HoBit Report on Gencot Development

Gunnar Teege

January 5, 2019

Chapter 1

Introduction

Gencot (GENerating COgent Toolset) is a set of tools for generating Cogent code from C code. It is developed by UniBw as part of the HoBit project conducted with Hensoldt Cyber. In the project it is used for reimplementing the mbedTLS library in Cogent as sembedTLS. However, Gencot is not specific for mbedTLS and should be applicable to other C code as well.

Gencot is used for parsing the C sources and generating templates for the required Cogent sources, antiquoted Cogent sources, and auxiliary C code.

Gencot is not intended to perform an automatic translation, it prepares the manual translation by generating templates and perfoming some mechanic steps.

Roughly, Gencot supports the following tasks:

- translate C preprocessor constant definitions and enum constants to Cogent object definitions,
- generate function invocation entry and exit wrappers,
- generate Cogent abstract function definitions for invoked exit wrappers,
- translate C type definitions to default Cogent type definitions,
- generate C type mappings for abstract Cogent types referring to existing C types,
- generate Cogent function definition skeletons for all C function definitions,
- rename constants, functions, and types to satisfy Cogent syntax requirements and avoid collisions,
- convert C comments to Cogent comments and insert them at useful places in the Cogent source files,
- generate the main files `<package>.cogent` and `<package>.ac` for Cogent compilation.

To do this, Gencot processes in the C sources most comments and preprocessor directives, all declarations (whether on toplevel or embedded in a context), and all function definitions. It does not process C statements other than for processing embedded declarations.

Chapter 2

Design

2.1 General Context

We assume that there is a C application <package> which consists of C source files `.c` and `.h`. The `.h` files are included by `.c` files and other `.h` files. There may be included `.h` which are not part of <package>, such as standard C includes, all of them must be accessible. Every `.c` file is a separate compilation unit. There may be a `main` function but Gencot provides no specific support for it.

From the C sources of <package> Gencot generates Cogent source files `.cogent` and antiquoted Cogent source files `.ac` as a basis for a manual translation from C to Cogent. All function definition bodies have to be translated manually, for the rest a default translation is provided by Gencot.

Gencot supports an incremental translation, where some parts of <package> are already translated to Cogent and some parts consist of the original C implementation, together resulting in a runnable system.

2.1.1 Modularization and Interfacing to C Parts

Every C compilation unit produces a set of global variables and a set of defined functions. Data of the same type may be used in different compilation units, e.g. by passing it as parameter to an invoked function. In this case type compatibility in C is only guaranteed by including the `.h` file with the type definition in both compilation units. In the compiled units no type information is present any more.

This organisation makes it possible to use different `.h` files in different compilation units. Even the type definitions in the included files may be different, as long as they are binary compatible, i.e., have the same memory layout.

We exploit this organisation for an incremental translation from C to Cogent as follows. At every stage we replace some C compilation units by Cogent sources. All C data types used both in C units and in Cogent units are mapped to binary compatible Cogent types. Compiling the Cogent sources again produces C code which together with the remaining C units are linked to the target program. The C code resulting from Cogent compilation is completely separated from the code of the remaining C units, common include files are only used for types which are abstract in Cogent, i.e., have no Cogent definition.

All interfacing between C compilation units is done by name. All names of C objects with external linkage can be referred from other compilation units. This is possible for functions and for global variables. Interfacing from and to Cogent works in the same way.

Interfacing to Functions

Cogent functions always take a single parameter, the same is true for the C functions generated by the Cogent compiler. Hence for interfacing from or to an arbitrary C function, wrapper functions are needed which convert between arbitrary many parameters and a single structured parameter. These wrapper functions are implemented in C.

The "entry wrapper" for invoking a Cogent function from C has the same name as the original C function, so it can be invoked transparently. Thus the Cogent implementation of the function must have a different name so that it does not collide with the name of the wrapper. Often, the function names in a C package use one or more specific prefixes. Gencot uses a configurable prefix mapping schema for generating the name of a Cogent function from the name of the corresponding C function. The default for C functions with external linkage is to map the empty prefix to `cogent_`, i.e. prepend this string to every function name.

The Cogent implementation of a C function generated by Gencot is never polymorphic. This implies that the Cogent compiler will always translate it to a single C function of the same name.

The "exit wrapper" for invoking a C function from Cogent invokes the C function by its original name, hence the wrapper must have a different name. We use the same renaming scheme for these wrappers as for the defined Cogent functions. This implies that every exit wrapper can be replaced by a Cogent implementation without modifying the invocations in existing Cogent code. Note that for every function either the exit wrapper or the Cogent implementation must be present, but not both, since they have the same name.

Note that if the C function has only one parameter, a wrapper is not required. For consistency reasons we generate and use the wrappers also for these functions.

Cogent translates all function definitions to C definitions with internal linkage. To make them accessible the entry wrappers must have external linkage. They are defined in an antiquoted Cogent (.ac) file which includes the complete code generated from Cogent, there all functions translated from Cogent are accessible from the entry wrappers. The exit wrappers are only invoked from code generated from Cogent. They are defined with internal linkage in an included antiquoted Cogent file.

Interfacing to Global Variables

Accessing an existing global C variable from Cogent is not possible in a direct way, since there are no "abstract constants" in Cogent. Access may either be implemented with the help of abstract functions which are implemented externally in additional C code and access the global variable from there. Or it may be implemented by passing a pointer as (part of) a "system state" to the Cogent function which performs the access.

Accessing a Cogent object definition from C is not possible, since the Cogent compiler does not generate a definition for them, it simply substitutes all uses of the object name by the corresponding value. Hence, all global variable definitions need to remain in C code to accessible there.

Since the way how global state is treated in a Cogent program is crucial for proving program properties, Gencot does not provide any automatic support for accessing global C variables from Cogent, this must always be implemented manually.

Cogent Compilation Unit

As of December 2018, Cogent does not support modularization by using separate compilation units. A Cogent program may be distributed across several source files, however, these must be integrated on the source level by including them in a single compilation unit. It would be possible to interface between several Cogent compilation units in the same way as we interface from C units to Cogent units, however this will probably result in problems when generating proofs.

Therefore Gencot always generates a single Cogent compilation unit for the <package>. At every intermediate stage of the incremental translation the package consists of one Cogent compilation unit together with all remaining original C compilation units and optionally additional C compilation units (e.g., for implementing Cogent abstract data types).

Textually integrating different C compilation units may cause name conflicts. C identifiers with external linkage denote the same object in the whole C program, for them no conflicts can occur. C identifiers with internal linkage denote different objects if they occur in different compilation units. To avoid conflicts for them, Gencot applies a renaming scheme optionally based on the compilation unit's file name.

The renaming scheme is applied to C functions and C objects defined with internal linkage in a file `x.c`, when the name for the corresponding Cogent function or value is generated. The names are mapped by a configurable prefix mapping schema which is different from that used for functions with external linkage. The default is to prepend the prefix `local_x_` where `x` is the basename of file `x.c`. If the mappings are configured with additional prefix mapping rules it must also be guaranteed that a name generated for an identifier with internal linkage cannot conflict with a name generated for an identifer with external linkage.

Name conflicts could also occur for type names and tags defined in a `.h` file. This would be the case if different C compilation units include individual `.h` files which use the same identifier for different purposes. However, most C packages avoid this to make include files more robust. Gencot assumes that all identifiers defined in a `.h` file are unique in the <package> and does not apply a file-specific renaming scheme. If a <package> does not satisfy this assumption Gencot will generate several Cogent type definitions with the same name, which will be detected and signaled by the Cogent compiler.

2.1.2 Cogent Source File Structure

Although the Cogent source is not structured on the level of compilation units, Gencot intends to reflect the structure of the C program at the level of Cogent

source files.

