# Gencot User Manual

Gunnar Teege

September 17, 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`, `parmod`, and `items` (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)

Prepare the use of Gencot by executing the command

```
items used
```

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`. It contains a replacement for the original `main` function, implemented in C with embedded Cogent code ("antiquoted C").

Next enter the command

```
gencot unit
```

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

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

which is also valid Cogent code and declares `cogent_puts` as an abstract function. File `all.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 `all-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. Open `hello.cogent` in a text editor, replace its content by

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

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 `all.cogent`.

**Step 4:** (make c)

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

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

It creates the files `all-gen.c` and `all-gen.h` and two `_pp_inferred.c` files generated from the `.ac` files.

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

```
auxcog unit
```

It creates the file `all.c` which includes `all-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. Additionally it creates `hello-entry.c` and `all-externs.c` from the `_pp_inferred.c` files and also includes them into `all.c`. Additionally, `all.c` includes the files `all-gencot.h` and `all-cogent-common.c`. These files are provided by the Gencot distribution. They must be copied and renamed from `$GENCOT_HOME/c/gencot.h` and `$GENCOT_HOME/c/cogent-common.c`, respectively.

**Step 5:** (make cogent-run)

Try whether it still works. Compile `all.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 `all.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` "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
```

It reads the file `all.unit` which must contain the name of the C source file (`"foo.c"`) in a single line. It creates the following files:

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

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

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

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

`all.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 `all-exttypes.cogent` for only those types which are actually used by the C program.

The external functions and types used by the C program are determined seperately using the command

```
items used
```

It creates the file `all-external.items` which is read by most other Gencot commands. Therefore it should be executed before any other command is issued to make the file available.

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 `all-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 using the "unit name" `all`. A different unit name may be specified with the option `-u` for the `gencot` command (and also the `items` command), such as in

```
gencot -u other unit
```

Then it reads the file `other.unit` and creates files `other.cogent`, `other-externs.cogent`, `other-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

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

where `all.unit` 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 translation step (`make cogent`) corresponds to the commands

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

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 `all.unit` read by `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

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

The generated file `all.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 `all.unit`. The translation step (`make cogent`) corresponds to the commands

```
items -I. used
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
```

The resulting file `all.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`. All three must be specified now for the option `-infer-c-funcs` in the Cogent compilation step (`make c`) and are included by the final C source `all.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 `all.unit` 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.unit` with the content

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

To work with this file instead of `all.unit` explicitly specify the unit name `part` in all commands.

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

```
items -I. -u part used
gencot -u part hfile cards.h
gencot -u part hfile rank.h
gencot -I. -u part cfile rank.c
gencot -I. -u part cfile main.c
gencot -I. -u part unit
```

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 UNIT=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 UNIT=part 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 -u part unit
```

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 `cogent-cards` (`make UNIT=part cogent-binary`). When executing this binary it should behave like the original C program.

# Chapter 3

# Translation of Types

Gencot is able to translate most C types to a corresponding Cogent type. In this chapter we will look at the way how Gencot does this.

The goal of Gencot is to translate C types to "binary compatible" Cogent types. This means, if the type is translated back to C by the Cogent compiler, it must have the same memory layout. This property is required for partial translation (see Section 2.4): it must be possible to pass values between translated Cogent parts and original C parts at runtime without need for conversion.

## 3.1 Automatic Type Translation

For every C type Gencot has a basic way how to automatically translate it to a binary compatible Cogent type.

### 3.1.1 Example

To get a first impression, go to `examples/types`. The file `types.c` contains example type definitions for several different kinds of C types. Use Gencot to translate this file, as described in Section 2.1 (`make cogent`):

```
items used
gencot cfile types.c
gencot unit
```

Now look at the generated file `types.cogent`. For every C type definition in `types.c` it contains a Cogent type definition where the type expression on the right-hand side is the translation of the original C type.

### 3.1.2 Direct Types

A C type is called a "direct type", if it is primitive, or an enumeration, a struct or a unit type.

As you can see from the examples, primitive C types are directly translated to primitive Cogent types, if they are integral types. For floating point types an error specification is generated which is not a syntactically valid Cogent type and will be signaled as an error by the Cogent compiler. Floating point types must be translated manually by the developer.

Note that Gencot generates Cogent type names of the form `Cogent_<name>` where `<name>` is the "typedef name" used in C.

