Elk Scheme Extensions for Multimedia Programming

J. P. Lewis

NEC C&C Research
Princeton, NJ 08540 USA
zilla@ccrl.nj.nec.com

Abstract

Multimedia programming has diverse requirements, including interactivity, complexity management and rapid development facilities, high numerical performance, and compatibility with C. While Lisp and Scheme address most of these requirements, the design decisions in many Lisp and Scheme implementations make them unsuitable for the development of high performance mixed-language applications. This paper describes several extensions to the Elk implementation of Scheme which make it a suitable base for rapid prototyping of multimedia applications under Unix. The extensions include a foreign function interface, C-compatible data types, and vector and parallel computation facilities. The resulting system has been used in both high-performance numerical computation and in the construction of Unix utilities. The extensions have been released to the original author of Elk and will be included in Elk version 2.0.
KEYWORDS: Multimedia, Scheme, vector programming, Linda.

Introduction

The goal of this work was to obtain a development environment suitable for multimedia work. Multimedia applications involve several distinct styles of computation. In the user interface,

- object oriented techniques, interactive development, and other approaches to managing the complexity of the user interface are needed
- multi-threaded control is desirable—the interface should not 'die' when an operation is started
- efficiency is not critical; languages such as Lisp and Smalltalk have adequate performance.

In contrast, the 'back end' computation of multimedia data types is similar in character to scientific computation:

- Supercomputer-class performance is required for real-time operation. The bandwidth of digital video, for example, is roughly 15 megabytes/second, so real time operation entails 15 MIPS per algorithm instruction. A simple signal processing algorithm would require hundreds or thousands of MIPS to operate in real time on video data.
- The algorithms are very data parallel and are well suited to vectorization and parallelization.

Lastly, in the graphics portion of a multimedia application we may require mixed language programming in order to make use of existing GUI libraries such as X-windows or the Silicon Graphics GL library. These libraries generally have C language interfaces.

Languages such as Lisp, Scheme, and Smalltalk can in theory support the multiple programming styles required in multimedia computation due to their extensibility. Scheme in particular is a simple but highly configurable language. While Scheme is neither object oriented nor multi-threaded, it is possible to build sophisticated multiple inheritance object systems with little effort and no change to the language [3]; coroutines are also easy to implement.

World Views: Lisp versus Unix

Lisp, Scheme, and Smalltalk may have practical drawbacks when used with Unix-like operating systems, [1] however. Most implementations adopt what could be called the 'Lisp Machine' world view; the implementation is intended to provide a complete environment and as such effectively replaces the underlying operating system from the user's point of view. While the Lisp Machine world view has advantages, a practical drawback is that it does not make effective use of existing Unix software (and hardware). Complete lisp environments may require many megabytes of memory and may take many seconds to start; such implementations are not appropriate for writing a Unix filter program, for example.

The design decisions of such Lisp Machine-like implementations favor efficient execution of Lisp code; the foreign function interface is generally added as an afterthought. This means that calling foreign code is often relatively inefficient and inconvenient. For example, it is often necessary for the foreign code to convert data from the internal Lisp format to the C storage format. Existing foreign library routines may not be directly callable for this reason.

Elk Scheme

Elk (Extension Language Kit) is a mostly Revision 3 compatible Scheme interpreter for Unix written by Oliver Laumann. It is designed to serve as a user and run-time extension language, that is, to be bound into large C (or other) language programs in the same way that Emacs has a Lisp

[1] In this paper Unix does not refer to UNIX, which is a proprietary trademark. Rather, Unix is intended to refer to a Unix-like operating system having a command shell user interface.

extension language. Elk has several features which support its use as an extension language:

- Dynamic loading of C object files. Extensions to Elk can be written in C (or a C-calling convention compatible language) and can be dynamically loaded into the running interpreter. Without our foreign function interface (see below), such extensions must understand the Elk calling convention.
- Definition of new types from C. Elk extensions can define new Elk types by providing a few routines to create, destroy, print, copy, etc. objects of the new type.

The Elk distribution includes glue (calling convention conversion) code for many X windows and Motif routines, and it includes some sample Motif widgets written in scheme. Elk is free and is available by anonymous ftp from `nexus.yorku.ca`.

Numerical Computation in Lisp and Scheme

Some existing Lisp and Scheme compilers generate code which is competitive with that generated by C compilers for certain classes of programs (e.g. [2]). Unfortunately, few Lisp or Scheme compilers handle numerical (and in particular floating point) computation well. Our Elk Scheme extensions include three alternate approaches to obtaining high-performance numerical computation in a Lisp language:

1. A C foreign function interface. Using the rule of thumb that most compute time is spent in a few inner loops, we expect that we can obtain nearly the performance of a pure C program while writing only a few routines in C.
2. Vector primitives, or "APL in Scheme". The relative inefficiency of Lisp or Scheme can be diluted by overloading the computation on large vectors.
3. Support for parallel computation.