Note, that there are four kinds of include statements available in Cogent source files. One is the `include` statement which is part of the Cogent language. When it is used to include the same file several times in the same Cogent compilation unit, the file content is automatically inserted only once. The second kind is the Cogent preprocessor `#include` directive, it seems obsolete since it can be replaced by the Cogent language `include` statement. The third kind is the preprocessor `#include` directive which can be used in antiquoted Cogent files where the Cogent `include` statement is not available. This is only possible if the included content is also an antiquoted Cogent file. The fourth kind is the `#include` directive of the C preprocessor which can be used in antiquoted Cogent files in the form `$esc:(#include ...)`. It is only executed when the C code generated by the Cogent compiler is processed by the C compiler. Hence it can be used to include normal C code.

Gencot assumes the usual C source structure: Every `.c` file contains definitions with internal or external linkage. Every `.h` file contains preprocessor constant definitions, type definitions and function declarations. The constants and type definitions are usually mainly those which are needed for the function declarations. Every `.c` file includes the `.h` file which declares the functions which are defined by the `.c` file to access the constants and type definitions. Additionally it may include other `.h` files to be able to invoke the functions declared there. A `.h` file may include other `.h` files to reuse their constants and type definitions in its own definitions and declarations.

Cogent Source Files

In Cogent a function which is defined may not be declared as an abstract function elsewhere in the program. If the types and constants needed for defining a set of functions should be moved to a separate file, like in C, this file must not contain the function declarations for the defined functions. Declarations for functions defined in Cogent are not needed at all, since the Cogent source is a single compilation unit and functions can be invoked at any place in a Cogent program, independently whether their definition is statically before or after this place.

Therefore we map every C source file `x.c` to a Cogent source file `x-impl.cogent` containing definitions of the same functions. We map every C include file `x.h` to a Cogent source file `x-types.cogent` containing the corresponding constant and type definitions, but omitting any function declarations. The include relations among `.c` and `.h` files are directly transferred to `-impl.cogent` and `-types.cogent` using the Cogent include statement.

Although it is named `x-types.cogent`, the file also contains Cogent value definitions generated from C preprocessor constant definitions and from enumeration constants (see below). It would be possible to put the value definitions in a separate file. However, then for other preprocessor macro definitions it would not be clear where to put them, since they could be used both in constant and type definitions. They cannot be moved to a common file included by both at the beginning, since their position relative to the places where the macros are used is relevant.

This file mapping implies that for every translated `.c` file all directly or indirectly included `.h` files must be translated as well. Alternatively, instead of

using a Cogent type definition for every C type in an included `.h` file, a Cogent abstract type can be used. In this way further included `.h` files may become unnecessary and need not be translated. However, this must be decided and realized manually. Gencot always generates default Cogent type definitions and the include statements for all `-types.cogent` files.

Wrapper Definition Files

The entry wrappers for the functions defined with external linkage in `x.c` are implemented in antiquoted Cogent code and put in the file `x-entry.ac`. The exit wrappers for invoking the functions defined with external linkage in `x.c` are put in the separate file `x-exit.ac`. Note, that these are not functions invoked *by* the code in `x.c` but functions *defined* in `x.c` and invoked by other compilation units.

Abstract Functions as Interface to C Functions

If a Cogent function in `x-impl.cogent` invokes a function originally defined in `y.c` which has not yet been translated, this function must be declared as an abstract function in Cogent. For the functions defined in `y.c` we put corresponding abstract function declarations in the file `y-decls.cogent` and use this file instead of `y-impl.cogent`. The implementation for these functions is provided by the exit wrappers in `y-exit.ac`.

Abstract Types as Interface to C Types

C type names and tags usually start with a lowercase letter. In Cogent all type names must start with an uppercase letter, hence we cannot use the same name in Cogent to refer to a C type. Basically, Gencot uses a similar prefix mapping scheme for type names and tags as for function names. The default for generating Cogent type names from C type names or tags is to prepend the prefix `Cogent_`.

A C type can be used in Cogent in two possible ways. Eiter it is defined as a Cogent abstract type, or it is defined by providing a Cogent type expression as definition. In the second case the Cogent compiler will generate a C type definition with the same name. The Cogent type expression must be chosen in a way, that this C type definition is binary compatible to the original C type definition, i.e., it has the same memory layout.

For an abstract type the Cogent compiler does not generate any definition, it is intended to directly refer to the original C type. Since the name is different, we have to generate a C type definition mapping the Cogent type name to the C type name. However, this definition may only be present for abstract types, for the other types it would conflict with the C type definition generated by the Cogent compiler. To make the definitions independent from the Cogent code, Gencot avoids these conflicts by using a different renaming scheme for abstract types. The default for generating Cogent abstract type names is to prepend the prefix `Abstract_` to the C type name or tag.

Hence Gencot generates abstract type mapping definitions for all types in the form

```
typedef typename Abstract_typename
```

or using a configured prefix mapping. This is done for all types defined in `x.h`, irrespective of their translation to an abstract or non-abstract type in Cogent. If a type is translated to a non-abstract Cogent type the mapping typedef for it is not used.

For all types defined in file `x.h` we put the corresponding type mapping definitions in file `x-exttypes.h`. To use these definitions in the Cogent compilation unit the file `x.h` must be included there as well. Note that this is possible without conflicts, since we renamed all types used in Cogent.

Global Variables

In C a compilation unit can define global variables. Gencot does not generate an access interface to these variables from Cogent code. However, the variables must still be present in a compilation unit, since they may be accessed from other C compilation units (if they have external linkage).

Gencot assumes that global variables are only defined in `.c` files. For every file `x.c` Gencot generates the file `x-globals.c` containing all toplevel object definitions with external linkage in `x.c`. For these definitions, some type and constant definitions may be required, so they must also be added to `x-globals.c`. Since the required types may be defined in included `.h` files, these files must be included in `x-globals.c`. Instead of tracking, what is required for the global variable definitions, Gencot simply generates `x-globals.c` from `x.c` by removing all function definitions and all object definitions with internal linkage.

Toplevel object definitions with internal linkage cannot be accessed from other C compilation units. They cannot be accessed from Cogent code either, hence they are useless, they must be replaced manually by a Cogent solution for managing the corresponding global state.

However, to inform the Cogent programmer about the global variables defined in `x.c` and their types, Gencot generates corresponding Cogent value definitions for all toplevel object definitions with internal or external linkage. For each of them the initializer is transferred unmodified from C, no Cogent expression for the defined value is generated. Either the initializer is manually converted to a Cogent expression, or the value definition is replaced by another solution.

All Cogent value definitions for global variables in `x.c` are put in the file `x-globals.cogent`. Since it is only intended as an information for the Cogent programmer it is *not* included automatically by any generated Cogent source file.

Abstract Data Types

There may also be cases of C types where no corresponding Cogent type can be defined, in this case it must be mapped to an abstract data type T in Cogent, consisting of an abstract type together with abstract functions. Both are put in the file `abstract/T.cogent` which is included manually by all `x-types.cogent` where it is used. The types and functions of T must be implemented in additional C code. In contrast to the abstract functions defined in `x-decls.cogent`, there are no existing C files where these functions are implemented. The implementations are provided as antiquoted Cogent code in the file `abstract/T.ac`.

If T is generic, the additional file `abstract/T.ah` is required for implementing the types, otherwise they are implemented in `abstract/T.h`.

Gencot does not provide any support for using abstract data types, they must be managed manually according to the following proposed schema. All related files should be stored in the subdirectory `abstract`. An abstract data type T is defined in the following files:

`T.ac` Antiquoted Cogent definitions of all functions of T.

`T.ah` Antiquoted Cogent definition for T if T is generic.

`T.h` Antiquoted Cogent definitions of all non-generic types of T.

Using the flag `-infer-c-types` the Cogent compiler generates from `T.ah` files `T_t1...tn.h` for all instantiations of T with type arguments t1...tn used in the Cogent code.

File Summary

Summarizing, Gencot uses the following kinds of Cogent source files for existing C source files `x.c` and `x.h`:

`x-impl.cogent` Implementation of all functions defined in `x.c`. For each file `y.h` included by `x.c` the file `y-types.cogent` is included.

`x-decls.cogent` Abstract function definitions for all functions with external linkage defined in `x.c`. Includes all `y-types.cogent` for which `y.h` is included by `x.c`.

`x-globals.cogent` Value definitions for all objects defined in `x.c`. No files are included, the file is not included by any other file.