Enumeration types are always translated to type `U32`. According to the C standard, enumeration types are always equivalent to type `int`, irrespective how many values are declared for them. Gencot additionally introduces a Cogent constant for every declared enumeration value. If the enum type has a tag, Gencot introduces an additional type alias of the form `Enum_Cogent_<tag>`.

Structure types are translated to Cogent record types. If the struct type has a tag, Gencot introduces a type alias of the form `Struct_Cogent_<tag>`. For an anonymous struct (without a tag), Gencot instead introduces a type alias of the form `Struct<line>_<file>` using the line number and source file name where the struct definition occurs.

Although the struct type corresponds to an unboxed record type in Cogent, the type definition defines an alias for the boxed record type instead. The unbox operator is applied to all *references* to the type alias, as you can see for the fields in the record `Struct_Cogent_line`.

Union types are translated to abstract types, omitting the fields. Semantically, the corresponding Cogent types would be variant types, however, these are not binary compatible. Therefore Gencot leaves the translation of union types to the developer, it only generates type aliases in a similar way as for struct types.

### 3.1.3 Derived Types

In C there are three kinds of types called "derived types": function types, pointer types, and array types.

As you can see from the examples, C function types are directly translated to Cogent function types. Note that these are not binary compatible, however, this is not a problem since in C functions cannot occur in data structures (in contrast to function *pointers*).

In Cogent every function has a single value as parameter and a single value as result. Gencot wraps multiple parameters in a tuple and uses the unit value, if a C function has no parameter or no result value (specified as `void`). A C function declared to take a variable number of parameters (a "variadic" function) is translated as if the additional parameters would always be a single value. The used pseudo type alias is not defined and will be signaled as an error by the Cogent compiler, this must be handled manually by the developer.

The only pointer types directly supported by Cogent are boxed record types, which correspond to pointers to structs. Accordingly, pointers to a struct type are translated to boxed record types in Cogent. The difference from the translation of a struct type can be seen at the fields in the example `Struct_Cogent_sphere`: for the struct pointer no unbox operator is applied.

However, instead of an unbox operator, the generic type `MayNull` is applied to the referenced pointer type. In Cogent, a value of a boxed record type must be guaranteed to be not `NULL`. Since this is not the case for an arbitrary C pointer type, both are not strictly binary compatible. Gencot defines the generic type `MayNull` and applies it to all translated C pointer types. Gencot provides an abstract data type for `MayNull` which allows access to the record only after an explicit test whether the pointer is not `NULL`. Like the unbox operator, Gencot

does not apply `MayNull` in the type definition, but instead to all references of the defined type alias.

C pointer types where the base type is a primitive type, an enum type, or another pointer type, are translated with the help of the generic type `CPtr`. Gencot defines a corresponding abstract type with functions to access the referenced value. Again, whenever such a type is used, the `MayNull` type is applied to it to care for `NULL` values.

C function pointers are treated in a different way by Gencot. Since there is no binary compatible Cogent type for them (function type values are represented by integers numbering all known functions in Cogent), Gencot translates function pointer types to abstract Cogent types. To support a unique correspondence, the C function type expression is encoded into the name used for the abstract Cogent type. As an exercise, you may try to find out the encoding scheme by looking at the examples. The example defines function pointer types for all function types defined previously in the same example. The additional example `Cogent_funHigher_t` shows that this also works for higher order functions: as specified by the C standard, a parameter of function type is "adjusted" to function pointer type and translated accordingly by Gencot.

Translated C function pointer types are always used with an unbox operator applied. Thus they behave in the same way as primitive types in Cogent. This is correct, since although the values are pointers, they cannot be dereferenced in a Cogent program. Instead, Gencot provides functions to convert between these pointers and values of the corresponding Cogent function type.

Cogent provides means to represent arrays, but none of them is fully binary compatible with C arrays. Therefore Gencot translates all array types to abstract Cogent types and provides abstract functions to work with their values. Gencot uses generic types of the form `CArr<size>` where the array size (number of elements) is encoded into the type name. This works for all kinds of element types.

Depending on its use, an array in C may be represented by a pointer to the first element or by the sequence of all elements. Accordingly, Gencot translates an array type either to the boxed abstract type (corresponding to pointer values) or it applies the unbox operator (corresponding to the sequence of elements). In the example this can be seen in the types `Struct56_types_c` and `Cogent_funArray_t`: when used as a struct member, the array is used in its unboxed form and the sequence of elements is embedded in-place; when used as a function parameter, the array type is used in its boxed form corresponding to a pointer, because according to the C standard the array type is "adjusted" to an array pointer type.

