ecc: the ELI-to-C Compiler

User's Guide

December 2, 2013

1. Introduction

ecc is an ELI-to-C compiler which translates an ELI program into ANSI Standard C code. The translated C code can then be compiled by a C compiler to produce machine code executable outside the ELI environment. In most cases, the resulting C executable so produced runs much faster than the corresponding ELI source code runs under ELI interpreter. However, the speedup of a compiled ELI program over the interpreted version greatly depends on the style of the program being compiled. For an ELI programmer who intends to deliver his application developed in ELI for outside use, the ability to distribute object code provides a mechanism for basic protection. This also makes a piece of code developed in ELI much easier to be integrated into a large C/C++ based application.

The ***ecc*** compiler comes with substantial restrictions: it basically can only compile ELI programs using flat-arrays. We will detail these in the next section. We intend to remove some of these restrictions in future versions of *ecc*. In addition, we plan to incorporate automatic parallelization for multi-core machine capability into *ecc* in the near future. We note that while iterative ELI programs gain most in speedup from compilation, only ELI programs written in array-oriented style will benefit from automatic parallelization.

Section 3 will describe how to prepare ELI code for compilation, the compilation procedure and how to run the compiled code in various platforms. No extra declarations are required for compiling ELI code, and in general no modification to the ELI source code is needed. No knowledge of C is required to compile or run compiled code other than typing in required command lines.

2. Compiler Restrictions

A *compilation unit* (***cu*** for short) in ELI is a group of ELI defined functions with a main function `fmain` and all defined functions called directly or indirectly by `fmain`. A ***cu*** is translated by the compiler into a single C source file. A ***cu*** must be *self-contained* in the sense that any functions in it cannot access global variables but the main function can take one or two parameters each of which can be a list of nested arrays. We recall that ELI provides a system variable `[]IO`, the index origin, which can be 0 or 1. One restriction is that one `[]IO` value must be chosen for the whole ***cu***. From here on, when we say a compiler restriction, it applies to all (defined) functions in a ***cu***.

1. nowhere the execute function !. can be used.
2. other than the parameter(s) to the main function, variables must be scalars or arrays (see ELI Primer).
3. currently, the data of type symbol, complex numbers and temporal data cannot be used*.
4. no system variable or system functions can be used.
5. `[]` input are not allowed.
6. the type and rank of a variable must not change thru all paths of execution.*

If a variable in a function violates the last restriction, it is usually the case that the same variable name has been used for, in a real sense, two different variables. Hence, this violation can be avoided by renaming the variable at the point where type/rank change takes place. We will soon remove restriction 3 as well as part of 4 by allowing `[]CR`, `[]TS`. There is an exception to restriction 2 for the main function `fmain` in a *cu*. Suppose

```
larg fmain rarg       (*larg can be empty*)
```

is the headline of `fmain`, then `larg/rarg` can be *one-level nested lists*, i.e. each can be of the form `(v1;..;vn)` where for each `vk,k=1,..n,` is either a scalar or an array.

3. Preparing an ELI Program for Compilation

To prepare a program, i.e. a compilation unit or *cu*, for compilation, one put all functions in that *cu* into a ELI *.esf file, say `cmain.esf` which can be just an ordinary *.txt file with file extention *esf* or a file resulting from

```
)out cmain
```

in an ELI workspace (see [1]). We note that it is ok for `cmain.esf` to contain functions not reachable from the designated main function `fmain` of the *cu*, i.e. outside the cu. They just do not get compiled.

In addition to the function text included in `cmain.esf`, one needs to put two variables `lparm` and `rparm`, in `cmain.esf`. These are the left and right arguments to the main function `compile` in *ecc*. The type designations of a scalar or array in ELI are the following:

```
boolean: 'B', integer:'I', floating point: 'E', character: 'C'
```

`lparm` is a character string whole initial portion is the name of the main function of the *cu*, then followed by two characters indicating the types left and right parameters to the main function in the cu. For example,

```
lparm<-'fmain CI'
```

if the type of the left parameter to `fmain` is character and the type of the right parameter to `fmain` is integer. In case the left or/and the right is a list, then the type indicators of the component elements of this list are presented with surrounding parentheses. For example, suppose the right parameter above is a list whose components have types boolean, floating point and integer, then we put the following in `cmain.esf`

```
lparm<-'fmain C(BEI)'
```

If `fmain` has no left parameter then the part indicating its type is empty, and in case it has no parameters at all, i.e. *cu* has no input, `lparm` is simply `'fmain'`.