`x-types.cogent` Constant and type definitions for all constants and types defined in `x.h`. If possible, for every C type definition a binary compatible Cogent type definition is generated by Gencot. Otherwise an abstract type definition is used. Includes all `y-types.cogent` for which `x.h` includes `y.h`.

`x-entry.ac` Antiquoted Cogent definitions of entry wrapper functions for all function definitions with external linkage defined in `x.c`.

`x-exit.ac` Antiquoted Cogent definitions of exit wrapper functions for all function definitions with external linkage defined in `x.c`.

`x-exttypes.h` C definitions of abstract Cogent types used to reference existing C types defined in `x.h`.

`x-globals.c` All C object definitions with external linkage occurring in `x.c`.

For each file `x.c` only one of `x-impl.cogent` and `x-decls.cogent` may exist.

Main Files

To put everything together we use the files `<package>.cogent` and `<package>.ac`. The former includes all existing `x-impl.cogent` and `x-decls.cogent` files. It is the file processed by the Cogent compiler which translates it to files `<package>.c` and `<package>.h` where `<package>.c` includes `<package>.h`.

The file `<package>.ac` includes all existing files `x-entry.ac`, `x-exit.ac` and the file `<package>.c` and is processed by the Cogent compiler through the `-infer-c-funcs` flag. The resulting file is `<package>_pp_inferred.c` which is the C compilation unit for all parts of <package> already translated to Cogent. All existing files `x-exttypes.h` are `$esc`-included in `<package>.ac`, thus the corresponding normal includes for them are present in `<package>_pp_inferred.c`. For all existing files `x-impl.cogent` the corresponding file `x-globals.c` is `$esc`-included in `<package>.ac`, to make all global variables with external linkage in `x.c` a part of `<package>_pp_inferred.c`.

Every abstract type T yields an additional separate C compilation unit `T_pp_inferred.c`.

The content of `abstract/T.h` and all `abstract/T_t1...tn.h` is required in the compilation unit for T and in that for `<package>.c`. The Cogent compiler automatically generates includes for all `abstract/T_t1...tn.h` in `<package>.h`, thus they are available in `<package>_pp_inferred.c`. By manually `$esc`-including `<package>.h` in every `abstract/T.ac` they are made available there as well. In the same way `abstract/T.h` can be `$esc`-included in `abstract/T.ac`. To make it available in the `<package>.c` unit Gencot also `$esc`-includes all existing `abstract/T.h` files in `<package>.ac`.

2.2 Processing Comments

The Cogent source generated by Gencot is intended for further manual modification. Finally, it should be used as a replacement for the original C source. Hence, also the documentation should be transferred from the C source to the Cogent source.

Gencot uses the following heuristics for selecting comments to be transferred: All comments at the beginning or end of a line and all comments on one or more full lines are transferred. Comments embedded in C code in a single line are assumed to document issues specific to the C code and are discarded.

2.2.1 Identifying and Translating Comments

Gencot processes C block comments of the form `/* ... */` possibly spanning several lines, and C line comments of the form `// ...` ending at the end of the same line.

Identifying C comments is rather complex, since the comment start sequences `/*` and `//` may also occur in C code in string literals and character constants and in other comments.

Comments are translated to Cogent comments. Every C block comment is translated to a Cogent block comment of the form `{- ... -}`, every C line comment is translated to a Cogent line comment of the form `-- ...`. Only the start and end sequences of identified comments are translated, all other occurrences of comment start and end sequences are left unchanged.

2.2.2 Comment Units

Gencot assembles sequences of transferrable comments which are only separated by whitespace together to comment units as follows. All comments starting in the same line after the last existing source code are concatenated to become one unit. Such units are called "after-units". All comments starting in a separate line with no existing source code or before all existing source code in that line are concatenated to become one unit. Such units are called "before-units".

Additionally, all remaining comments at the end of a file after the last after-unit are concatenated to become the "end-unit". At the beginning of a file there is often a schematic copyright comment, to allow for a specific treatment a configurable number of comments at the beginning of a file are concatenated to become the "begin-unit". The default number of comments in the begin-unit is 1.

As a result, every transferrable comment is either part of a comment unit and every comment unit can be uniquely identified by its kind and by the source file line numbers where it starts and where it ends.

Heuristically, a before-unit is assumed to document the code after it, whereas an after-unit is assumed to document the code before it. Based on this heuristics, comment units are associated to code parts. A begin-unit and an end-unit is assumed to document the whole file and is not associated with a code part.

2.2.3 Relating Comment Units to Documented Code

Basically, Gencot translates source code parts to target code parts. Source code parts may consist of several lines, so there may be several before- and after-units associated with them: The before-unit of the first line, the after-unit of the last line and possibly inner units. Target code parts may also consist of several lines. The before-unit of the first line is put before the target code part, the after-unit of the last line is put after the target code part.

If there is no inner structure in the source code part which can be mapped to an inner structure of the target code part, there are no straightforward ways where to put the inner comment units. They could be discarded or they could be collected and inserted at the beginning or end of the target code part. If they are collected no information is lost and irrelevant comments can be removed manually. However, in well structured C code inner comment units are rare, hence Gencot discards them for simplicity and assumes, that this way no relevant information will be lost.

If the source code part has an inner structure units can be associated with subparts and transferred to subparts of the target code part. Gencot uses the following general model for a structured source code part: It may have one or more embedded subparts, which may be structured in a similar way. Every subpart has a first line where it begins and a last line where it ends. Before and after a subpart there may be lines which contain code belonging to the surrounding part. Subparts may overlap, then the last line of the previous subpart is also the first line of the next subpart. Subparts may overlap with the surrounding part, then the first or last line of the subpart contains also code from the surrounding part.

For a structured source code part Gencot generates a target code part for the main part and a target code part for every subpart. The subpart targets

may be embedded in the main part target or not. If they are embedded they may be reordered.

The inner comment units of a structured source code part can now be classified and associated. Every such unit is either an inner unit of the main part, a before-unit of the first line of a subpart if that does not overlap, an inner unit of a subpart, or an after-unit of the last line of a subpart, if that does not overlap. The units associated with a subpart are transferred to the generated target according to the same rules as for the main part.

Inner units of the main part may be before the first subpart, between two subparts, or after the last subpart. Following the same argument as for inner units of unstructured source code parts, Gencot simply discards all these inner units.

As a result, for every source code part atmost the before-unit of the first line and the after-unit of the last line is transferred to the target part. If the source code part is structured the same property holds for every embedded subpart.

2.2.4 Function Declaration Comments

Since function declarations are not translated to a target code part in Cogent, all comments associated with them would be lost. However, often the API documentation of a function is associated with a function declaration instead of the function definition.

Therefore Gencot treats before- and after-units associated with a function declaration in a specific way and moves them to the target code part generated for the corresponding function definition. There they are placed around the comments associated with the function definition.

Gencot assumes, that only one declaration exists for each function definition. If there are more than one declarations in the C code the comments associated with one of them are moved to the definition, the comments associated with the other declarations are lost.

2.3 Processing Constants Defined as Preprocessor Macros

Often a C source file contains constant definitions of the form

```
#define CONST1 123
```

The C preprocessor substitutes every occurrence of the identifier `CONST1` in every C code after the definition by the value 123. This is a special application of the C preprocessor macro feature.

Constant definitions of this form could be used directly in Cogent, since they are also supported by the Cogent preprocessor. By transferring the constant definitions to the corresponding file `x-types.cogent` the identifiers are available in every Cogent file including `x-types.cogent`.

However, for generating proofs it should be better to use Cogent value definitions instead of having unrelated literals spread across the code. The Cogent value definition corresponding to the constant definition above can either be written in the form

```
#define CONST1 123
const1: U32
const1 = CONST1
```

preserving the original constant definition or directly in the shorter form

```
const1: U32
const1 = 123
```

Since the preprocessor name `CONST1` may also be used in `#if` directives, we use the first form. A typical pattern for defining a default value is

```
#if !defined(CONST1)
#define CONST1 123
#endif
```

This will only work if the preprocessor name is retained in the Cogent code.

