging strategies are often complex and basic operations are implemented differently for most types.

Representation of types by zones. The heap is divided in zones with one zone for each type. Individual objects don't have to carry type information. The type is recovered from the address of the object by identifying the zone in which it's 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 that representation with a real-time garbage collector without falling into a very complex management.

Representation of types by pages. The heap is divided in pages of equal size. Each page contains only objects of the same type. The type is recovered by rounding the address of an object down to the previous page boundary and obtaining 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, which should be allowed to be longer than a page.

We consider that the tagged representation is better than the uniform representation. This is because of the 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 didn't find any satisfying solution using one of the last two representations. So our implementation uses a tagged representation.

Figure 3 shows the actual tagging chosen for our implementation. The `0` and `1` digits are the tags. An `N` bit represents immediate information, that is, part of a number or index. An `A` bit represents a part of an address[2]. An `X` bit indicates that the value isn't important. It's put 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.

[2]The `A` bits don't exactly represent an address. In fact, they encode the index of the handle to the object (see Section 4).

The domain of the integers is -16384 to 16383. It's even more restricted than what one would expect on a 16 bit microcontroller but it's the best we can do without allocating the integers in the heap. There can exist at most 8192 heap-allocated objects. This is more than enough given the maximum size of the heap except for the pairs in certain circumstances. In the worst case, the number of references could be a limiting factor in the allocation of pairs. 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 creates 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 8192 fields.

3.2 Symbols

There are some interesting possibilities with the symbols. First, it isn't clear whether we should represent symbols as 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 aren't removed from this table. Knowing that, we consider the following representations:

- A symbol is a two-field object: one reference to its name 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 (plus a link).
- A symbol is a simple index in a table of names. This way, the symbol becomes a non-allocated object and the table of names can be represented compactly as, for instance, a vector of strings.

The second option is the least interesting because variable-sized objects are heavy to manipulate. It's 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's a little problem with the third representation as it's been presented. In order for it to be as compact as the second representation, the table of names has to be full. Otherwise, it's 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 longer. This strategy makes our representation a little bit less space-efficient than the second, but the loss can be reduced by changing the ratio.

3.3 Continuations

We consider three possibilities for the representation of continuations. First, a continuation can be represented by a stack. When `call/cc` is called, a copy of the stack is created in the heap. Second, the source can be CPS-converted (see [7]). The reification of the current continuation becomes natural 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 doesn't allow the sharing of common parts between different continuations. At least not in a simple implementation. Since we decided to keep continuations mostly to allow multi-threading, the representation should be efficient. The CPS-conversion has a tendency to increase the size of programs, which isn't desirable. So we use an *ad hoc* structure. It's a fixed-sized object able to save the registers of the virtual machine executing the byte-code (see Section 5). Among the registers that are saved, there is the 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. Note that we don't consider global variables, only local ones. Here are some representations.

Associative lists. It's the simplest representation to use in a Scheme interpreter. However, they aren't space efficient since they carry identifiers unnecessarily. In a compiled system like ours, identifiers can be discarded completely.

Lists. It's also a very simple implementation. It takes one pair per variable. Each access to a variable is made using a position in the list.

Blocks of bindings. It's possible to do better than what lists do and still have a very simple implementation. 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 are made using the number of binding levels (or blocks) and the position in the block. Single-variable bindings can still be represented by pairs and multi-variable bindings can be represented by vectors. The representation with vectors is more compact than with a sequence of pairs in case of multi-variable bindings.

Blocks of bindings with display. Instead of only a link to the next block, we can have a display, and thus have a link to every surrounding binding block. Access to variables can always be done in constant time, no matter 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. The advantage of this representation is the selection of the variables to keep in the environment (see [4]). On the other hand, a new block of variables is created each time a closure is. Moreover, it isn't a representation for general environments since it can only represent the definition environments of closures. A representation for invoke-time bindings still has to be present.

