Gencot User Manual

Gunnar Teege

January 14, 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 Rationale

Normally, Cogent is intended to implement from scratch a program which both compiles to a C implementation and can be used as a high-level specification for formally verifying semantic properties. Cogent generates a formal proof that the C implementation is a refinement of the high-level specification, thus all proven semantic properties also hold for the C implementation.

Gencot is intended for the case where there already exists a legacy C implementation for a task. To apply Cogent as described above, the task must be re-implemented in Cogent. Gencot supports this re-implementation by automatically translating several parts of the legacy C program to Cogent code. This should help developers to start with the re-implementation, especially if they have more experience in C programming than in Cogent.

However, Gencot does not provide the following:

- a guarantee, that the translation to Cogent is equivalent to the legacy C program. There is no formal proof for such an equivalence and Gencot itself is not verified. Also, a part of the re-implementation must be done manually, so no automatic evidence can be provided.

  If you are lucky, there is a test suite for the legacy C program. Then you can apply the test suite to the Cogent re-implementation and get some evidence, that there is no difference in relevant behaviors. Even if there is no test suite, building on the legacy C program should be easier and faster than to start the re-implementation from scratch. In any case, the formal verification starts at the Cogent program and does not extend to the legacy C program.

- a guarantee that the translation to Cogent is type-safe and respects the constraints of the uniqueness type system. Gencot translates the types as they are specified in the C program. It maps some memory safety properties to the Cogent type system, but it does not make unsafe parts safe. In particular, if the legacy C program shares and discards pointers, so will the translation to Cogent.

  The advantage is that the Cogent typechecker will statically detect all theses cases and signal them as an error, so that the developer can concentrate on them. The Cogent program must then be refactored to avoid such cases, until it can be successfully processed by the Cogent compiler.

- a guarantee that the translation to Cogent is useful for formally verifying the desired semantic properties. The Cogent program generated by Gencot has the same overall structure as the legacy C program. If this structure is not adequate for formal proofs, e.g., because it massively depends on reading and modifying a global state, the generated Cogent program (or the legacy C program) must be refactored to better support formal verification.

1.2 Gencot 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`, `auxcog` 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.

1.3 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 the file `unit.cogent` and 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. File `unit-externs.cogent` contains

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

which is also valid Cogent code, but uses the non-standard generic type `CPtr`. File `unit.cogent` includes all generated Cogent files and is the main source file to be processed by the Cogent compiler.

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. In `unit-externs.cogent` replace the argument type by `String`:

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

and save.

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

Step 4: (make cogent-c)

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

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

It creates the files `unit-gen.c` and `unit-gen.h` and two `_pp_inferred.c` files.

Gencot is required to perform some postprocessing steps. Enter the command

```
auxcog unit unit.files
```

It creates the file `unit.c` which includes `unit-gen.c` and the other files generated by `auxcog`. It wraps together the C program generated from the Cogent program by the Cogent compiler and the Gencot postprocessing step.

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) and to the `c` directory in the Gencot distribution. The latter contains the files `gencot.h` and `cogent-common.c` which are included by `unit.c`

Step 6: (make clean)

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

Chapter 2

C Program Structure

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 1.3. 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. Using the entry wrappers, all translated C functions can still be invoked from a C program according to their original API.

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`.

`unit.cogent` The main Cogent source file which includes all other Cogent sources.

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.cogent`, `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. In this case Gencot must additionally be invoked for translating the `.h` file in the form

```
gencot hfile foo.h
```

It will create the additional file `foo-incl.cogent` which contains the translated content of `foo.h`. The include directive in `foo.c` will automatically be adapted, so that `foo.cogent` includes `foo-incl.cogent`. So the resulting Cogent program has a similar source file structure as the C program.

If there are several `.h` files included by the `.c` file gencot must be invoked for every `.h` file separately.

The other files are translated as before, however, the path must be specified where to look for the included files, even if they are in the same directory. It is specified with an option `-I` as for `cpp`. Several such options may be specified if included files are in different directories. Include paths may be specified relative to the current directory. Do not append a slash `/` at the end. Paths for system include files cannot be specified, the `cpp` default is used for accessing them.

Together, the commands for translating the C program are

```
gencot hfile foo.h
... <other .h files> ...
gencot -I. cfile foo.c
gencot -I. unit unit.files
```

where `unit.files` still contains only the name `foo.c` and not `foo.h` (because that is automatically read together with `foo.c` which includes it).

2.2.1 Example

The example program in `examples/cards` consists of the file `cards.c` and the included files `cards.h` and `rank.h`. The tranlation step (`make cogent`) corresponds to the commands

```
gencot hfile cards.h
gencot hfile rank.h
gencot -I. cfile cards.c
gencot -I. unit unit.files
```

Note that the function declarations in the `.h` files are not translated to Cogent. However, the comments specified there are moved to the function definitions in the translation of the `.c` file. This feature depends on the order of processing the `.h` files before the `.c` file. Otherwise, the ordering of the commands is irrelevant.

2.3 Multiple Source Files

A larger C program often consists of several `.c` files. Every `.c` file is a separate translation unit and must be translated by the C compiler to a seperate `.o` file. The `.o` files are then linked together to yield the executable program binary.