If different C compilation units use the same preprocessor name for different constants, the generated Cogent value definitions will conflict. This will be detected and signaled by the Cogent compiler. Gencot does not apply any renaming to prevent these conflicts.

For the Cogent value definition the type must be determined. It may either be the smallest primitive type covering the value or it may always be U32 and, if needed, U64. The former requires to insert upcasts whenever the value is used for a different type. The latter avoids the upcast in most cases, however, if the value should be used for a U8 or U16 that is not possible since there is no downcast in Cogent. Therefore the first approach is used.

Constant definitions are also used to define negative constants sometimes used for error codes. Typically they are used for type `int`, for example in function results. Here, the type cannot be determined in the way as for positive values, since the upcast does not preserve negative values. Therefore we always use type U32 for negative values, which corresponds to type `int`. This may be wrong, then a better choice must be used manually for the specific case.

Negative values are specified as negative integer literals such as -42. To be used in Cogent as a value of type U32 the literal must be converted to an unsigned literal using 2-complement by: `complement(42 - 1)`. Since Cogent value definitions are translated to C by substituting the *expression* for every use, it should be as simple as possible, such as `complement 41` or even `0xFFFFFFD6` which is `4294967254` in decimal notation.

By convention, C preprocessor constant definitions use uppercase identifiers. In Cogent the name bound in a value definition must start with a lowercase letter, hence a conversion must be applied. Possible conversions are

1. convert all uppercase letters to lowercase
2. convert only some letters to lowercase at the beginning
3. replace a prefix, where the replacement starts with a lowercase letter

Solution 1 ist most readable in Cogent, however it may collide with the lowercase name when that is used for a C variable. Solution 2 reduces the possibility of a collision. Solution 3 is more eneral and corresponds to the way

how Gencot renames function and type names, therefore it is used. The default is to replace the empty prefix by `cogent_`, as for function names.

For comment processing, every preprocessor constant definition is treated as an unstructured source code part.

2.4 Processing Other Preprocessor Directives

A preprocessor directive always occupies a single logical line, which may consist of several actual lines where intermediate line ends are backslash-escaped. No C code can be in a logical line of a preprocessor directive. However, comments may occur before or after the directive in the same logical line. Therefore, every preprocessor directive may have an associated comment before-unit and after-unit, which are transferred as described in Section 2.2. Comments embedded in a preprocessor directive are discarded.

We differentiate the following preprocessor directive units:

- Constant definitions of the form `#define XXX val`, where `val` is a C constant or the defined name of a constant,
- all other macro definitions and `#undef` directives,
- conditional directives (`#if, #ifdef, #ifndef, #else, #elif, #endif`),
- include directives (quoted or system)
- all other directives, like `#error` and `#warning`

To identify constant definitions we assume that the names of constants are exactly the parameterless macro names starting with an uppercase letter. A C constant starts with a digit, a minus sign, or a single quote (string literals syntactically do not belong to the C constants). All constant definitions are processed as described in Section 2.3.

For comment processing every preprocessor directive is treated as an unstructured source code part.

2.4.1 Configurations

Conditional directives are often used in C code to support different configurations of the code. Every configuration is defined by a combination of preprocessor definitions. Using conditional directives in the code, whenever the code is processed only the code for one configuration is selected by the preprocessor.

In Gencot the idea is to process all configurations at the same time. This is done by removing the conditional directives from the code, process it, and re-insert the conditional directives into the generated Cogent code.

However, it may be the case that Gencot cannot process two configurations at the same time, because they contain conflicting information needed by Gencot. An example would be different definitions for the same type which shall be translated from C to a Cogent type by Gencot.

For this reason Gencot supports a list of conditions for which the corresponding conditional directives are not removed and thus only one configuration is processed at the same time. Then Gencot has to be run separately for every such configuration and the results must be merged manually.

Conditional directives which are handled this way are still re-inserted in the generated target code. This usually results in all branches being empty but the branches which correspond to the processed configuration. Thus the branches in the results from separate processing of different configurations can easily be merged manually or with the help of tools like diff and patch.

Keeping Conditional directives for certain configurations in the processed code makes only sense if the corresponding macro definitions which are tested in the directives are kept as well. Therefore also define directives can be kept. The approach in Gencot is to specify a list of regular expressions in the format used by awk. All directives which mach one of these regular expressions are kept in the code to be interpreted before processing the code.

2.4.2 Conditional Directives

Conditional directives are used to suppress some source code according to specified conditions. Gencot aims to carry over the same suppression to the generated code.

Associating Conditional Directives to Target Code

Conditional directives form a hierarchical block structure consisting of "sections" and "groups". A group consists of a conditional directive followed by other code. Depending on the directive there are "if-groups" (directives `#if, #ifdef, #ifndef`), "elif-groups" (directive `#elif`), and "else-groups" (directive `#else`). A section consists of an if-group, an optional sequence of elif-groups, an optional else-group, and an `#endif` directive. A group may contain one or more sections in the code after the leading directive.

Basically, Gencot transfers the structure of conditional directives to the target code. Whenever a source code part belongs to a group, the generated target code parts are put in the corresponding group.

This only works if the source code part structure is compatible with the conditional directive structure. In C code, theoretically, both structures need not be related. Gencot assumes the following restrictions: Every source code part which overlaps with a section is either completely enclosed in a group or contains the whole section. It may not span several groups or contain only a part of the section. If a source code part is structured, contained sections may only overlap with subparts, not with code belonging to the part itself.

Based on this assumption, Gencot transfers conditional directives as follows. If a section is contained in an unstructured source code part, its directives are discarded. If a section is contained in a structured source code part, its directives are transferred to the target code part. Toplevel sections which are not contained in a source code part are transferred to toplevel. Generated target code parts are put in the same group which contained the corresponding source code part.

It may be the case that for a structured source code part a subpart target must be placed separated from the target of the structured part. An example is a struct specifier used in a member declaration. In Cogent, the type definition generated for the struct specifier must be on toplevel and thus separate from the generated member. In these cases the condition directive structure must be

partly duplicated at the position of the subpart target, so that it can be placed in the corresponding group there.

Since the target code is generated without presence of the conditional directives structure, they must be transferred afterwards. This is done using the same markers `#ORIGIN` and `#ENDORIG` as for the comments. Since every conditional directive occupies a whole line, the contents of every group consists of a sequence of lines not overlapping with other groups on the same level. If every target code part is marked with the begin and end line of the corresponding source code part, the corresponding group can always be determined from the markers.

The conditional directives are transferred literally without any changes, except discarding embedded comments. For every directive inserted in the target code origin markers are added, so that its associated comment before- and after-unit will be transferred as well, if present.

2.4.3 Macro Definitions

Macro definitions are transferred literally, the intention is that they are used in a similar way in the Cogent code. If the definition occurs in a file `x.h` it is transferred to file `x-types.cogent` to a corresponding position, if it occurs in a file `x.c` it is transferred to file `x-impl.cogent` to a corresponding position.

This implies that the macro definitions are not available in the files `x-decls.cogent`, `x-globals.cogent` and in the files with antiquoted Cogent code. If they are used there (which mainly is the case if they are used in a conditional preprocessor directive which is transferred there), a manual solution is required.

The replacement text for a defined macro usually is C code. Thus the definition can normally not be used directly in the Cogent file, it must be adapted manually.

If different C compilation units use the same name for different macros, conflicts are caused in the integrated Cogent source. These conflicts are not detected by the Cogent compiler. A renaming scheme based on the name of the file containing the macro definition would not be safe either, since it breaks situations where a macro is deliberately redefined in another file. Therefore, Gencot provides no support for macro name conflicts, they must be detected and handled manually.

2.4.4 Include Directives

In C there are two forms of include directives: quoted includes of the form

```
#include "x.h"
```

and system includes of the form

```
#include <x.h>
```

Files included by system includes are assumed to be not a part of the translated <package>, therefore system include directives are discarded.

Translating Quoted Include Directives

Quoted include directives are always translated to the corresponding Cogent language include statement

```
include "x-types.h"
```

If the original include directive occurs in file `y.c` the translated statement is put into the files `y-impl.c` and `y-decls.c`. In the latter it may be required to make the types for parameters and function results available.