Of the first four representations, the one using simple binding blocks is clearly the best. Unfortunately, the fifth representation is incomparable with the others. Figure 4 shows two functions that create thunks. The thunks produced by the first function have a more compact representation using blocks. Those produced by the second, using flat closures. In the first case, it's the sharing of the blocks between environments that is advantageous. In the second, it's the selection of variables.

```
(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 4: Thunks with different environments.

We prefer the representation with blocks because it's simpler, complete and doesn't require a new data type.

4 Garbage collection

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

4.1 Requirements

The fact that the microcontroller doesn't have much memory means that the heap is quite small. It's tempting to assume that a blocking GC on such a small heap would be fast enough. But microcontrollers like ours aren't very fast. A complete GC cycle may provoke pauses that are too long for many control tasks. So we need a true real-time GC in order to provide a really useful system.

Our GC must compact live data in some way. We cannot afford to let the fragmentation ruin the possibilities of allocation of long objects. For example, it doesn't take many badly positioned small objects in a heap of 40 KB to block the allocation of a string of only 400 characters: only 100.

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 a semi-space to the other. The transfer has the effect of compacting the objects together. This process avoids fragmentation from forming. Still, the use of semi-spaces represents a serious waste of space.

In fact, we didn't find a real-time GC technique in the literature that tries to minimize the waste of space. We proposed a new GC technique that addresses exactly that problem.

Before describing it, we give our definition of a real-time memory manager. Of course, a real-time implementation cannot execute *every* operation within a bounded amount of time. Some of those are long operations, even when the GC doesn't work at all. What we expect from a real-time implementation is that an operation shouldn't get longer by an unpredictable amount of time.

More formally, let $c >= 1$, T(op), the time required to perform 'op' on a non-real-time system when the GC isn't invoked (in the ideal case), and RT(op), the time required to perform 'op' on a real-time system (in the worst case), then $\mathrm{RT(op)} \leq c * \mathrm{T(op)}$. This real-time property holds only if the program doesn't try to keep too many live objects. The performance of any GC degrades when the heap is too full.

For information on GC in general, see [8].

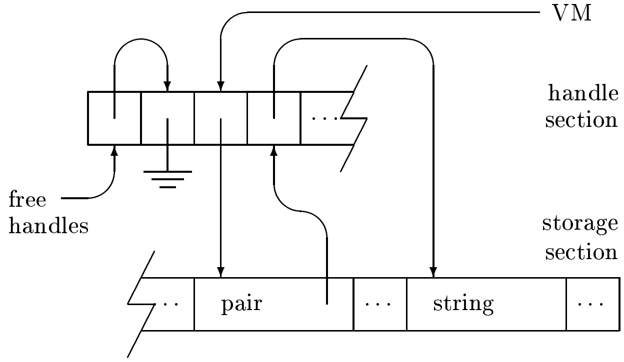

Figure 5: Sketch of the heap with handles.

4.2 Overview of the GC

Our GC technique (for a complete description, see [3]) is basically an adaptation of a mark and compact blocking GC using ideas from Brooks (see [1]). The first phase consists in incrementally marking all the live objects of the heap. The second compacts the marked objects 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's 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 don't possess the address of other 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 update all the references to it by changing the value of its handle.

Figure 5 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.

Unfortunately, the handle section must have a fixed size. Its size depends on the size of the smallest objects. In our implementation, the fraction of the heap occupied by this section is 1/5. Nevertheless, the space lost because of the handle section is much smaller than the space lost in a two semi-space heap.

The usage of handles eliminates the need for a *read barrier* for short objects since the handles always point on their object. However, a *write barrier* is still needed to avoid collecting live objects. This a classic problem with real-time garbage collectors. To solve it, we use a Dijkstra barrier. The access to long objects is more complicated and read

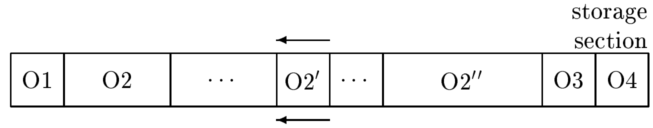

Figure 6: Long objects are incrementally slid down.

and write barriers must be used. This is because the GC cannot move a long object atomically. The object has to be momentarily separated in two parts. During this time, access to one of its fields is done either in the new (moved) part or in the old (not yet moved) part. Figure 6 illustrates this situation with a long object named O2. It has a new part (O2), an old part (O2″), and a part (O2′) that is currently being slid atomically by the GC.

The sharing of the time between the program and the GC is ruled by a *time bank*. It's a counter indicating how much work the GC can do before it has to give the control back to the program. Each allocation adds some units to the time bank. If the time bank is positive, the GC immediately goes to work and so, until the bank is empty or negative. All the work involved in a GC cycle is divided in small, constant-time work units. 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 the control back after a pause of $O(l)$ time units. That makes the GC work in real-time.

The constant R is adjusted so that, by the time the rest of the free space gets allocated, the GC completes its cycle. So, 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's a function of the maximal fraction (α) of the heap that can be occupied by live objects. If it's known that the "live" heap occupation is never higher than α, then R will always be sufficiently large. However, this is only theory because, in practice, obtaining the fraction α is too difficult. So, in our implementation, R is computed at the beginning of every GC cycle and its value is sufficiently high to guarantee that *this* cycle finishes in time.

