

Hedgehog: A Tiny LISP for Embedded Applications

Lars Wirzenius
liw@iki.fi

Kenneth Oksanen
cessu@iki.fi

Feb 27th, 2005

Abstract

Hedgehog is a very concise implementation of a Lisp-like language for low-end and embedded devices. It consists of a compiler to byte code and a corresponding interpreter. The byte code interpreter is written in standard conforming C, is efficient and easily portable, and can be compiled to a very small executable of only some 20-30 kilobytes.

The Hedgehog Lisp dialect has proper support for first class functions, lexical scoping, variable argument functions, garbage collection, exceptions, macros, and over a hundred predefined functions or special forms. The built-in types are lists, symbols, strings, 32-bit integers, AVL-trees, and tuples up to 16 elements wide. Proper 32-bit wide integers are necessary for various bit-level operations in embedded systems.

Hedgehog was developed by Oliotalo, a company specializing in machine-to-machine communication. It has used Hedgehog for such diverse telemetric applications as sensing whether trash compactors are full and need to be emptied, whether tanks of gas or liquid are empty and need to be filled, and reporting the position and usage patterns of various kinds of vehicles. Hedgehog is used to measure things locally and to send the data over wireless networks (SMS, GPRS, Bluetooth) to a server.

In this paper, we discuss why Oliotalo decided to develop Hedgehog, what requirements on the implementation different applications put on Hedgehog, what it is like to write embedded applications in Lisp, how Hedgehog has been implemented, and what positive and negative experiences we have had with Hedgehog.

1 Introduction

Embedded software development is today done mostly in the C programming language. The benefits of higher level languages would also benefit embedded development. Traditionally these benefits are: faster development, fewer bugs, easier modification, and portability.

Modern embedded hardware is capable of running modern operating systems, such as Linux, and also interpreted programming languages. This implies that porting or writing a high level programming language is realistic even if there are not enough resources to compile to native machine code. Byte code interpretation is sufficiently fast for many if not most applications. Experimenting with a high level language therefore requires only a little bit of effort.

Most programming language implementations are quite big, however. For example, on a Debian GNU/Linux PC, Perl takes up 11 megabytes and Python 9.4 megabytes, although much of that is not required for an embedded system. Even the stripped down version of Perl in Debian that is used for installation takes 1.9 megabytes. Although modern embedded hardware is fast enough to execute Perl scripts with adequate speed, spending megabytes of space on flash may be too wasteful.

Other small Lisps and Schemes somewhat similar to Hedgehog do exist [1, 2], but naturally with different emphases and features. And as far as we know, Hedgehog is the only one actively maintained and distributed as free software.

2 Overview

Oliotalo implements systems that contain machine-to-machine communication or telemetry. In a typical system, there are some number of small devices that measure something or control some equipment. The devices communicate wirelessly with a server. The users can then get reports from one central location. As an example, the devices might measure how full trash compactors are and the user is responsible for emptying the full ones. The system allows users to optimize their operation so that they don't empty a compactor before it is almost full.

In the fall of 2002, Oliotalo decided to develop a small Lisp interpreter for its own use for developing such embedded applications. The two primary goals were to improve productivity by using a higher level language, and to make it easier to update the applications after delivery.

Some embedded computers Oliotalo uses are running a proprietary op-

erating system that is statically linked to the application and then loaded onto flash via a serial port. Updating the application in devices that are already installed in the field is tedious, time-consuming and expensive: you have to visit each device with a laptop. Yet, such updating is required since customers often want changes to their system.

Thus, Hedgehog was born. It consists of an embedded interpreter for a custom byte code, a compiler to translate application code written in Lisp to byte code, and a Lisp function library. When the application in a device needs to be updated, only the byte code file is actually replaced. From the operating system's point of view, it is just data and no linking or screwdrivers are necessary. Oliotalo regularly updates devices over GPRS and Bluetooth networks.

Only such operations are built into the interpreter that must be implemented in C (bindings to operating system calls) or that give significant performance improvements by being implemented in C. On the Lisp level, all these operations look like any other function calls.

