Gencot User Manual

Gunnar Teege

January 10, 2020

Chapter 1

Introduction

Gencot (GENerating COgent Toolset) is a set of tools for generating Cogent code from C code.

Gencot is used for parsing the C sources and generating Cogent sources, antiquoted C sources, and auxiliary C code. It does not perform a fully automatic translation, it is intended to be used in combination with several manual steps of pre- and post-processing. These steps are described in this manual.

The manual assumes that you are familiar with C and Cogent and know how to work with both.

1.1 Distribution

The Gencot distribution consists of the following folders:

manual this manual,

bin the main command scripts `gencot` and `parmod` and many auxiliary commands used by them,

include Cogent include files used by the generated code,

c C code implementing abstract types and functions used by the generated code,

examples example C programs used in this manual for introducing Gencot,

src the Haskell source code of Gencot components,

doc a comprehensive documentation of Gencot design and implementation.

Gencot is a command line tool. To use it make sure that you can invoke the commands `gencot` and `parmod` (e.g., by linking them in a folder in your command path or by adding the `bin` folder of the Gencot distribution to your command path.

Additionally you have to set the environment variable `$GENCOT_HOME` to the root folder of the Gencot distribution.

We also assume that you have a working distribution of Cogent and can invoke the cogent compiler using the command `cogent`.

All example folders contain a UNIX Makefile. You can either run the examples by manually typing the commands to process them or by using `make` with a separate target for each step.

1.2 First Encounter

As usual we will start with a "Hello World" example. Go to `examples/helloworld`. Ignoring the other files, look at `hello.c`. It contains the C program

```
#include <stdio.h>

int main() {
  puts("Hello World");
}
```

Step 1: (make run)

Try it: compile the program with a C compiler, name it `hello` and run it. It should do what you expect.

Step 2: (make cogent)

Now use Gencot to translate the program to Cogent. Enter the command

```
gencot cfile hello.c
```

It creates the file `hello.cogent`. Look at it. It contains

```
cogent_main : () -> U32
cogent_main () =
   0
   {-
       cogent_puts("Hello World");
   -}
```

This is a Cogent function corresponding to the C function `main`. However, the function body is still C code and put in comment. Instead, the dummy result `0` is used, so the file is already valid Cogent code.

The command also creates the file `hello-entry.ac` (Not yet implemented! Provided with the example).

Next enter the command

```
gencot unit unit.files
```

It creates three files `unit-externs.cogent`, `unit-exttypes.cogent`, and `unit-dvdtypes.cogent` where the latter two are empty and can be ignored for this example. The first file contains

```
cogent_puts : (CPtr U8)! -> U32
```

which is also valid Cogent code, but uses the non-standard generic type `CPtr`.

The command also creates the file `unit-externs.ac` (Not yet implemented! Provided with the example).

Step 3: (make edit)

Now comes the part where your manual work is demanded. You have to translate the function bodies and you have to adapt some types. Open `hello.cogent` in a text editor, replace its content by

```
cogent_main : () -> U32
cogent_main () =
       cogent_puts("Hello World"); 0
```

and save as `ed-hello.cogent`. In `unit-externs.cogent` replace the argument type by `String`:

```
cogent_puts : String -> U32
```

and save as `ed-unit-externs.cogent`.

Now you have a Cogent program which is equivalent to the original C program. It consists of several files, which are all included by the provided file `unit.cogent`.

Step 4: (make cogent-c)

To see that it works, process `unit.cogent` by the Cogent compiler. Enter the command

```
cogent -o gen-unit -g unit.cogent --infer-c-funcs="unit-externs.ac hello-entry.ac"
```

It creates the files `gen-unit.c` and `gen-unit.h` and two `_pp_inferred.c` files. All these files are included by the provided file `unit.c`, so this file wraps together the C program generated from the Cogent program.

Step 5: (make cogent-run)

Try whether it still works. Compile `unit.c` with a C compiler, name it `cogent-hello` and run it. The result should be the same as in Step 1.

Note that for the compilation you need to set the include path to the standard library folder of the Cogent distribution (`STDGUM` in the Makefile).

Step 6: (make clean)

Clean up all generated files. If you like you can perform the steps again.

Chapter 2

Simple C Programs

2.1 Single Source File

The simplest case is a C program which consists of a single `.c` file, such as the "Hello World" program introduced in Section 1.2. If the C file is named `foo.c` the command to process it by Gencot is

```
gencot cfile foo.c
```

It creates the following files:

`foo.cogent` The content of `foo.c` translated to Cogent, as far as Gencot supports a translation. Function bodies are not translated.

`foo-entry.ac` (*Not yet implemented*) "Entry wrapper" functions for all functions defined in `foo.c` with external linkage.

An "entry wrapper" converts from the original C function API to the Cogent function API, which is usually different since Cogent functions always take one argument and return one value. Functions in `foo.cogent` are automatically renamed so that they do not collide with the entry wrappers after translation back to C by Cogent. For a standalone C program there is atleast an entry wrapper `main` for the translated main program.

Additionally, Gencot must be invoked using the command

```
gencot unit unit.files
```

where `unit.files` contains the name of the C source file (`"foo.c"`) in a single line. It creates the following files:

`unit-externs.cogent` Abstract definitions of all external functions used by the C program.

`unit-exttypes.cogent` Type definitions for all external types used by the C program.

`unit-dvdtypes.cogent` Type definitions for all derived types used in the C program.

`unit-externs.ac` (*Not yet implemented*) "Exit wrapper" functions for all functions defined in `unit-externs.cogent`.

An external function is used by the C program if it is actually invoked. Such functions must be declared in C, usually the declarations are contained in standard include files. In the "Hello World" example the file `hello.c` includes `stdio.h` where function `puts` is declared. Here, Gencot reads `stdio.h` to generate the abstract definition for `gencot_puts` and the corresponding exit wrapper. Although there are many functions declared in `stdio.h`, Gencot does not generate definitions for them, since they are not invoked by the C program.

An external type is a type defined in a standard include file. Again, Gencot provides a Cogent type definition in `unit-exttypes.cogent` for only those types which are actually used by the C program. A derived type is a pointer type, an array type, or a function type in C. For some of them Gencot generates auxiliary type definitions in `unit-dvdtypes.cogent`. In the "Hello World" example both files are empty.

An "exit wrapper" is similar to an "entry wrapper", it converts from the Cogent function API back to the C function API. Exit wrappers are also renamed so that they do not collide with the originally invoked C function.

All generated files are named according to the name of the file argument to the `gencot` command. If the file is named `foo.files` you will get `foo-externs.cogent`, `foo-exttypes.cogent` etc.

2.2 Single Source File with Include Files

Often, a C program consists of a `.c` file with function definitions and an included `.h` file for data type definitions.

2.3 Multiple Source Files

2.4 Partial Translation