If the original include directive occurs in file `y.h` the translated statement is put into the file `y-types.h`.

It may be the case that a file `x.h` included by a quoted include is not a part of the translated <package>. In this case nevertheless the file `x-types.cogent` should be created and filled with abstract Cogent type definitions for all types used from `x.h`. This can be done either manually or using Gencot.

Including Files for Gencot Processing

When Gencot processes the C code in a source file, it may need access to information in files included by the source file. An example is a type definition for a type name used in the source file. Hence for C code processing Gencot always reads the source file together with all included files. Since in the source file all conditional preprocessor directives are removed, they must also be removed in the included files which belong to the same <package>. Gencot assumes these are the files included by a quoted include directive.

There are two possible approaches how this can be done.

The first approach is to use the C preprocessor which is invoked by language-c before parsing. It processes include directives as usual, hence it would be sufficient to leave the include directives in the source code when removing the other preprocessor directives. However, this would include the original `.h` files and *process* all preprocessor directives there, instead of removing them. The directives have to be removed from all included files in <package> in a separate step, then the include directives have to be modified to include the results of that step instead of the original files.

The second approach is to process all quoted include directives before the other directives are removed, resulting for every source file in a single file containing all included information and all other preprocessor directives. Then the directives are removed from this file with the exception of the system include directives. When the result is fed to the language-c parser its preprocessor will expand the system includes as usual, thus providing the complete information needed for processing the C code.

Gencot uses the second approach, since this way it can process every source file independently from previous steps for other source files and it needs no intermediate files which must be added to the include file path of the language-c preprocessor.

2.4.5 Other Directives

All other preprocessor directives are discarded. Gencot displays a message for every discarded directive.

2.5 Mapping C Datatypes to Cogent Types

Here we define rules how to map common C types to binary compatible Cogent types. Since the usefulness of a mapping also depends on the way how values of the type are processed in the C program, the rules here are only defaults which are used, if no better mapping is suggested by the way of value processing.

Numerical Types

The Cogent primitive types are mapped to C types in `cogent/lib/cogent-defns.h` which is included by the Cogent compiler in every generated C file with `#include <cogent-defns.h>`. The mappings are:

```
U8 -> unsigned char
U16 -> unsigned short
U32 -> unsigned int
U64 -> unsigned long long
Bool -> struct bool_t { unsigned char boolean }
String -> char*
```

The inverse mapping can directly be used for the unsigned C types. For the corresponding signed C types to be binary compatible, the same mapping is used. Differences only occur when negative values are actually used, this must be handled by using specific functions for numerical operations in Cogent.

The C99 standard defines the "exact-width integer types", such as `uint32_t`, provided by `#include <stdint.h>`. They can safely be mapped to the corresponding primitive Cogent types.

Type `size_t` is defined by the C99 standard to be the type of all values returned by the `sizeof` operator. On 64 Bit architectures this is equivalent to type `long long` and corresponds to the Cogent type `U64`.

Together we have the following mappings:

```
char, unsigned char, uint8_t -> U8
short, unsigned short, uint16_t -> U16
int, unsigned int, uint32_t -> U32
long long, unsigned long long, uint64_t -> U64
size_t -> U64
```

Note that the mappings for `int`, `unsigned int` and `size_t` are architecture dependent. Here we ignore this dependency and always use the mapping for 64 Bit architecture. An architecture independent approach would be possible using a conditional mapping with the help of preprocessor `#if` directives.

Structure and Union Types

A C structure type of the form `struct { ... }` is equivalent to a Cogent unboxed record type `#{ ... }`. The Cogent compiler translates the unboxed record type to the C struct and maps all fields in the same order. If every C field type is mapped to a binary compatible Cogent field type bot types are binary compatible as a whole.

The field names have local scope in C and in Cogent, they can be transferred without changes. However, in Cogent all field names must begin with a lowercase

letter. If the C field name begins with an uppercase letter (which is unusual) Gencot applies a prefix mapping where the default is to prepend the prefix `cogent_`.

For a `const` qualified C structure type the fields may not be modified. This is equivalent to the behavior of an unboxed record in Cogent, the same mapping is used here.

A C structure may contain bit-fields where the number of bits used for storing the field is explicitly specified. Gencot maps every consecutive sequence of bit-fields to a single Cogent field with a primitive Cogent type. The Cogent type is determined by the sum of the bits of the bit-fields in the sequence. It is the smallest type chosen from `U8, U16, U32, U64` which is large enough to hold this number of bits. ***–> test whether this is correct. The name of the Cogent field is `cogent_bitfield`<n> where <n> is the number of the bit-field sequence in the C structure. Gencot does not generate Cogent code for accessing the single bit-fields, if needed this must be done manually in Cogent. However, Gencot adds comments after the Cogent bitfield showing the original C bit-field declarations.

A C union type of the form `union { ... }` is not equivalent to any type generated by the Cogent compiler. The semantic equivalent would be a Cogent variant type. However, the Cogent compiler translates every variant type to a `struct` with a field for an `enum` covering the variants, and one field for every variant. Even if a variant is empty (has no additional fields), in the C `struct` it is present with type `unit_t` which has the size of an `int`. Therefore Gencot maps every union type to an abstract Cogent type.

Since an abstract Cogent type is always linear, Gencot maps a `const` qualified C union type to the corresponding readonly type.

Together we have the mapping rules:

```
struct s, const struct s -> unboxed record
union s -> abstract type S
const union s -> abstract type S!
```

Pointer Types

In general, a C pointer type `t*` is the kind of types targeted by Cogent linear types. The linear type allows the Cogent compiler to statically guarantee that pointer values will neither be duplicated nor discarded by Cogent code, it will always be passed through.

If a pointer points to a C `struct` there is additional support for field access available in Cogent by mapping the pointer to a Cogent boxed record type. For all other pointer types the Cogent type must be abstract, then the pointer is opaque in Cogent code, it can only be passed around but no operations can be performed directly in Cogent. All processing must be implemented externally by an abstract data type.

Since a C `void*` pointer type is also opaque in C, it corresponds directly to a Cogent abstract type.

For a C pointer there are two cases of readonly types. A "pointer to const" type of the form `const t*` means that the data structure pointed to cannot be modified, whereas the pointer itself can be replaced (if, e.g., it is stored in a variable of that type). A "constant pointer" type of the form `t* const` instead

means that the pointer itself cannot be modified, whereas the data structure pointed to can. In Cogent no difference is made between the pointer and its target, both together are always immutable, which corresponds to the combination of both C cases. However, for linear types, Cogent internally supports modification of the data structure using `put` and `take`. The Cogent readonly types prevent this, therefore they correspond to the first case in C. The second case is always respected by Cogent, if a pointer should be replaced, this must be implemented by an adequate processing approach in Cogent.

Together we have the basic mapping rules:

```
struct s *, struct s * const -> boxed record
void *, void * const -> abstract type
t *, t * const -> abstract type T
const t *, const t* const -> type T!
```

In C a `t*` pointer can be assigned to a `const t*` pointer, but not vice versa. This corresponds to the Cogent property that a linear value may be made readonly, but not the other way round.

Array Types

The implementation of a C array type `t[n]` is equivalent to that of the pointer type `t*`. Whereas the type `t*` has the meaning of a pointer to a single `t` instance, type `t[n]` has the meaning of a pointer to a consecutive sequence of n `t` instances.

Basically, Cogent does not support accessing elements by an index value of an array represented by a pointer. This is an important security feature since the index value is computed at runtime and cannot be statically compared to the array length by the compiler. Therefore, a C array type can only be mapped to an abstract type in Cogent, which prevents accessing its elements in Cogent code. Element access must be implemented externally with the help of abstract functions.

The Cogent standard library contains three abstract data types for arrays (`Wordarray, Array, UArray`. However, they cannot be used as a binary compatible replacement for C arrays, because they are implemented by pointers to a `struct` containing the array length together with the pointer to the array elements. Only if the C array pointer is contained in such a struct, it is possible to use the abstract data types. In existing C code the array length is often present somewhere at runtime, but not in a single `struct` directly before the array pointer.

