Using Portable CLU

Sanjay Ghemawat
Stephen Garland
Dorothy Curtis

February 10, 1992

This document explains how to use portable CLU 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. The topmost procedure should be named `start_up`.

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 several 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. Then we compile the modules against this library. (Portable CLU differs from earlier non-portable CLU compilers in that the interfaces of all modules must be extracted into an interface library before any module is compiled with portable CLU. Earlier compilers were more relaxed about this requirement.) Finally we link the resulting object files together into an executable program.

Keep in mind that the format of interface libraries and object files produced by portable CLU is different from that produced by earlier compilers. Therefore, you should not mix libraries and object files generated by different compilers.

If the program should allow debugging, then the `-debug` flag should be supplied consistently to each PCLU invocation. If the program should be optimized, then the `-optimize` flag should be supplied to each PCLU invocation. (Certain kinds of PCLU invocations ignore `-debug` and `-optimize` flags because they do not generate any code. However, users are allowed to specify `-debug` and `-optimize` on all PCLU invocations for consistency.)

If the option `-verbose` is specified, PCLU prints out each command before executing it. This is usually only useful to system administrators.

Section 3 describes how to use PCLU to generate an executable program from CLU source files.

3 Syntax

3.1 Creating a type library

Syntax

```
PCLU -create <file>.lib -spec <source>.clu ...
```

Overview

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

Options

`-debug`	Generate output for debugging
`-optimize`	Generate optimized output
`<foo>.lib`	Read type info from library before compiling
`<foo>.equ`	Use equates from specified file
`-use <foo>.lib`	Read type info from library before compiling (Obsolete)

Comments

`-spec` is implied by default and can be omitted. `-debug` and `-optimize` flags are ignored. Multiple `.lib` files and equate files can be specified.

3.2 Compiling CLU files

Syntax

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

Overview

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

Options

`-debug`	Generate output for debugging
`-optimize`	Generate optimized output
`<foo>.lib`	Read type info from library before compiling
`<foo>.equ`	Use equates from specified file
`-use <foo>.lib`	Read type info from library before compiling (Obsolete)

Comments

`-compile` is implied by default and can be omitted. At most one of `-debug` and `-optimize` should be specified. Multiple `.lib` files and equate files can be specified.

3.3 Linking a program

Syntax

```
PCLU -link <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 output for debugging
`-optimize`	Generate optimized output

Comments

At most one of `-debug` and `-optimize` should be specified.

3.4 Spec-checking CLU files

Syntax

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

Overview

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

Options

`-debug`	Generate output for debugging
`-optimize`	Generate optimized output
`<foo>.lib`	Read type info from library before compiling
`<foo>.equ`	Use equates from specified file
`-use <foo>.lib`	Read type info from library before compiling (Obsolete)

Comments

`-debug` and `-optimize` flags are ignored. Multiple `.lib` files and equate files can be specified.

3.5 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 -create factorial.lib factorial.clu
```

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

```
PCLU factorial.lib -compile factorial.clu
```

Now link the program together.

```
PCLU -link factorial factorial.o
```

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

```
PCLU -debug -create prog.lib define.equ main.clu support.clu
PCLU -debug prog.lib define.equ -compile main.clu support.clu
PCLU -debug -link prog main.o support.o
```

3.6 A More Complicated Example

Suppose somebody else implemented some abstractions and provided me 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>`. I write code that uses these supplied abstractions and put my code in two CLU files `main.clu` and `spanning.clu`. In addition, I create an equate file `x.equ` containing some common abbreviations. First I need to create an interface library for the whole program. I do this by using the supplied interface library and also checking the two CLU files I 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 -debug <dir>/ps9.lib x.equ -create my.lib main.clu spanning.clu
```

Now I can compile my CLU files against the interface library I just generated. Note that I do not need to mention the original interface library in this step because the contents of that interface library were copied into my 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 I write, and equate files supplied by somebody else.

```
PCLU -debug my.lib x.equ main.clu spanning.clu
```

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

```
PCLU -debug -link program main.o spanning.o <dir>/graph.o <dir>/tree.o
```

4 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 different reasons. First, the example given above was 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 sometimes help us avoid unnecessary recompilation. Because of certain peculiarities in CLU compilers, this feature of `make` cannot be utilized to its full extent for CLU programs. Therefore, I will not stress the use of `make` to void recompilation. I will focus more on the automation of the compilation process.

Let us try and write a makefile to perform the task described in the previous example where some of the code was supplied by someone else. I will describe this task in a top-down manner. The main goal of the makefile should be to produce the executable `program`. The files required to generate `program` are `main.o` and `spanning.o`. (I will assume that the supplied object files will always exists and therefore my makefile does not have to create them).

```
program: main.o spanning.o
        PCLU -debug -link 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`. So let us write down the rules for generating these files.

```
main.o: main.clu my.lib
        PCLU -debug my.lib x.equ main.clu

spanning.o: spanning.clu my.lib
        PCLU -debug my.lib x.equ spanning.clu
```

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

```
my.lib: main.clu spanning.clu
        PCLU -debug <dir>/ps9.lib x.equ -create my.lib \
             main.clu spanning.clu
```

We can combine all these rules and put them into a file named `Makefile`.

```
program: main.o spanning.o
        PCLU -debug -link program main.o spanning.o \
                <dir>/graph.o <dir>/tree.o

main.o: main.clu my.lib
        PCLU -debug my.lib x.equ main.clu

spanning.o: spanning.clu my.lib
        PCLU -debug my.lib x.equ spanning.clu

my.lib: main.clu spanning.clu
        PCLU -debug <dir>/ps9.lib x.equ -create my.lib \
             main.clu spanning.clu
```

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