5 The virtual machine

The development of our virtual machine was done in two stages. The first machine is very simple. The second is optimized so that the byte-code generated by the compiler is much smaller. We will use the first one for most of the explanations because it's 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 7 shows the index and the name of the instructions of the machine. Some instructions have many indices. 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 are specified by "blocks to jump over" and "position in the block" pairs of operands. The second operand is omitted in certain cases: when the designated block has only one variable, the second operand can be assumed to be 0.

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 7: Instructions of the first virtual machine.

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

- $\mathcal{C}[\![$ ⟨exp⟩ $]\!]$
- Write variable ⟨operand_1⟩ [⟨operand_2⟩]
- Restore continuation

Figure 8: Compilation rule for `set!` in terminal position.

The compilation rules are quite straightforward. The only part that is a little more sophisticated is the set of rules for calls. It all depends on what we know about the operator: we know nothing, or it's a kernel function, a closure from the library, or the direct result of a lambda-expression. Figure 8 shows one of the compilation rules. The $\mathcal{C}^*$ and $\mathcal{C}$ functions are the compilation functions in terminal and non-terminal position, respectively.

5.2 The final machine

The final virtual machine has a different instruction set. We don't present every detail, just the main classes of modifications to the first machine:

- Specialized instructions. Some original instructions are almost always used with the same operands. In these cases, we created instructions that are specialized for those typical operands. For instance, 90% of the local variables that are read are located in one of these locations: `(0, 0)`, `(0, 1)`, `(0, 2)`, `(1, 0)`, `(1, 1)`, and `(2, 0)`.
- Merged instructions. Some instructions always precede or follow 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.
- New instructions; such as "Pop the first block from the environment".
- 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 isn't desired.

This new virtual machine allows the byte-code to be much more compact. We have two benchmarks: one is a little program that forces the inclusion of all the library; the other is a module taken from a bigger project (a parser generator).

When they are compiled for the first virtual machine, they both produce about 10 500 bytes of byte-code. When they are compiled for the second machine, they produce about 5400 and 5600 bytes, respectively. Also, it demonstrates that our entire R^4RS Scheme library fits in less that 5.5 KB of byte-code, which is surprisingly compact.

6 The actual implementation

Although our goal was to demonstrate that a Scheme implementation for the 68HC11 is possible, we actually made no tests directly on the chip.

6.1 The byte-code compiler

The byte-compiler is able to handle all the subset of Scheme that we wanted to support. We didn't really take care of making it fast. Its speed is reasonable, though. In only 8 seconds, it's able to compile Aubrey Jaffer's test file for Scheme implementations [10] while it's interpreted by the `scm` interpreter [9] which is running on a 150 MHz DEC Alpha. The test file is a 27 KB source, in which we commented out the sections concerning I/O, *bignum*, and *flonum* functions.

Only very minor additions to the byte-compiler are required to adapt to a real microcontroller. In fact, it only needs additional primitive function declarations in the library corresponding to the addition of primitive functions in the kernel. Those functions are necessary to actually control the chip.

6.2 The runtime system

A weakness of our current runtime is that it doesn't proceed with the creation of the constants the way it's described in Section 2.3: it doesn't discard the description of the constants. So, when the executable runs, it keeps both the description *and* the constants themselves.

Also, the runtime kernel is written in C. Our system requires a C compiler that produces executable code for the microcontroller. As we mentioned above, additional kernel functions are required to give to the executable the means to control the chip. Still, it shouldn't be that much work. It would be a more important piece of work to translate the kernel into assembly language in order to obtain an even more compact executable.

Implementation		Size of interpreter
`fools 1.3.2`	[12]	288 KB
`minischeme 0.85`	[13]	95 KB
`scm 4e1`	[9]	368 KB
`siod 3.0`	[14]	166 KB
`bit` (byte-code interpreter with full library)		72 KB

Figure 9: Size of different small Scheme implementations.

6.3 Experiments

We made a test off-chip to verify if it is possible to fit our executables on the 68HC11. We used a modified `gcc` compiler [11] that produces code for the 68HC11. The code that it produces is poor. The main problem seems to be that `gcc` expects many registers on the target machine. The 68HC11 has only one all-purpose register. So the back-end has to pretend that there are enough registers and has to simulate their existence using cells in memory.