As of December 2018 there is an experimental Cogent array type written `T[n]`. It is binary compatible with the C array type `t[n]`. It is not linear, however it only supports read access to the array elements, the element values cannot be replaced. Thus it can be used as replacement for a pure abstract type, if the array is never modified and if it does not contain any pointers (directly or indirectly). If it is modified, replacing elements can be implemented externally with the help of abstract functions.

An array of type `const t []` cannot be modified, hence it is fully supported by the corresponding Cogent array type.

The primitive Cogent type `String` is mapped to C type `char*`. It is used to pass the usual null terminated C strings through Cogent code. The characters

in the string cannot be accessed in Cogent, neither for replacing them nor for reading them. Thus, if the characters are not accessed, Cogent type `String` is a useful mapping for all kinds of C character arrays.

In C, arrays of type `t[n]` can also be accessed through a Pointer of type `t*` using pointer arithmetics. In this case type `t*` can be mapped to Cogent in the same way as C type `t[n]`.

In C the incomplete type `t[]` can be used in certain places. It may be completed statically, e.g. when initialized. Then the number of elements is statically known and the type can be mapped like `t[n]`. If the number of elements is not statically known (this is also often the case when type `t*` is used for an array) the type cannot be mapped to a Cogent array, it must be mapped to an abstract type in the same way as type `t*`.

The Cogent array type `T[1]` can be used as an alternative mapping for arbitrary pointer types which only point to a single element, if it does not contain pointers. Then it becomes possible to differentiate between the pointer and the pointed value in Cogent code. Since the array cannot be modified, this fully supports only the functionality of type `const t*`. Semantically, in Cogent there should be no difference to the non-linear type T, however, for binary compatibility the difference becomes relevant.

Together we have the following mapping rules for C arrays with element type `el`. Here, C type `el*` is a pointer type used to access an array.

```
char[n], char[], unsigned char[n], unsigned char[],
   const char[n], const char[],
   const unsigned char[n], const unsigned char[] -> String
char*, unsigned char*, const char*, const unsigned char*
   -> String, if used to access a C string
const el[n], const el[], const el*
   -> El[n], if n can be statically determined
const el[], const el*
   -> abstract type, if array size cannot be statically determined
el[n], el[], el* -> abstract type
const t* -> T[1], if t contains no pointers
```

Enumeration Types

A C enumeration type of the form `enum e` is a subset of type `int` and declares enumeration constants which have type `int`. According to the C99 standard, an enumeration type may be implemented by type `char` or any integer type large enough to hold all its enumeration constants.

A natural mapping for C enumeration types would be Cogent variant types. However, the C implementation of a Cogent variant type is never binary compatible with an integer type (see above).

Therefore C enumeration types must be mapped to a primitive integer type in Cogent. Depending on the C implementation, this may always be type `U32` or it may depend on the value of the last enumeration constant and be either `U8`, `U16`, `U32`, or maybe even `U64`. Under Linux, both cc and gcc always use type `int`, independent of the value of the last enumeration constant. Therefore we always map enumeration types to Cogent type `U32`.

The enumeration constants must be mapped to Cogent constant definitions of the corresponding type. In C the value for an enumeration constant may be explicitly specified, this can easily be mapped to the Cogent constant definitions.

The rule for mapping enumeration types is

```
enum e -> U32
```

An enumeration declaration of the form `enum e {C1, C2, C3=5, C4}` is translated as

```
cogent_C1: U32
cogent_C1 = 0
cogent_C2: U32
cogent_C2 = 1
cogent_C3: U32
cogent_C3 = 5
cogent_C4: U32
cogent_C4 = 6
```

Note that the C constant names are replaced in the same way as for preprocessor constants.

Function Types

C function types of the form `t (...)` are used in C only for declaring or defining functions. In all other places they are either not allowed or automatically adjusted to the corresponding function pointer type of the form `t (*)(...)`.

In Cogent this distinction does not exist. A Cogent function type of the form `T1 -> T2` is used both when defining functions and when binding functions to variables. If used in a function definition, it is mapped by the Cogent compiler to the corresponding C function type, when used in other places it is mapped to the corresponding C function pointer type.

Binary compatibility is only relevant when a function is stored, then it is always a function pointer. All function pointers are of the same size, hence a C function pointer type can be mapped to an arbitrary Cogent function type. Of course, to be useful the types of the parameters and result should be mapped as well. In Cogent every function has only one parameter. To be mapped to Cogent, the parameters of a C function with more than one parameter must be aggregated in a tuple or in a record. A C function type `t (void)` which has no parameters is mapped to the Cogent function type `() -> T` with a parameter of unit type.

The difference between tuple and record is that the fields in a record are named, in a tuple they are not. In a C function definition the parameters may be omitted, otherwise they are specified with names in a prototype. In C function types the names of some or all parameters may be omitted, specifying only the parameter type.

It would be tempting to map C function types to Cogent functions with a record as parameter, whenever parameter names are available in C, and use a tuple as parameter otherwise. However, in C it is possible to assign a pointer to a function which has been defined with parameter names to a variable where the type does not provide parameter names such as in

```
int add (int x, int y) {...}
int (*fun)(int,int);
fun = &add;
```

This case would result in Cogent code with incompatible function types.

For this reason we always use a tuple as parameter type in Cogent. Cogent tuple types are equivalent, if they have the same number of fields and the fields have equivalent types. To preserve the C parameter names in a function definition, the parameter can be matched with a tuple pattern containing variables of these names as fields.

C function types where the parameters are omitted, such as in `t ()` or where a variable number of parameters is specified such as in `t (...)` cannot be mapped to a Cogent function type in this way. They can only be mapped using an abstract type as parameter type. This can again lead to incompatible Cogent types if a function pointer is assigned where parameters have been specified, these cases must be treated manually in specific ways. Gencot maps these function types to an Cogent types with an abstract parameter type.

Together the rules for mapping function types are

```
t(t1, ..., tn), t (*)(t1, ..., tn)
  -> (T1, ..., Tn) -> T
t(void), t (*)(void)
  -> () -> T
t(), t(*)(), t(t1,...,tn,...), t (*)(t1,...,tn,...)
  -> P -> T, where P is abstract
```

2.6 Processing C Declarations

A C declaration consists of zero or more declarators, preceded by information applying to all declarators together. Gencot translates every declarator to a separate Cogent definition, duplicating the common information as needed. The Cogent definitions are generated in the same order as the declarators.

A C declaration may either be a `typedef` or an object declaration. For every declarator in a typedef Gencot generates a Cogent type definition. For every declarator in an object declaration, Gencot generates a Cogent field definition, variable binding, or nothing, depending on the position of the object definition and the type derived by the declarator.

Additionally, whenever a struct-or-union-specifier or enum-specifier occurring in the C declaration has a body, a Cogent type definition is generated for the corresponding type. A C declaration may contain atmost one struct-or-union-specifier or enum-specifier directly. Here we call such a specifier the "full specifier" of the declaration, if it has a body. In the generated Cogent code the type definition for the full specifier is positioned after the definitions generated for the declarators.

A C declaration may occur on toplevel or embedded in the body of a C function definition. If the C declaration is contained in file `x.h` it must be on toplevel, the generated Cogent definitions are put on toplevel in file `x-types.cogent`. If the C declaration is on toplevel in file `x.c` the generated Cogent definitions are put on toplevel in file `x-impl.cogent`. Since the position of a toplevel Cogent definition is irrelevant for its availability, and since all Cogent code is included

in one single source, the generated Cogent toplevel definitions are available everywhere in the Cogent code.

If a C declaration occurs embedded in a C function body all geretade Cogent type definitions are put after the target code part generated for the function body, since in Cogent type definitions are only allowed on toplevel position.

A C declaration may also occur as a part of another declaration: either in a field declaration in the full specifier or in the parameter list of a function type in a declarator. In the former case the declarators of the inner declaration are translated to Cogent field definitions in the Cogent type definition generated for the full specifier. The Cogent type definitions generated for the full specifiers of the inner declarations are (recursively) positioned immediately after the type definition generated for the full specifier of the outer declaration, in the order of the inner declarations.