When Hedgehog is ported to a new hardware platform, the set of built-in functions may need to change. One hardware platform might have a CAN bus, another might replace it with an I2C bus. The core language does not change, however, so applications that don't use either bus will work on both platforms. Adding new built-in functions is easy and quick, so adapting to new platforms does not take a lot of time.

To support bootstrapping a device, the interpreter can be compiled with a bootstrap application that it runs if it cannot find the real application.

Lisp was chosen for the language because it is relatively simple to implement and the implementation can be made rather concise, yet the language is quite powerful. Since application development productivity was a primary goal, the language had to be quite expressive and Lisp fills that requirement well.

Lisp also has garbage collection, which is even more important in embedded programming than in server or desktop programming. Memory leaks hurt a lot when there is only a little memory.

We chose to omit data mutation primitives from our Lisp dialect and aimed towards a purer functional language to help avoid problems with data corruption in the device. Pointer problems and unintended data mutation are common programming errors. They can be quite difficult to debug at the best of times, but especially so when the only method of debugging is through a slow serial port.

Programming language implementation sounds like a formidable task, but for a small, interpreted language it was assumed the task would be manageable. Having one's own implementation gives flexibility and occasionally a

competitive edge. It is then possible to adapt the language easily to the hardware and operating system platform. The language can also be designed to the programming task, if necessary.

3 The Language

The Lisp dialect of Hedgehog is a custom one that is not directly compatible with any of the existing ones, such as Common Lisp or Scheme. For example, it does not have primitives for in-place mutation of data and the syntaxes for function definitions, `cond`, and `let` are different. Having a custom dialect gives flexibility to design the language especially for embedded programming. This allows the interpreter to be very small, a little over 20 kB in its smallest configuration on an x86 Linux.

The other big reason to not implement Scheme or Common Lisp was that Wirzenius, who did the initial design and implementation, did not know either, and it was deemed more efficient time-wise to design a new dialect than to learn an existing one and remove all unnecessary parts from it. Also, writing in a subset of a well-known language can be more tedious than in a new one that is similar to an existing one.

We will illustrate the flavor of the language through several code examples without assuming previous exposure to Lisp. To start with, the classic "hello, world" program in Hedgehog looks like this:

```
(pr "hello, world")
```

`pr` is a function to print values to the "log file", which may be either a dedicated serial port in a dedicated device, or the standard output when running under a Unix-like system. In Lisp, parentheses surround the function name and its arguments. Hedgehog does not have a debugger so debugging is done by printing out values in suitable places.

The simple data types in Hedgehog are 32-bit signed integers, strings, and symbols. The usual combined data type is a list. The following function computes the length of a list:

```
(def (list-len list)
  (def (helper list length)
    (if (nil? list)
        length
        (tailcall (helper (cdr list) (+ length 1))))))
(helper list 0))
```

The function `nil?` tests whether its argument is the empty list or the null pointer, and `cdr` returns a list that is the same as its argument, but without the first argument.

This example also demonstrates the use of local functions. The function `helper` is local to the function `list-len`, meaning that its name is visible only inside `list-len`. Hedgehog Lisp uses tail recursion to implement loops. The compiler implements tail recursion the same way it would implement loops so there is no speed or space impact.

It is, however, easy to mistakenly write non-tail recursion especially when the surrounding code contains macros. To make sure tail recursion is used when intended, the compiler provides the `tailcall` construct. It does not affect program execution, but the compiler will give a fatal error if a call is not a tail call.

To make sure the program works correctly, simple unit testing is provided.

```
(fail-unless-equal (list-len nil) 0)
(fail-unless-equal (list-len '(1)) 1)
(fail-unless-equal (list-len '(1 2)) 2)
(fail-unless-equal (list-len '(1 2 3)) 3)
```

List traversal is a common operation and as such it is a good candidate for abstraction. Indeed, the Hedgehog library contains the function `accumulate` that will visit each item in a list by calling a user-supplied function and maintaining a running value from one item to the next, returning the final value. `list-len` can be written using `accumulate`:

```
(def (list-len list)
  (def (helper so-far item)
    (+ so-far 1))
  (accumulate helper 0 list))
```

`helper` is called for each item and gets two arguments: the current running value and the current item in the list. It returns the new running value. `accumulate` gets three arguments: the function `helper`, the initial running value, and the list. Since `helper` always adds one to the running value, the final value will be the length of the list.