Still, we obtained an executable of 22 KB for the kernel and the complete library. Even if they are far from ideal, the results allow us to conclude that integration into the 68HC11 is already possible. The same experiment on an H8 (a microcontroller with more registers used in the Lego Mindstorms robot) gave a 15KB executable.

We compared the size of our executables on a DEC Alpha workstation with other "small" implementations. Figure 9 shows the results. The only implementation whose size is close to ours is `minischeme`. But this implementation is far from being R^4RS compliant. These comparisons aren't necessarily fair, though, because the other implementations are interactive interpreters. The `bit` byte-compiler fits in 72 KB with the full library but it doesn't include an evaluation function to perform interaction. It is possible that an adaptation of one of the other interpreters to an off-line version might give good results.

Our implementation performs poorly when it comes to time efficiency. It's roughly 10 times slower than `scm` and 5 times slower than the Gambit interpreter (`gsi` [15]). While we took care of the space-efficiency aspects, we didn't bother about the 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 quite inefficient (see [6]). Second, we don't 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, our virtual machine keeps the arguments of a call in a list. It means that a pair must be allocated for each argument. Given that memory management is slow, this process becomes pretty heavy. Finally, the concise style in which the library is written adds to the inefficiency. Higher-order functions are intensively used, even in many apparently basic operations.

6.4 Improvements

This work could be extended in many ways:

- Drop the unnecessary machinery that rebuilds the allocated constants. If no constants of a certain type have to be rebuilt, the construction code specific to this type becomes useless. Also, when it's possible, the description string of the constants should be dropped after decoding.
- Drop the symbol names when possible. Sometimes, only the identity of the symbols is required, not their name.
- Add other number representations. From the most useful to the least: flonums, bignums, complex, rationals.
- Provide a better implementation of environments. Environment representations that are tailored to the local needs of the Scheme expressions would be preferable.
- Improve the time efficiency.
- Provide the user with flags to give him control of the inclusion of features and declare properties about his program.
- Use various analyses well known in speed optimization areas, but that can be put to contribution in space optimization areas too. Such analyses include flow analyses (see [7]), dead code analyses, representation analyses, useless-variable detection, and storage use analyses.

7 Conclusion

Our goal was to determine whether it's possible to program microcontrollers such as the 68HC11 in Scheme. The two major constraints concern size and real-time-ness of the implementation. In order to obtain a small implementation, we took advantage of the non-interactivity 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. Choices concern: run-time representation of Scheme objects like 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 didn't compromise the asymptotic complexity of Scheme programs.

Our results clearly demonstrate that it's feasible to program microcontrollers in Scheme. Scheme sources, once compiled, become byte-codes several times smaller. The two biggest weaknesses are the low speed of the execution, about 10 times slower than one of the fastest Scheme interpreters available, and the poor performance of the C compiler that translates the runtime kernel to microcontroller machine code.

References

[1] R. A. Brooks. 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*, pages 108–113, aug 1984.

[2] W. Clinger and J. R. (editors). *Revised^4 Report on the Algorithmic Language Scheme*, 1991.

[3] D. Dubé, M. Feeley, and M. Serrano. Un gc temps réel semi-compactant. In *Actes des Journées Francophones des Langages Applicatifs 1996*, jan 1996.

[4] M. Feeley and G. Lapalme. Closure generation based on viewing lambda as epsilon plus compile. *Journal of Computer Languages*, 17(4):251–267, 1992.

[5] D. Gudeman. Representing type information in dynamically typed language. Technical Report TR 93-27, Department of Computer Science, The University of Arizona, oct 1993.

[6] M. Larose and M. Feeley. A compacting incremental collector and its performance in a production quality compiler. In *Proceedings of the International Symposium on Memory Management (ISMM-98)*, volume 34, pages 1–9, 1998.

[7] O. Shivers. The semantics of scheme control-flow analysis. In *Proceedings of the Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 190–198, jun 1991.

[8] P. R. Wilson. Uniprocessor garbage collection techniques. *Lecture Notes in Computer Science*, 637:1–42, sep 1992.

[9] Author: Aubrey Jaffer.
`ftp://ftp.cs.indiana.edu/pub/` —
— `scheme-repository/imp/scm4e1.tar.gz`

[10] Author: Aubrey Jaffer.
`ftp://ftp.cs.indiana.edu/pub/` —
— `scheme-repository/code/lang/test.scm`

[11] Author: Otto Lind.
`ftp://wattson.ee.ualberta.ca/pub/` —
— `motorola/68hc11/gcc/gcc-6811-fsf.tar.gz`

[12] Author: Jonathan Lee.
`ftp://ftp.cs.indiana.edu/pub/` —
— `scheme-repository/imp/fools.1.3.2.tar.gz`

[13] Author: Atsushi Moriwaki.
`ftp://ftp.cs.indiana.edu/pub/` —
— `scheme-repository/imp/minischeme.tar.gz`

[14] Author: George Carrette.
`ftp://ftp.cs.indiana.edu/pub/` —
— `scheme-repository/imp/siod-3.0.tar.gz`

[15] Author: Marc Feeley.
`ftp://ftp.iro.umontreal.ca/pub/` —
— `parallele/gambit/gambc.tgz`