If a full specifier occurs in a parameter declaration its scope is restricted to the parameter list. This implies that it cannot be used as type for an actual argument for the parameter and is thus useless. Therefore Gencot assumes that parameter declarations never contain a full specifier. Moreover, a parameter declaration must contain exactly one declarator. This declarator is translated to a Cogent field definition in the parameter tuple type if the function type owning the parameter list is translated to a Cogent function type.

2.6.1 Relating Comments

A declaration is treated as a structured source code part. The subparts are the full specifier, if present, and all declarators. Every declarator includes the terminating semicolon, thus there is no main part code between or after the declarators.

The target code part generated for a declaration is the sequence of target code parts generated for the declarators, including the target code part generated for the full specifier if that immediately follows. This implies that the before-unit of the declaration is put immediately before the before-unit of the first declarator.

A declarator may derive a function type specifying a parameter-type-list. If that list is not `void`, the declarator is a structured source code part with the parameter-declarations as embedded subparts. Every parameter-declaration includes the separating comma after it, if another parameter-declaration follows, thus there is no main part code between the parameter-declarations. In all other cases a declarator is an unstructured source code part.

2.6.2 Typedef Declarations

For a C typedef declaration Gencot generates a separate toplevel Cogent type definition for every declarator.

For every declarator a C type is determined from the declaration specifiers together with the derivation specified in the declarator. As described in Section 2.5, either a Cogent type expression is determined from this C type, or the Cogent type is decided to be abstract. Depending on this decision, the Cogent type name to be defined is generated from the C identifier in the declarator as described in Section 2.1.2.

2.6.3 Object Declarations

For a C object declaration which is not embedded in another declaration, Gencot generates a separate Cogent object definition for every declarator, if the type derived in the declarator is not a function type. Declarators with a function type declare functions which may be defined later, this is not possible in Cogent. All other declarators declare global variables ***–> generate additional files x-global.cogent, put global variables as field definitions there, define record type SysState in <package>.cogent and include all x-global.cogent as fields. ***–> initializers must be handled!

For a C object declaration which is embedded in another declaration, Gencot generates a separate Cogent field definition for every declarator. This is a named record field definition if the declaration is embedded in the body of a struct-or-union-specifier, it is an unnamed tuple field definition if the declaration is embedded in the parameter-type-list of a function type. In the first case declarators with function type are not allowed, in the second case they are adjusted to function pointer type. In both cases the Cogent field type is determined from the declarator's C type as described in Section 2.5. In the case of a named record field the Cogent field name is determined as described in Section 2.5.

2.6.4 Struct or Union Specifiers

The relevant parts of a struct-or-union-specifier with a body are the struct-declarations in the body.

2.6.5 Enum Specifiers

The subparts parts of an enum-specifier with a body are the enumerators in the body. Enumerators are unstructured source code parts.

Chapter 3

Implementation

Gencot is implemented by a collection of unix shell scripts using the unix tools `sed`, `awk`, and the C preprocessor `cpp` and by Haskell programs using the C parser `language-c`.

Every step is implemented as a Unix filter, reading from standard input and writing to standard output. A filter may read additional files when it merges information from several steps. The filters can be used manually or they can be combined in scripts or makefiles. Gencot provides some predefined scripts for filter combinations.

Since the `language-c` parser does not support parsing preprocessor directives and C comment, the general approach is to remove both from the source file, process them separately, and re-insert them into the generated files.

3.1 Comments

In a first step all comments are removed from the C source file and are written to a separate file. The remaining C code is processed by Gencot. In a final step the comments are reinserted into the generated code.

Additional steps are used to move comments from function declarations to function definitions.

3.1.1 Filters and Positions

The filter `gencot-selcomments` selects all comments from the input and writes them to the output. The filter `gencot-remcomments` removes all comments from the input and writes the remaining code to the output. The filter `gencot-mrgcomments <file>` merges the comments in `<file>` into the input and writes the merged code to the output. <file> must contain the output of `gencot-selcomments` applied to a code X, the input must have been generated from the output of `gencot-remcomments` applied to the same code X.

To be merged, it must be possible to relate the comments to the generated target code parts. The general approach is to insert markers in the input of `gencot-mrgcomments`. The markers have one of the following forms, each in a single separate line:

```
#ORIGIN <bline>
```

```
#ORIGIN <bline> <aline>
#ENDORIG <aline>
```

where `<bline>` and `<aline>` are line numbers. A marker tells `gencot-mrgcomments` to prepend to the following line the before-unit of original source line `<bline>` and/or append to the following line the after-unit of original source line `<aline>`.

The filters `gencot-selcomments` and `gencot-remcomments` never delete lines, hence in their output the original line numbers can still be determined.

To distinguish before-units and after-units, `gencot-selcomments` inserts a separator between them. The separator consists of a newline followed by `*/_`. It is constructed in a way that it cannot be a part of or overlap with a comment and to be easy to detect when processing the output of `gencot-selcomments` line by line. The newline and the `*/` would end any comment. The underscore (any other character could have been used instead) distinguishes the separator from a normal end-of comment, since `gencot-selcomments` never inserts an underscore immediately after a comment.

The separator is inserted after every unit, even if the unit is empty. The first unit in the output of `gencot-selcomments` is always a before-unit.

3.1.2 Filter `gencot-remcomments`

The filter for removing comments is implemented using the C preprocessor with the option `-fpreprocessed`. With this option it removes all comments, however, it also processes and removes `#define` directives. To prevent this, a sed script is used to insert an underscore `_` before every `#define` directive which is only preceded by whitespace in its line. Then it is not recognized by the preprocessor. Afterwards, a second sed script removes the underscores.

Instead of an underscore an empty block comment could have been used. This would have the advantage that the second sed script is not required, since the empty comments are removed by the preprocessor. The disadvantage is that the empty comment is replaced by blanks. The resulting indentation does not modify the semantics of the `#define` statenments but it looks unusual in the Cogent code.

The preprocessor also removes `#undef` directives, hence they are treated in the same way.

The preprocessor preserves all information about the original source line numbers, to do so it may insert line directives of the form `# <linenumber> <filename>`. They must be processed by all following filters. The Haskell C parser `language-c` processes these line directives.

3.1.3 Filter `gencot-selcomments`

The filter for selecting comments is implemented as an awk script. It scans through the input for the comment start sequences `/*` and `//` to identify comments.

To keep it simple, the cases when the comment start sequences can occur in C code parts are ignored. This may lead to additional or extended comments, which must be corrected manually. It never leads to omitted comments or missing comment parts. Note that `gencot-remcomments` always identifies

comments correctly, since there comment detection it is implemented by the C preprocessor.

When in an input line code is found outside of comments all this code with all embedded comments is replaced by the separator. Only the comments before and after the code are copied to the output, if present. Note, that the separator includes a newline, hence every source line with code outside of comments produces two output lines.

An after-unit consists of all comments after code in a line. The last comment is either a line comment or it may be a block comment which may include following lines. After this last comment the after-unit ends and a separator is inserted.

All whitespace in and between comments and before the first comment in a before-unit is preserved in the output, including empty lines. After a before-unit only empty lines are preserved. Whitespace around code is typically used to align code and comments, this must be adapted manually for the generated target code. Whitespace after an after-unit is not preserved since the last comment in an after unit in the target code is always followed by a newline.

State Machine

The implementation processes the input line by line using a finite state machine. It uses the variables `before` and `after` to collect block comments at the beginning and end of the current line, initially both are empty. The collect action appends the input from the current position up to and including the next found item in the specified variable. The separate action appends the separator to the specified variable. The output action writes the specified content to the output. The newline action advances to the beginning of the next line and clears `before`.