New list nodes are created using `cons`. A list node (also known as a cons cell) consists of two values: the value in that node and a pointer to the next node (`nil` in the last node of the list). The following code creates a list of numbers of a desired length:

```

(def (numbers count)
  (def (helper remaining list)
    (if (eq? remaining 0)
        list
        (helper (- remaining 1)
                 (cons remaining list))))
    (helper count nil))

```

Note that a list can only be grown at the beginning, since an existing cons cell may not be modified.

There is no way to explicitly free memory, but unused memory is automatically garbage collected. This applies to all data: numbers, strings, cons cells, and so on.

4 The Library

The Hedgehog library resides completely on the compiler side. The compiler includes the parts of the library the application uses in the byte code file. This increases the size of the byte code file, but makes it much more flexible to change the library. As far as the interpreter is concerned, it is just byte code data.

The library contains functions for simple math, string manipulation, data structures (lists, queues, and AVL-trees), state machines, and more. For better performance we have added one special byte code instruction for AVL-trees: given a key, value, and left and right subtrees whose height differs with at most two, construct a new AVL-tree node and perform necessary rotations to bring the new tree into balance.

System calls are easily introduced to Lisp by writing a corresponding byte code instruction and Lisp builtin function definition. For a system call like UNIX `open` this takes a little over ten lines of C, including dynamic type checks and document strings. That code is rather generic and portable to all UNIXes, but a more significant problem is caused by the more fluid factors such as flag values and C structure layouts. In order to introduce them to Lisp, we employ a little perl program, which reads in a list of flag, structure and field names and generates a C program. The C program consists of corresponding statements which test (with `#ifdefs`) whether the specified features are defined, and writes out a Hedgehog Lisp definition for the flag name and its value, structure and field sizes, field offsets, and byte order. When building Hedgehog, that C program is (cross-)compiled, emulated if possible, and its output is copied to the installed Lisp library.

Note that because of the kind of applications we write, not all UNIX system calls are supported: for example, since we assume the only user in the small embedded box is root, a system call like `chown` would be useless.

Hedgehog is strictly single-tasking. Hedgehog applications may simulate multi-tasking by structuring the application as a set of co-operating state machines that send messages to each other or wait for timeouts or input from an external source. The interpreter then executes one state machine at a time, for a while, and then switches to the next one. These state machines and their execution are implemented purely in Lisp. Because the language does not allow in-place mutation, the state machines are safely isolated from each other.

Each state is implemented by a two-argument function which takes as input the state machine's private data and the delivered message, and returns a list of actions, such as messages to be sent to other state machines, a request to block or transit to a new state. In practice states are defined by a little macro which hides many dirty details. For example the state `open-file` below tries to open a fifo in `/tmp/FIFO`. On failure it retries after 5 seconds, on success it transits to state `read-fd` passing it the newly opened file descriptor:

```
(def-state (open-file _ msg)
  fd (unix-open "/tmp/FIFO" (| unix-O_RDONLY unix-O_NONBLOCK))
  (wait (< fd 0) _ nil nil 5000000)
  (goto read-fd t fd))
```

The symbol `_` is habitually used to indicate an ignored value. The state `read-fd` might proceed as follows: ignore messages caused by timeouts and entering the state, try to read up to 256 bytes from the fifo, on failure go to a state which closes the file, on success send the read bytes to a state machine called `logger`, and re-enter the same state when there is more data to read or when five seconds has passed.

```
(def-state (read-fd fd msg)
  (wait (or (eq? msg 'enter) (eq? msg 'timeout))
        fd (list fd) nil 5000000)
  data (unix-read fd 256)
  (goto close-file (or (nil? data) (eq? data "")) fd)
  (send 'logger data)
  (wait t fd (list fd) nil 5000000))
```

Because states are described in such an orderly fashion, the `def-state` macro can be implemented so that it produces a graph definition suitable for layout by the Graphviz tools.