To process such a program by Gencot every `.c` file must be processed on its own in the form

```
gencot -I. cfile foo.c
```

and yields a separate Cogent source file `foo.cogent`. All these translations together comprise a single Cogent program which is compiled by the Cogent compiler in a single step and results in a single C compilation unit.

The file `unit.files` passed to `gencot unit` must now contain the names of all `.c` files, each in a separate line.

Together, the commands for translating the C program are

```
gencot hfile foo.h
... <other .h files> ...
gencot -I. cfile foo.c
... <other .c files> ...
gencot -I. unit unit.files
```

The generated file `unit.cogent` includes all translations of `.c` files. As before, it is the file to be processed by the Cogent compiler.

2.3.1 Example

In the example program in `examples/cards2` the contents of `card.c` from `examples/cards` has been distributed to the three files `main.c`, `cards.c` and `rank.c`, which are listed in `unit.files`. The tranlation step (`make cogent`) corresponds to the commands

```
gencot hfile cards.h
gencot hfile rank.h
gencot -I. cfile cards.c
gencot -I. cfile rank.c
gencot -I. cfile main.c
gencot -I. unit unit.files
```

The resulting file `unit.cogent` includes all three Cogent sources `cards.cogent`, `rank.cogent`, and `main.cogent`.

The entry wrappers are now distributed in the same way to the three files `cards-entry.ac`, `rank-entry.ac`, and `main-entry.ac` (Not yet implemented! Provided with the example). All three must be specified now for the option `-infer-c-funcs` in the Cogent compilation step (`make cogent-c`) and are included by the final C source `unit.c`.

2.4 Partial Translation

Gencot supports partially translating a C program to Cogent, if the C program consists of multiple `.c` source files. In this case, some of the `.c` files are translated to Cogent and together yield a syntactically complete Cogent program. When translated back by the Cogent compiler it results in a single C compilation unit, which must be linked with the remaining `.c` files to be executed.

In this way it is possible to incrementally translate a large C program to Cogent, starting with a single `.c` source and then extending it to more and more `.c` files until eventually all `.c` files have been translated. At every intermediate step the program should still be executable after processing the translated part with the Cogent compiler and can be tested whether it still behaves in the same way as the original C program.

To partially translate a C program, proceed as before, but put into `unit.files` only the names of those `.c` files which shall be translated. You need only translate those `.h` files which are (directly or transitively) included by the translated `.c` files.

For a partially translated C program Gencot generates entry wrappers only for the translated functions, but it now generates exit wrappers for all functions in the remaining `.c` files which are invoked from the translated functions. In this way function invocations between the translated part and the remaining part are supported in both directions. Additionally, Gencot translates all data types in a binary compatible way (not yet implemented for structs! requires Dargent or modification of Cogent), so that data can be passed back and forth between the translated and remaining part.

2.4.1 Example

The example program in `examples/cards2` can be partially translated by applying Gencot only to the files `main.c` and `rank.c` and using the resulting Cogent program together with the original `cards.c`. The example provides the file `part.files` with the content

```
rank.c
main.c
```

The partial translation step (`make part-cogent`) corresponds to the commands

```
gencot hfile cards.h
gencot hfile rank.h
gencot -I. cfile rank.c
gencot -I. cfile main.c
gencot -I. unit part.files
```

The resulting file `part.cogent` includes only the Cogent sources `rank.cogent`, and `main.cogent`. Note that you still have to translate `cards.h` because it is also included by `main.c`.

To execute the partially translated program, first edit the Cogent files (`make part-edit`) so that they contain the following Cogent code:

```
rank.cogent:
  #include "rank-incl.cogent"

  cogent_ranking_to_string : Cogent_hand_ranking_t -> String
  cogent_ranking_to_string r =
    if r == cogent_STRAIGHT_FLUSH then "STRAIGHT_FLUSH" else
    if r == cogent_FOUR_OF_A_KIND then "FOUR_OF_A_KIND" else
    if r == cogent_FULL_HOUSE then "FULL_HOUSE" else
    if r == cogent_FLUSH then "FLUSH" else
    if r == cogent_STRAIGHT then "STRAIGHT" else
    if r == cogent_THREE_OF_A_KIND then "THREE_OF_A_KIND" else
    if r == cogent_TWO_PAIR then "TWO_PAIR" else
    if r == cogent_PAIR then "PAIR" else
    if r == cogent_NOTHING then "NOTHING" else
    "Invalid"

main.cogent:
  #include "cards-incl.cogent"
  #include "rank-incl.cogent"

  cogent_main : () -> U32
  cogent_main () =
    cogent_puts("Just a test:");
    cogent_print_card(cogent_card_from_num(42));
    cogent_puts("  is card no 42");
    cogent_print_card(cogent_card_from_letters('Q', 'd'));
    cogent_puts("  is diamonds queen");
    0
```

Then compile and postprocess the Cogent program (`make part-cogent-c`):

```
cogent -o part-gen -g part.cogent \
   --cogent-pp-args="-I$GENCOT_HOME/include" \
   --infer-c-funcs="part-externs.ac main-entry.ac rank-entry.ac"
auxcog unit part.files
```

Compile the resulting C program `part.c` with the C compiler to yield `part.o`, compile the original `cards.c` to yield `cards.o`, and link both files to the binary `part-cogent-cards` (`make part-cogent-binary`). When executing this binary it should behave like the original C program.

Chapter 3

Item Properties

Chapter 4

Preprocessor Directives