Finally, the example shows that multidimensional arrays are translated to array-of-array. Again, the unboxed form is used since the elements of the inner array are directly embedded in the outer array.

### 3.1.4 Generated Abstract Types

We have seen that Gencot uses some abstract types when it translates C "derived types" to Cogent. For some of them, like `MayNull` and `CPtr`, the Gencot distribution includes their definitions. However, other abstract types depend on the translated C program: those used for array types and those used for function pointer types. Where are their definitions provided?

Look at the file `all-dvdtypes.cogent` in `examples/types`. As described in Section 2.1 it was generated by the command

```
gencot unit
```

It contains all abstract type definitions required for translated derived C types. Note that for every array size used in the program a single type of the form `CArr<size>` is defined, even if several types of this size with different element types are used in C. You can also see that `CArr<size>` is actually not abstract, instead the array is wrapped in a record for technical reasons.

As a convenience, for every function pointer type alias a similar alias for the corresponding function type is defined. This is intended to be used when converting between both.

### 3.1.5 Continuing the Example

To verify that the type definitions generated in Section 3.1.1 can actually be processed by the Cogent compiler, you may perform the following steps:

**Step 2:** (make edit)

Remove all type definitions generated from the floating point types and variadic function types from the files `types.cogent` and `all-dvdtypes.cogent`

**Step 3:** (make c)

Now you can process `all.cogent` by the Cogent compiler. Enter the command

```
cogent -o all-gen -g all.cogent \
   --cogent-pp-args="-I$GENCOT_HOME/include" \
   --infer-c-funcs="all-externs.ac types-entry.ac"
```

It creates the files `all-gen.c` and `all-gen.h` and two (empty) `_pp_inferred.c` files.

The option `-cogent-pp-args` adds the directory `$GENCOT_HOME/include` to the include path used by Cogent. It is needed to load the definitions for the types `MayNull` and `CPtr` from the Gencot distribution.

The corresponding include directives can be found in the file `gencot.cogent` which is provided with the example. Whenever Gencot finds a file with this name in the current directory it includes it in the generated file `all.cogent`. By adding include directives to `gencot.cogent` all types and functions predefined by Gencot can be made available.

As usual, enter the command

```
auxcog unit
```

for postprocessing.

**Step 4:** (make cogent-binary)

Compile `all.c` with a C compiler to an object file, such as in

```
cc -c -I‘cogent --libgum-dir‘ -I$GENCOT_HOME/c all.c
```

This should work and produce `all.o`, however, it is of no much interest, since Cogent does not translate most of our type definitions to C, because they are not used in functions.

**Step 5:** (make clean)

Clean up all generated files.

## 3.2 Default Item Properties

The basic way of Gencot's type translation can be affected with the help of "item properties". An item is every unit in the C program which has a type (such as variables, functions, parameters, struct members, ...). Gencot identifies items by specific path names. Using these path names item properties can be declared outside the C program which affect how the item's type is translated by Gencot.

Some item properties (the "default properties") are automatically derived from the C program. Instead of directly using the information for Gencot's type translation, it is externalized in the form of item property declarations. Thus it is possible for the developer to modify it before it is used by Gencot.

### 3.2.1 Example

To see how it works, go to `examples/items`. The file `items.h` contains example type definitions which demonstrate the impact of default item properties. Translate this file with Gencot, as described in Section 2.2 (`make h-cogent`):

```
items -u part used
gencot -u part hfile items.h
```

and save the resulting file `items-incl.cogent` as `items-incl-auto.cogent`. Now derive the default properties for `items.h` with the command (`make h-props`):

```
items -u part file items.h > items.h-itemprops
```

and translate `items.h` again with the same command as above.

Now look at the generated file `items-incl.cogent`. You can see the effect of the default properties by comparing it to the saved `items-incl-auto.cogent`. When Gencot translates a source file `x` it looks for a file named `x-itemprops` in the current directory. If it finds it, it uses it for its type translations. Therefore the two translation results differ.