5 The Implementation

Because the interpreter is ran on a possibly very constrained system, we moved as much functionality as possible from the interpreter to the compiler. For example, there is no interactive read-eval-print-loop, all macros are expanded during compilation time, and the language is lexically scoped. Because stack space can be limited to a few kilobytes and because tail recursion removal is mandatory in a long-running application, we employ a byte code instruction dispatcher instead of a recursive evaluation of the parse tree. The immediate value field in the byte code instructions is of variable length in order to reduce byte code program size.

The virtual machine abstraction provided by the byte code interpreter contains a stack, the corresponding stack pointer, and an `accu` register which caches the value of most recently evaluated expression. `accu` reduces traffic to and from the stack and saves a little bit of computation as well as executable and byte code program size. Consequently `accu` is used also for passing the last argument in a function call and returning the result of the call. Other arguments and the return address are stored on stack. The virtual machine contains also a number of other, more dedicated registers, such as the program counter, the current environment, and a new environment being constructed when instantiating a lambda function with free variables.

Language implementations with garbage collection must usually allow the collector to distinguish pointer values. A standard solution is to indicate the type of a word by its few lowermost bits - two zeros would usually then imply a properly aligned pointer. The higher bits are then free for other purposes. But because Hedgehog Lisp programs must study and control the machines otherwise programmed with typical C code, Hedgehog Lisp programs must be able to manipulate proper 32-bit integers. Many systems choose to store integer values in heap-allocated objects, but this tends to consume excess memory and computations. We solved this problem by keeping integers from -2^{30} to $2^{30} - 1$ in the 31 upper bits of the word. The integer value can then be recovered by arithmetically shifting the word one right. Otherwise the integer is stored in heap.

In surprizingly many cases it was possible to know already when implementing the byte code instructions that an integer would be in the range -2^{30} to $2^{30} - 1$ and we could consequently use the simpler integer handling code. For example, UNIX file descriptors, `errno` values, character values, list and string lengths can all be represented by such short integers. Of course, when testing the Lisp programs on desktops we hade run-time checks that ensured the integers indeed were short.

Knowing that the byte code programs are executed in small devices al-

lowed us to assume that not all 32 bits would be used to represent pointers. In fact we assumed proper word alignment and limited the maximum heap size to 64 MB — still more than hefty for our target devices — and thereby freed several bits in the pointer to indicate some type information of the referred object: for example, whether it is a tuple, how wide a tuple it is, and whether the object is in the constant pool submitted in the byte code program. Consequently a cons-cell consumes exactly two words in heap, and a large integer value in heap consumes exactly one word.

Because Hedgehog programs use Lisp strings to represent C structs, nul characters may not terminate a string as in C. Instead we use a header word to tell the length of the string, but we also have an additional nul character so that the strings in the Lisp heap can be passed to C functions and system calls without any conversion.

While most of the techniques presented above are not new, they serve as an example of how the assumptions of the target devices lead to a certain design which would be suboptimal for some other purposes. Smallness must be designed, it is not a matter of optimization after implementation. In fact, after our initial rather straight-forward implementation we were not able to reduce executable size by more than a third. Gotos were considered embarrassingly useful in that game.

The garbage collector is currently a simple stop© collector, which undeniably incurs a significant memory overhead. But fortunately all constant data is stored in the byte code program image, not in the garbage collected heap, thereby clearly reducing memory consumption. And someday we nevertheless may shift to a more memory-conserving garbage collector.

Since some embedded devices have standard C libraries of rather depressing quality, the byte code interpreter tries to use it as lightly as possible. For example, Hedgehog comes with its own version of printf, memory allocation is limited to from four to six malloc calls, and calls to possibly non-standard (read: broken) implementations of library functions are hidden behind macros such as HH_MEMMOVE so that they can be easily replaced with calls to better ones.