Foreign Function Interface

The Elk foreign function interface extension (hereafter referred to as FFI) obtains Scheme data, puts it in registers or on the stack according to the C calling convention, calls a foreign function, retrieves any returned results, and packages these results as Scheme data. Because the FFI must access machine registers including the stack and floating point registers, and because C (despite its reputation as a 'systems programming' language) cannot access such registers directly, the FFI must in general be written in assembly language. The FFI currently exists for the Sparc and MIPS/SGI architectures.

The FFI allows the following C types to be passed from/to Scheme: `float`, `double`, `char`, `short`, `int`, `char *`, `FILE *`. The types `char,short` are always passed as a four-byte `int` due to C type elevation rules on 32-bit architectures. Functions which return `char *` result in the creation of a Scheme string (allocated on the Scheme heap). This works correctly for C functions such as `getenv` which return a pointer to a statically allocated string. Routines which return a pointer to a string allocated with `malloc` will leak memory if called with our FFI; such routines need to be called via a glue function which frees the string. Scheme Ports are passed as C `FILE` streams. Scheme Booleans are passed to C as the integers zero or one.

Foreign functions are defined by the call

```
Define_Foreign(char *name,
               void (*fun)(), char *args)
```

where `name` is the name of the new function in Scheme, `fun` is the address of the C function, and `args` is a string specifying the argument types of the C function. Characters in the `args` string specify argument types as follows:

```
B Scheme boolean <-> C int 0 or 1
I integer
F double
f float
S string (char *)
P port (FILE *)
A foreign array (see below)
```

The character `R` precedes the data type of the return value (if any). Examples:

```
Define_Foreign("strlen",strlen,"SRI");
Define_Foreign("getenv",getenv,"SRS");
Define_Foreign("pow",pow,"FFRF");
```

The argument specification `"SRI"` for the function `strlen` can be read as "String, Returns Integer".

C structure datatypes can be used in Scheme programs in one of two ways: one can define C routines to create, initialize, print, free, and otherwise operate on the datum. These routines are declared via the FFI. The 'create' routine returns a pointer to the allocated object and passes this address to Scheme as an integer. Alternatively, the create routine can be written in Scheme and can allocate space in a foreign array (see below). In this approach Scheme routines can be written to access the fields of the structure.

The FFI allows a hybrid Scheme/C programming style in which program subroutines are written in either Scheme or C as appropriate. The choice of Scheme versus C coding is of course a program design decisions which must be considered carefully. More generally, the FFI itself reflects a design compromise. We feel that the current design is justified by its simplicity and by the fact that it does not require the use of unusual coding styles or reference to Scheme internal data structures in the foreign code. A more extensive FFI design is described in [4].

Foreign Arrays

Foreign arrays (farrays) are essentially a mechanism for allocating storage from the Scheme heap for use by foreign routines. Foreign arrays appear to Scheme as a normal datatype, i.e., they are assigned, printed, and (if unreferenced) garbage collected like other variables.

Foreign arrays are conventionally interpreted as homogeneous arrays of one of the C datatypes `char, int, float.` Objects of other type are mapped onto `char` (byte) foreign arrays. Arrays of `double` are not currently implemented. Foreign arrays are distinguished from Scheme (or Lisp) vectors or arrays in that the latter are inhomogeneous arrays of pointers to arbitrary data and thus cannot be passed to C without conversion.

A hypothetical example of the use of a foreign array is a three-dimensional graphics program. This program reads the geometry of a model from a file into a foreign array using a foreign call to `read` or `fread`, transforms the geometry using vector calls (described below), and writes the transformed data to a display library. Overall control is in Scheme, but all of the computation except the calls is in C, and no data conversion between Scheme and C is required.

Foreign arrays are created as the value of the call

```
(farray <type> <length>)
```

where `type` is one of `'integer,'real,'string` (`string` signifies a byte array) and `length` is the number of elements in the array.

The following Scheme functions manipulate foreign arrays:

```
(farray? <farray>)
(farray-type <farray>)
(farray-length <farray>)
(farray-copy <farray>)
(farray-of <values>)
(farray-ref <farray> <index>)
(farray-set! <farray> <index> <value>)
```

In order to avoid adding new syntax to Scheme, farrays are printed as

```
(% <values >)
```

and the function `%` (also known as `farray-of`) returns its arguments as an farray.

