Using Portable CLU

Stephen Garland
Dorothy Curtis

February 10, 1992; Updated November 29, 2016

This document provides a short explanation of how to compile and link CLU programs to produce executables.

1 Overview

A CLU program consists of one or more modules, also known as *abstractions*. The *interface specification* for an abstraction completely describes how clients (i.e., other abstractions) see the abstraction and how they can use it. The interface specification for a procedural or iteration abstraction is determined by the header for that procedure or iterator; the interface specification for a data abstraction (i.e., a cluster) is determined by the header for the cluster together with the headers for the operations named in the cluster header. The implementation of each abstraction is theoretically invisible to its clients.

Generally, the code for each module is kept in a separate file, the name of which ends with `.clu`. Sometimes it is convenient for several modules to employ common declarations for compile-time constants, e.g., $maxLen = 100$ or $intSeq = seq[int]$. Such "equates" are generally kept in a separate file, the name of which ends with `.equ`.

2 Using the Compiler

There are three steps in compiling a CLU program. First, we create an interface library that contains the interface specifications of all of the modules that will make up the program. Second, we compile the modules against this library. (Portable CLU differs from earlier native CLU compilers in that the interfaces of all modules must be extracted into an interface library before any module is compiled. Earlier compilers were more relaxed about this requirement.) And third, we link the resulting object files together into an executable program.

The default of the pclu compiler is to produce optimized code. Use the `-debug` flag consistently to each pclu invocation if debugging is desired.

2.1 Creating a type library

Syntax

```
pclu -spec <source>.clu ... -dump <file>.lib
```

Overview

Checks the interface specifications of the specified source files and dumps the resulting interface information into the library file `<file>.lib`

Options

`-ce <foo>.equ`	Use equates from specified file
`-use <foo>.lib`	Read type info from library before compiling

Comments

Multiple `.lib` files and equate files can be specified.

2.2 Compiling CLU files

Syntax

```
pclu -compile <source>.clu ...
```

Overview

Generates `<source>.o` for each specified CLU file `<source>.clu`.

Options

`-debugt`	Generate output for debugging
`-opt`	Generate optimized output (the default)
`-ce <foo>.equ`	Use equates from specified file
`-merge <foo>.lib`	Read type info from library before compiling

Comments

Multiple `.lib` files and equate files can be specified.

2.3 Linking a program

Syntax

```
plink -o <program> [<object>.o] [<object-library>.a] ...
```

Overview

Links the specified `.o` and `.a` files together and creates executable `<program>`. `.o` files are generated by pclu (and various other compilers). `.a` files contain a number of different `.o` files and can be generated via the `ar` program.

Options

`-debug`	Generate a program for debugging
`-opt`	Generate an optimized program

2.4 Spec-checking CLU files

Syntax

```
pclu -spec <source>.clu ...
```

Overview

Spec checks each source file `<source>.clu`

Options

`-ce <foo>.equ`	Use equates from specified file
`-merge <foo>.lib`	Read type info from library before compiling

Comments

Multiple `.lib` files and equate files can be specified.

2.5 Examples

The first example demonstrates how to compile a CLU program with all code in a single file. We first need to create a type library for the program.

```
pclu -spec factorial.clu -dump factorial.lib
```

Now we compile the file against the type library created during the last step.

```
pclu -merge factorial.lib -compile factorial.clu
```

Now link the program together.

```
plink factorial factorial.o
```

Suppose we have a multiple file program with an equate file that we wish to compile with optimization enabled. Here are the three pclu invocations that achieve that goal.

```
pclu -ce define.equ -spec main.clu support.clu -dump prog.lib
pclu -merge prog.lib -ce define.equ -compile main.clu support.clu
plink -o prog main.o support.o
```

2.6 A More Complicated Example

Suppose someone else implemented some abstractions and provided us with an interface library and object files for those abstractions. Suppose the supplied interface library is called `ps9.lib`, the supplied object files are `graph.o` and `tree.o`, and all of these files are stored in some directory `<dir>`.

We write code that uses these supplied abstractions and put our code in two CLU files, `main.clu` and `spanning.clu`. In addition, we create an equate file `x.equ` containing some common abbreviations.

Before we can compile our program, we need to create an interface library for it. We do this by using the supplied interface library and checking the two CLU files we wrote. (The backslash in the following text is a shell convention that indicates that the next line is part of the command started on this line.)

```
pclu -merge <dir>/ps9.lib -ce x.equ main.clu spanning.clu -dump my.lib
```

Now we can compile our CLU files against the interface library we just generated. Note that we do not need to mention the original interface library in this step because the contents of that interface library were copied into our library by the first step. However, the equate file is specified for all compiler invocations (except the last linking phase). This will be true both of equates files we write and of equate files supplied by somebody else.

```
pclu -merge my.lib -ce x.equ -co main.clu spanning.clu
```

Finally, we link everything (including the supplied object files) together.

```
plink -o program main.o spanning.o <dir>/graph.o <dir>/tree.o
```

3 Using Makefiles

This section describes how to write makefiles for CLU programs that are intended to be compiled by the portable CLU compiler. If you don't know what a makefile is or what the program `make` does, then skip this section. You can come back to it when you know more about `make`.

Makefiles can be used to automate the process of compiling CLU programs. This can be useful for two reasons. First, programs like the one in the example given above are quite complicated. Automating the compilation process can reduce the tedium and complexity of typing in long compilation commands over and over again.

Second, using `make` can also help avoid unnecessary recompilation. Because of the way the portable CLU compiler works, it is somewhat complicated to use this feature of `make` to its full extent. Therefore, these notes describe only a simple, but not the most efficient, use of `make` to avoid recompilation.

Let us construct a makefile to perform the task described in the previous example, where some code is supplied by someone else. We describe this task in a top-down manner. The main goal of the makefile is to produce the executable `program`. The following rule describes how to generate `program` from the `.o` files on which it depends.

```
program: main.o spanning.o <dir>/graph.o <dir>/tree.o
          plink program main.o spanning.o <dir>/graph.o <dir>/tree.o
```

If `main.o` and `spanning.o` exist, then `make` will produce `program` by linking the specified object files together. Otherwise, it will have to generate `main.o` and `spanning.o`. (We assume the supplied object files `<dir>graph.o` and `<dir>tree.o` already exist so that the makefile does not have to create them). The following rules generating these `.o` files.

```
main.o: main.clu my.lib
        pclu -merge my.lib -ce x.equ -co main.clu

spanning.o: spanning.clu my.lib
        pclu -merge my.lib -ce x.equ -co spanning.clu
```

Now we have to tell `make` how and when to create `my.lib`.

```
my.lib: main.clu spanning.clu
        pclu -merge <dir>/ps9.lib -ce x.equ \
             -spec main.clu spanning.clu -dump my.lib
```

Finally, we combine all these rules and put them into a file named `Makefile`.

```
program: main.o spanning.o
        plink -o program main.o spanning.o \
                 <dir>/graph.o <dir>/tree.o

main.o: main.clu my.lib
        pclu -merge my.lib -ce x.equ -co main.clu

spanning.o: spanning.clu my.lib
        pclu -merge my.lib -ce x.equ -co spanning.clu

my.lib: main.clu spanning.clu
        pclu -merge <dir>/ps9.lib -ce x.equ -co main.clu spanning.clu \
             -dump my.lib
```

The command `make` issued to the shell will perform any necessary recompilations and produce an executable file called `program`.