The compiler emphasizes optimizations that reduce code size, such as unreachable code elimination, constant folding of integer and boolean values, and using as small immediate field lengths as possible. We have also created a handful of aggregate instructions for very frequently occurring tasks (such as "add immediate value to accu", instead of the three instructions "push accu to stack", "load immediate value" and "add") and written a corresponding peephole optimization pass to create such aggregate instructions.

The compiler does not support separate compilation, but in practice this has never been a problem: compiling the whole standard library and thou-

sands of lines of application code typically happens in a blink of an eye. Separate compilation could also prevent some optimizations and language constructs.

6 Experiences

Although Oliotalo has not performed measurements, anecdotal evidence is ample that Hedgehog has greatly improved embedded application development speed. There are at least two reasons for this.

Firstly, a higher abstraction level, including automatic garbage collection, lets programmers concentrate more on application logic than on tedious details. This results in less code and fewer bugs.

Second, most applications can be tested on a desktop Linux machine, which takes less time than testing on an embedded device. Bugs are also easier and faster to find.

An added benefit for Oliotalo has been that more people can do embedded development. C programming in resource-poor environments without hardware protection against pointer problems is much harder than writing Lisp code.

Although typical C programming mistakes (pointer errors, memory leaks) are avoided, Hedgehog does introduce some new errors. Since the language is dynamically typed, type errors and errors in the number of function arguments are common. These are mostly caught by unit testing, if that is used properly, but it would be preferable for the compiler to catch such errors.

It is also fairly easy to run out of memory, often by creating large temporary values and not killing references to them quickly enough. A typical case would be to parse a string by deleting one character at a time from the beginning. Since strings (like other values) are immutable, each deletion results in a new slightly shorter copy of the string. This situation is best avoided by careful coding, and is usually easy to fix when found, but better tools for this is clearly needed. Live-precise garbage collection would have helped in many cases.

The big lesson learned several times during Hedgehog development has been that abstractions should be done at the Lisp level, not inside the interpreter. This applies particularly to hardware and operating system interfaces. For example, if the hardware provides a CAN bus, the interpreter should provide built-in functions for sending and receiving messages, but it should not build a transaction and data transfer facility. Implementing those on the Lisp level gives much more flexibility and keeps the interpreter smaller, simpler, and less likely to contain errors.

While Hedgehog Lisp has been a great enabler from C, assembly language or PLC programming, there is still room for significant improvements. Our plans include at least optional static typing, more libraries, better remote debugging facilities, better string processing and parsing tools, and reduced memory overhead from garbage collection. Perhaps we'll also do what the original LISP developers, despite their original intent, didn't have time to accomplish: a more sugared syntax.

7 Conclusions

For Oliotalo, developing Hedgehog has certainly been worthwhile. Hedgehog has worked well, and most of the effort has been gone into integrating it with the operating system and hardware platforms, particularly those that are not based on Linux. That work would have been required with any other programming language implementation in any case.

In fact, in our experience, implementing a small custom language is fairly easy and requires a modest effort. There are significant benefits from being able to design things to be small rather than trying to force a program — or even worse, a specification — written for large computers into a small device.

Customers have not cared whether Lisp is used or not. They have only cared about the results, and being able to change specifications with relative ease even after delivery and deployment has sometimes been a killer argument.

References

- [1] Rodney A. Brooks, Charles Rosenberg. *L – A Common Lisp for Embedded Systems*. Lisp Users and Vendors Conference, Sec. 2.4a, Aug. 1995.
<http://www-2.cs.cmu.edu/~chuck/pubpg/luv95.pdf>
- [2] Danny Dubé. *BIT: A very compact Scheme system for embedded applications*. Workshop on Scheme and Functional Programming, Montreal, 2000, page 35.
<http://www.ccs.neu.edu/home/matthias/Scheme2000/dube.ps>