### 3.2.2 Read-Only Types

The first group of type definitions in `items.h` demonstrates the generation of read-only types in Cogent. If the base type of a pointer is `const` qualified in C, as for `intCP_t` in the example, Gencot makes the translated type read-only. This works for all kinds of base types, such as structures (example: `coord3dCP_t`) and for all kinds of items, such as function parameters and results (examples: `funBinPtr_t, funResPtr_t`).

But why does it not work for `sphereP_t`, although its base type is `const` qualified? The reason is that `struct sphere` contains a member `center` which is a pointer without `const` qualified base type. In Cogent a value comprises *all* parts which can be transitively accessed following pointers. Gencot only translates a type as read-only if all these pointers have a `const` qualified base type and thus no part of the Cogent value may be modified. This is demonstrated by `sphereCP_t` in the example.

Array types are treated like pointers: if the element type is `const` qualified (and also all transitively accessible pointers and arrays), Gencot translates the array type as read-only. However, as described in Section 3.1.2 for the unbox operator, for array types the bang operator is only applied to type name references, not in the type definition. In the example you can see this for type `intCA_t`, the read-only form occurs when it is used in the translated `funArrPar_t`. The reason for this behavior is that translated array types may also occur in the unboxed form, as in the translation of `arrstr_t`, there the bang operator is omitted.

The next group of type definitions shows a special case: a `const` qualified `char` pointer is translated to the Cogent type `String`. (You have already seen this behavior in earlier examples, there the default item property files have been provided with the example.)

### 3.2.3 Additional Function Results

The last two groups show a feature provided for function types: for all parameters translated to a linear type (i.e. a a type which is or transitively accesses a pointer or array without `const` qualified base type) an additional component is added to the result. It is assumed that such parameters will be modified by the function and the modified value must be explicitly returned in Cogent.

For a function type the result type is changed to a tuple, if necessary, where the first component is the original function result. For a function pointer type, this property is encoded in the type name (an `"M"` marks "modifyable" parameters, an `"R"` marks "read-only" parameters).

### 3.2.4 Processing the Item Property Files

To use the item properties for Gencot's type translation you have to generate item property files before using the `gencot` command to translate C sources. First, for every C source file `foo.h` or `foo.c` to be translated by `gencot` process the file with the `items` command:

```
items -I. file foo.h > foo.h-itemprops
items -I. file foo.c > foo.c-itemprops
```

The include path option `-I` is required in all cases in which it is required for the `gencot` command as described in Section 2.2. Here we assume that all included (non-system) files are in the current directory.

Additionally, properties may be required for external items. The default properties for all external items used in the translated files are generated in file `all.unit-itemprops` with the command:

```
items -I. unit > all.unit-itemprops
```

For generating the additional Cogent sources with command `gencot unit` the file `all.unit-itemprops` must also contain properties for `all` items in all translated source files. These are provided by mergin all the files created before in `all.unit-itemprops`. Merging two item property files is done by

```
items merge <f1>-itemprops <f2>-itemprops > <f3>-itemprops
```

Together, the commands for creating all required item property files are:

```
items -I. used
items -I. file foo.h > foo.h-itemprops
... <other .h files> ...
items -I. file foo.c > foo.c-itemprops
... <other .c files> ...
items -I. unit > all.unit-itemprops
items merge all.unit-itemprops foo.h-itemprops > hfile
mv hfile all.unit-itemprops
... <other .h-itemprops files> ...
items merge all.unit-itemprops foo.c-itemprops > hfile
mv hfile all.unit-itemprops
... <other .c-itemprops files> ...
```

Note that you cannot output the result of `items merge` directly to `all.unit-itemprops`, this would clear the file before it is read.

## 3.3 Manual Item Properties

The developer can modify the way Gencot translates C types by manually adding or removing item properties. In the example look at the file `items.h-itemprops` which has been generated in Section 3.2.1. Every line names an item by its path name and specifies properties for it. Every property has a two-letter code, `"ro"` specifies a read-only type, `"ar"` specifies an additional result for a function parameter. Note that, if both are specified for an item, `"ro"` has higher priority and `"ar"` is ignored.

It should be relatively straightforward to identify the items from their pathnames. A slash is used as separator, for struct members a dot is used. Function parameters are identified by their position number, starting with position 1, if the name is not available. Function results are identified by `()`.

As you can see, items are listed even if no properties are specified for them. This is a convenience for the developer. Gencot always lists the pathnames for *all* items which occur in a C source file. The developer can then add properties for an item without first constructing its pathname.