`rparm` is a vector of integers of the form

qdio lshape rshape

where ***qdio*** is the desired `[]IO` of the *cu*, ***lshape*** and ***rshape*** are the *shape elements* of the left and right parameters to the main function (as for `lparm` one or both shape elements can be missing). A *shape element* is defined as follows: a scalar is indicated by a 0; arrays are indicated by their rank followed by their shape. Any dimension of the shape vector can be unknown at compile time, and is indicated by a `_1` in that case. Thus a vector is specified as

```
1 v1
```

where v1 is the length of the vector, and `1 _1` for a vector of unknown length. A matrix is specified as

```
2 d1 d2
```

where `d1` and `d2` are the first and second dimensions of the array with `_1` indicating unknown. In case one (or both) parameter(s) is a one-level list, then in place of one shape element there is a sequence of shape elements each indicating the shape of a list element which must be a scalar or an array. Note that, unlike in the case of type specification, no parentheses are needed or allowed in `rparm`. For example, for `fmain` above, suppose its left argument is a character vector of length 16, and the right argument is a list of a single bit, a floating point vector of unknown length and a 2 by variable length integer matrix; then we put

```
rparm<-0 1 16 0 1 _1 2 2 _1
```

in the `cmain.esf` assuming `[]IO` is 0 for the *cu*.

4. Compiling and running Compiled code

One first downloads the ***ecc*** compiler package from the [download] section at site http://fastarray.appspot.com/. The platform choices for the package are Windows, Linux and Mac OS. Unpack, and put the file into a designated directory. The file should contain the *ecc* executable. Type

```
./ecc -u
```

to check compiler usage.

Now assume a source code `cmain.esf` has been prepared as in sect.3 and has been put in the same directory. The command line to *compile* this code is the following

```
./ecc cmain
```

Once that line has been executed, if the ELI *cu* satisfies all the restrictions listed in sect.2, a C file `fmain.c` will appear in the same directory as that of *ecc* (`fmain` is the name of the main function in the *cu*). In case compilation is not successful, the compiler will emit appropriate error messages to help the user to modify his ELI source code, and one just re-attempt the compilation process. In Windows, the prefix '*./*' in the command line should be dropped, or one can use another way to compile by double click on the icon denoting *ecc* and fill in the name of `cmain.esf` in a box.

The next step is to compile the translated C code. Suppose we use the *gcc* compiler. First `cd` to where the translated C code resides, and dependent files *Apl3lib.h, Apl3lib.cpp, Aplc.h* and *elimacros.h* should be there too. Type

```
gcc -o fmain fmain.c
```

Again, in Windows one can double click on `fmain.c` and choose compile option. This will produce an executable file `fmain`.

To run that executable file in Windows, Linux or Mac OS, we need to prepare `inmain.lef` and `inmain.rig` data files representing the input data for the left and the right parameters to the function `fmain` in `cmain.esf`. (again, if there is no left argument, then `inmain.lef` is not present). The format of one of the files looks like below:

```
E 2 2 3
1 2 3 4 5 6
```

where the first line specifies the type and shape element of the argument, and from the second line on, these are the raveled elements of the argument itself. The one above represents a 2 by 3 floating point matrix `2 3#!6`.

The compiled code is executed by issuing

```
fmain inmain
```

or

```
fmain    (* by default fmain takes fmain.lef and fmain.rig as input data files *)
```

One can prepare another set of input files `inmain1.lef` and `inmain1.rig` and run

```
fmain inmain1
```

to get different result.

One convenient way to prepare the input files is to export a variable in an ELI workspace, and modify the exported file's content slightly as well as change its file extension. Suppose we already have a variable `mat` corresponding to the example above in an ELI session or workspace, then do

```
)out inmain mat
```

will generate a file `inmain.esf` in the directory `Program files/eli/ws` of your machine; if the system is directory not Windows the output file can be found in the same directory as that of `eli`. Its content is

```
&mat E 2 2 3
1 2 3 4 5 6
&
```

We'll take `&mat` out of the first line and eliminate the last line (`&`), and rename its file extension to `lef` or `rig`. In case of character data, the quote character ` in the beginning and the end of data should also be removed. Of course, input data can come from other sources, one only need to add the extra top line specifying type and shape.

There are three examples in paper [2] at http://fastarray.appspot.com/. They are *blackscholes*, *hotspot* and *k-means*. All these examples come from standard benchmarks, been rewritten in Eli and compiled by ***ecc***. One can follow the ReadMe in the download package to verify the workflow of the whole procedure.

References

[1] A Primer for ELI: a system for programming with arrays, last revision 2013.

[2] An ELI-to-C Compiler: Self-compilation and Performance, 2013.