Elk Scheme as a Unix Shell Language

The FFI made it convenient to bind many Unix library functions in Elk. With the addition of a few custom glue routines Elk now serves as a sophisticated shell language.

The motivation for Scheme as a shell programming language can be contrasted with the tool building philosophy underlying Unix. In the Unix philosophy tools are effectively domain specific "little languages". `csh, make, awk, find` are examples of tools having relatively complex (and ad hoc) input languages.

While the "little language" approach provides concise commands for accomplishing intended tasks, it also has a number of disadvantages:

- 'Cognitive overhead'–it requires that the skilled user be familiar with many inconsistent special purpose languages.
- Little languages do not have the support of 'bigger' languages. Such support includes debuggers, alternate implementations, extension languages, and professionally written manuals.
- Poor design–many little languages are collections of incrementally added features; the resulting language is poorly designed. Typical symptoms include single-character variable names, bizarre escape sequences, and numerous reserved special characters.
- Poor synergy–it is common to find that one tool has information which is needed, but it is in a form which cannot be used by a different tool. In fact much of the effort in writing Unix shells (and utility programs) is involved simply in parsing and reformatting data representations.
- Limited reusability– program components such as argument and input parsing, symbol tables and expression evaluation are rewritten from scratch many times.

An alternate approach, which might be termed the *extensible language* philosophy, adopts a complete and extensible language. This approach allows one to build reusable libraries and modules, and to extend existing tools without starting from scratch. Most importantly, this approach lets us reuse our own knowledge rather than requiring the learning of a different syntax for each tool. These preceding assertions will now be illustrated with several examples.

The function `traverse` walks through a Unix directory tree, calling the supplied function `dofile` with the name of each file in the tree:

```
(define (traverse path dofile)
  (if (equal? 'regular (file-status path))
      (dofile path)
      ; else
      (if (file-exists? path)
  (traverse-dir path dofile))
  );if
);traverse

(define (traverse-dir path dofile)
  (let ((dir (os-read-directory path))
        (back (os-getwd)))
    (os-chdir path)
    (dolist (entry dir)
    (if (not (member entry '("." "..")))
                (traverse entry dofile)))
    (os-chdir back)
  );let
);traverse-dir
```

Traverse can be saved in a separate file and loaded by any shell which requires this function. Thus, a shell to removes all core files from one's home directory is:

```
#! /usr/local/bin/elk

(traverse (os-getenv "HOME")
  (lambda (f)
    (when (equal? f "core")
          (os-delete-file f)
    )
  ))
```

Of course the Unix program `find` is designed to do this sort of thing. `Find` has its own syntax, however, and thus it lacks the "cognitive economy" of our function. One should also consider the probable effort required to develop `find` versus that required to write the traverse function– the latter is probably under an hour of effort despite having similar utility. Since the source for `traverse` is available and is simple, it can be easily modified to accomplish new tasks.

With a few suitable library functions such as `traverse`, Elk shell programs can subsume the roles of many special purpose Unix tools. Elk shell programs also serve well in the role of a general purpose shell language. While this is difficult to demonstrate in limited space, we may compare the Elk and Perl versions of a subroutine to add two vectors:

```
sub arrayadd {
  local(*a, *b) = @_;
  local($max) = $#a > $#b ? $#a : $#b;
  local(@sum);
  for (local($i) = 0; $i <= $max; $i++) {
    $sum[$i] = $a[$i] + $b[$i];
  }
  @sum;
}
@foo = (1,2,3);
@bar = (10,20,30);
@totals = &arrayadd(*foo, *bar);
```

The corresponding Scheme is:

```
(define (arrayadd a b)
  (map + a b))

(define foo '(1 2 3))
(define bar '(10 20 30))
(define totals (arrayadd foo bar))
```

The Perl code has the typical "little language" symptom of excessive syntax. This example requires the characters `*,$,#,@,&` as well as parentheses, brackets and curly braces.

Elk Vector Facility

Many scientific and multimedia algorithms have significant or overwhelming 'data parallelism', meaning that the same operation is applied across many pixels, sound samples, or other data. In contrast to the identification of parallelism in a compiler or in a typical AI program, data parallelism is very easy to identify.

Such parallelism can be 'exploited' in a language where primitive functions operate directly on the aggregate data, rather than on one datum at a time. Such primitive functions can be mapped directly onto vector or certain parallel hardware [5].

Scheme has several advantages over C for data parallel programming. Algorithms expressed as C programs cannot easily make use of such parallelism, since C programs are in a sense already 'compiled' *by the programmer* for the serial machine described by C (i.e., a PDP-11). Scheme is extensible: any function can be redefined with vector overloading. Scheme also has "first class syntax": user-defined extensions to the language are indistinguishable in form and convenience from intrinsic features.