### 3.3.1 Example

Open the file `items.h-itemprops` in a text editor. Add the `ro` property for the type item `intP_t` and remove the `ar` property from the first parameter of the type item `funLinPar1_t` (`make dflt-edit`). Then remove `items-incl.cogent` and translate `items.h` once more. Now the Cogent type `Cogent_intP_t` will be read-only and the Cogent type `Cogent_funLinPar1_t` will not have an additional result component anymore.

In `examples/items` additionally generate the item property file for `items.c` with the command (`make c-props`):

```
items -I. -u part file items.c > items.c-itemprops
```

In file `items.c-itemprops` you can see how functions are identified as items. For static functions the file name where they are defined is prepended to the function name, so that the item identifier should be unique, even if the same name is reused for static functions in different files.

Next, generate the default property declarations for all used external items with the command (`make e-props`):

```
items -I. -u part unit
```

Look at the output. It lists entries for the items `allocArr` and `allocInt` defined in file `alloc.c` (which are extern, because this file is not listed in `part.unit`) and for the item `printf` declared in the system include file. From the declaration of `printf` the properties for the first parameter are derived as usual. Note that, as described in Section 2.1 for the translation, only those external items are listed which are actually used in the source files specified in `part.unit`.

### 3.3.2 Read-Only Property

The main application of manual read-only properties is the case where an item is actually not modified in the C program, although it would be possible to do so according to its type. In this case a read-only property may be added to reflect the read-only behavior in the item's Cogent type.

A typical case is that a function parameter is a pointer without `const` qualified base type, but the function never modifies the data referenced by the pointer. Specifying the read-only property for the parameter item makes the parameter's type read-only in Cogent.

Another typical case is a pointer or array type `t` where the base type is `const` qualified but it transitively contains a pointer or array where this is not the case, because it has been forgotten in C or not considered relevant, although the referenced data will never be modified through this reference path. Specifying the read-only property for the type item `t` will translate it as read-only, corresponding to the intended semantics.

### 3.3.3 Add-Result Property

The main application of manual add-result properties is the case where no additional result for a parameter is required, although the parameter translates to a linear type. In this case the add-result property generated by Gencot should be removed to suppress the additional result component.

A typical case is when the parameter is already returned as the function result or as a part of it in C, as for the function `memcpy` in the C standard library.

Another case is when the function's semantics is to actually discard the parameter by deallocating the pointer. The most apparent example is the function `free` in the C standard library.

Adding add-result properties usually makes no sense. Gencot generates them for all parameters of linear type, for all others an additional result component is of no use.

### 3.3.4 No-String Property

The no-string property is used to suppress the translation to the Cogent type `String`. It is never generated by Gencot and may be manually specified as `"ns"`, such as in

```
item-path-name: ns
```

If the item has type `const char *` it is translated to type `(MayNull (CArr U8))!` instead of type `String`. For items of a type which is not translated to `String` the property is ignored.

In the example, add the `ns` property for the type item `charCP_t` (`make ns-edit`). Then remove `items-incl.cogent` and translate `items.h` once more. Now the Cogent type `Cogent_charCP_t` will be defined as `(CArr U8)!` instead of `String` (as usual, the `MayNull` type is only applied to references to `Cogent_charCP_t`).

### 3.3.5 Not-Null Property

The not-null property is used to suppress the addition of Cogent type `MayNull` for translated pointer types (see Section 3.1.3). It is never generated by Gencot and may be manually specified as `"nn"`, such as in

```
item-path-name: nn
```

If the item has type `int *` it is translated to type `CPtr U32` instead of type `MayNull (CPtr U32)`. For items which are no pointers the property is ignored.

The not-null property can be used to express that an item's value is never `NULL` in the C program. A possible reason is that it is immediately initialized and every pointer to be assigned to it is tested to be not `NULL` before. Thus the not-null property is used to statically remember this fact for the item in Cogent with the help of the Cogent type.

In the example, add the `nn` property for the struct member item `sphereC.center` (`make nn-edit`). Then remove `items-incl.cogent` and translate `items.h` once more. Now the Cogent type of field `center` in the Cogent record type `Struct_Cogent_sphereC` will be `Cogent_coord3dCP_t!` instead of `(MayNull Cogent_coord3dCP_t)!`.

# Chapter 4

# Preprocessor Directives