A subset of APL-like vector operations has been added to Elk Scheme. These include scans, reductions, gather/scatter/subscripting, reshaping, and various mathematical functions. Since the functions are written in C, the performance overhead of Scheme is diluted by the size of the vectors. In several simple benchmarks, vectorized Scheme code was found to be $1/5 \ldots 1/3$ of the speed of the same benchmark written in custom C code. We note that this is in the same performance range as current Lisp compilers, despite the fact that Elk is an interpreter.

Vector Expression Compiler

Rather than overloading functions with vectors (as in APL), we took the approach of adding a new form `parlet` (data parallel let) which compiles serial code in its body onto the vector operations. One motivation for this approach is that the `parlet` form serves to identify vectorized code in the program. A second motivation is that the APL approach is difficult to compile because the scalar/vector (scalar meaning non-vector) nature of a variable is determined at run-time. While Elk is interpreted, we wanted to write code which could also be compiled using an appropriate redefinition of `parlet`. The `parlet` form declares all vector variables and all unknown vector functions. The `parlet` form resembles the `elwise` form used in Paralation Lisp [5]; both forms are unusual for Lisp-like languages in that they make a type distinction explicit.

The syntax of `parlet` is

```
  (parlet <declarations> <body>)
```

As a simple example,

```
(parlet (v)
  (= v 1))
```

declares `v` to be a vector and returns the result of comparing 1 to each element of this vector. `Parlet` determines the vector/scalarness of each expression and elevates all scalar components of a mixed expression to vector form. Scalar functions are replaced by the corresponding vector operators if appropriate. Thus, the previous example expands to

```
(v-eq v (v-distribute 1 v))
```

All vector operators are functional, though `set!` can be used to assign vectors to variables inside a `parlet`. The scalar $\rightarrow$ vector expansion happens at the time the function is defined, not at run time. Writing scalar code in the body of the `parlet` enhances the readability of the code, particularly for those who are not familiar with APL and similar languages.

Support for Parallel Computation

`POSYBL`, a public domain implementation of the Linda parallel programming paradigm [1], has been included in Elk.

Our current Scheme binding consists of three primitives `linda-in,linda-rd,linda-out`. All tuple keys and data are Scheme strings. `Linda-in` and `linda-rd` accept a key string and retrieve the corresponding data; `linda-out` is passed a key string and the corresponding data. A fourth call `linda-run` runs a Unix process on a remote host; that process is typically a worker which will access the shared tuple space.

It is possible to build a coarse-grain version of the Linda `eval` on top of these calls. The eval call should print the expression to be evaluated to a string, add a unique identifier to this string, and put the identified string into the tuple space, i.e,

```
(linda-out "eval"
  (to-string (list (make-id) expr)))
```

An "eval server" process calls `linda-in` with the same key ("eval"), reads and evaluates the expression, and prints the result back into the tuple space using `"linda-out"` with another agreed-upon key and the identifier:

```
(let* ((r (from-string (linda-in "eval")))
       (id (car r))  (expr (cdr r)))
  (linda-out "result"
    (to-string (list id (eval expr))))
)
```

Using such a `linda-eval` mechanism and the Scheme force/delay construct, we were able to implement a `pseudofuture` form resembling the *future* construct in several parallel Lisp and Scheme dialects. Unlike the *future* construct, however, we must explicitly *force* our results.

It should be emphasized that the facilities described in this section are suitable only for very coarse grain computation. The basic Linda facilities have been used in a short three-dimensional animation involving the computation of a three-dimensional correlated random process. The computation was distributed across 15 workstations using the processor farm paradigm, with a single frame of the animation as the unit of granularity. Each frame required several minutes of computation, and the total computation animation required on the order of a few hours rather than a day or so thanks to the parallel computation.

References

[1] N. Carriero and D. Gelernter. How to write parallel programs: A guide to the perplexed. *ACM Computing Surveys*, 21:323–357, September 1989.

[2] D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams. Orbit: An optimizing compiler for scheme. In *Proc. SIGPLAN Symposium on Compiler Construction*, pages 255–263, 1986.

[3] Kurt Normark. *Simulation of Object-Oriented Concepts and Mechanisms in Scheme*. Aalborg University Institute for Electronic Systems, Aalborg, Denmark, 1990.

[4] John Rose and Hans Muller. Integrating the scheme and c languages. In *Proc. ACM Lisp and Functional Programming*, 1992.

[5] Gary Sabot. *The Paralation Model: Architecture-Independent Parallel Programming*. MIT Press, Cambridge, Mass., 1988.