The state machine has the following states, nocode is the initial state:

nocode If next is

- **end-of-line** output(`before`); newline; goto nocode
- **block-comment-start** collect(`before`); goto nocode-inblock
- **line-comment-start** collect(`before`); output(`before` + line-comment); newline; goto nocode
- **other-code** separate(`before`); clear(`after`); goto code

nocode-inblock If next is

- **end-of-line** collect(`before`); output(`before`); newline; goto nocode-inblock
- **block-comment-end** collect(`before`); goto nocode

code If next is

- **end-of-line** output(`before` + separator); newline; goto nocode
- **block-comment-start** append comment-start to `after`; goto code-inblock
- **line-comment-start** output(`before` + line-comment + separator); newline; goto nocode

code-inblock If next is

end-of-line collect(`after`); output(`before` + `after`); newline; goto aftercode-inblock

block-comment-end collect(`after`); goto code-afterblock

code-afterblock If next is

end-of-line output(`before` + `after` + separator); newline; goto nocode

block-comment-start collect(`after`); goto code-inblock

line-comment-start collect(`after`); output(`before` + `after` + line-comment + separator); newline; goto nocode

other-code clear(`after`); goto code

aftercode-inblock If next is

end-of-line collect(`before`); output(`before`); newline; goto aftercode-inblock

block-comment-end collect(`before`); separate(`before`); goto nocode

3.1.4 Filter `gencot-mrgcomments`

The filter for merging comments into the target code is implemented as an awk script. It consists of a BEGIN rule and a line rule. The BEGIN rule reads the <file> line by line and collects before- and after-units as strings in the arrays `before` and `after`. The arrays are indexed with the line number of the separator between before- and after-unit.

The line rule tests for the markers. It replaces every `#ORIGIN` marker by the `before` entry for `<bline>` and appends the following line and, if `<aline>` is present, appends the `after` entry for `<aline>`. It removes every `#ENDORIG` marker and appends the `after` entry for `<aline>`. All other input lines are written to the output without modification.

3.1.5 Function Declaration Comments

To safely detect C function declarations and C function definitions Gencot uses the language-c parser. Processing the function decalaration comments is implemented by the following filter steps.

Filter `gencot-deccomments`

The filter `gencot-deccomments` parses the input. For every function declaration it outputs a line

```
#FUNDEC <function name> <bline> <aline>
```

where `<function name>` is the name of the declared function, `<bline>` is the original source line number where the function declaration begins and `<aline>` is the original source line number where the function declaration ends. (In many cases the function declaration will be a single line and `<bline>` and `<aline>` will be the same.)

Filter `gencot-funcomments`

The filter `gencot-funcomments <file>` processes the output of `gencot-deccomments` as input. For every pair of lines as above, it retrieves the before-unit of `<bline>` and the after-unit of `<aline>` from `<file>` and stores them in the files `before-<function name>.comment` and `after-<function name>.comment` in the current directory. The content of `<file>` must be the output of `gencot-selcomments` applied to the same original source from which the input of `gencot-deccomments` has been derived.

The filter is implemented as an awk script. It consists of a BEGIN rule reading `<file>` in the same way as `gencot-mrgcomments`, and a rule for lines starting with `#FUNDEC`. For every such line it writes the associated comment units to the comment files. A comment file is even written if the comment unit is empty.

Filter `gencot-defcomments`

For inserting the comments around the target code parts generated from a function definition, Gencot uses the markers

```
#FUNDEF before-<function name>
#FUNDEF after-<function name>
```

The markers must be inserted by the filter which generates the function definition target code.

The filter `gencot-defcomments <dir>` replaces every marker line in its input by the content of the corresponding `.comment` file in directory `<dir>` and writes the result to its output.

It is implemented as an awk script with a single rule for all lines. If the line starts with `#FUNDEF` it is replaced by the content of the corresponding file in the output. All other lines are copied to the output without modification.

3.2 Preprocessor Directives

3.2.1 Filters for Processing Steps

Directive processing is done for the output of `gencot-remcomments`. All comments have been removed. However, there may be line directives present.

The filter `gencot-selpp` selects all preprocessor directives and copies them to the output without changes. All other lines are replaced by empty lines, so that the original line numbers for all directives can still be determined.

The filter `gencot-include <dirlist>` processes all quoted include directives and replaces them (transitively) by the content of the included file. Line directives are inserted at the begin and end of an included file, so that for all code in the output the original source file name and line number can be determined. The `<dirlist>` specifies the directories to search for include files.

The filter `gencot-rempp <file>` removes all preprocessor directives from its input, replacing them by empty lines. All other lines are copied to the output without modification. It is intended to be applied to the output of `gencot-include`. If `<file>` is specified it must contain a list of regular expressions for directives which shall be retained.

How the directives are processed depends on the kind of directives (see Section 2.4). For every kind `X` from `const`, `cond`, `macro`, `incl` Gencot provides the processing filter `gencot-prcX`. Since also the way of merging the results into the Cogent code depends on the kind, Gencot provides the merging filters `gencot-mrgX <file>`. It merges the content in `<file>` into the input and writes the merged code to the output. `<file>` must contain the output of `gencot-prcX` which originated from the same file as the input.

3.2.2 Separating Directives

Gencot supports to keep some directives in the output of `gencot-rempp` to handle cases where the C code of different groups in a section causes conflicts. These conditional directives are still selected by `gencot-selpp` and re-inserted by `gencot-mrgcond`.

Filter `gencot-selpp`

The filter for selecting preprocessor directives from the input for separate processing and insertion into the generated target code is implemented as an awk script.

It detects all kinds of preprocessor directives, which always begin at the beginning of a separate line. A directive always ends at the next newline which is not preceded by a backslash
. All corresponding lines are copied to the output without modifications with the exception of line directives.

Line directives in the input are expanded to the required number of empty lines which have the same effect. This is done to simplify reading the input for all `gencot-prcX` filters.

Every other input line is replaced by an empty line in the output.

Filter `gencot-include`

The filter for expanding the include directives is implemented as an awk script, heavily inspired by the "igawk" example program in the gawk infofile, edition 4.2, in Section 11.3.9.

As argument it expects a directory list specified with ":" as separator. The list corresponds to directories specified with the `-I` cpp option, it is used for searching included files. All directories for searching included files must be specified in the arguments, there are no defaults.

Similar to cpp, a file included by a quoted directive is first searched in the directory of the including file. If not found there, the argument directory list is searched.

Since the input of `gencot-include` is read from standard input it is not associated with a directory. Hence if files are included from the same directory, that directory must also be specified explicitly in an argument directory list.

In the output line directives are generated as follows.

If the first line of the input is a line directive, it is copied to the output. Otherwise the line directive

```
# 1 "<stdin>"
```

is prepended to the output.

If after a generated line directive with file name ẗfff¨ the input line NNN contains the directive

```
#include "filepath"
```

the directive is replaced in the output by the lines

```
# 1 "dir/filepath" 1
<content of file filepath>
# NNN+1 "fff" 2
```

The C preprocessor does not prevent a file from being included multipple times. Usually, C include files use an ifdef directive around all content to prevent multiple includes. The `gencot-include` filter does not interprete ifdef directives, instead, it simply prevents multiple includes for all files independent from their contents, only based on their full file pathnames. To mimic the behavior of cpp, if a file is not include due to repeated include, the corresponding line directives are nevertheless generated in the form

```
# 1 "dir/filepath" 1
# NNN+1 "fff" 2
```

The ẗdir/¨ prefix in the line directives for included files is determined as follows. If the included file has been found in the directory of its includer, the directory pathname is constructed from ẗfff¨ by taking the pathname up to and including the last "/" (if present, otherwise the prefix is empty). If the included file has been found in a directory from the argument directory list the directory pathname is used as specified in the list.

Filter `gencot-rempp`

The filter for removing preprocessor directives from its input is implemented as an awk script. Basically, it replaces lines which are a part of a directive by empty lines. However, there are the following exceptions:

- line directives are never removed, they are required to identify the position in the original source during code processing.
- system include directives are never removed, they are inteded to be interpreted by the language-c preprocessor to make the corresponding information available during code processing. Since it is assumed that all quoted include directives have already been processed by `gencot-include`, simply all include directives are retained.
- directives which match a regular expression from a specified list are not removed, they are intended to be interpreted by the language-c preprocessor to suppress information which causes conflicts during code processing.

For conditional directives always all directives belonging to the same section are treated in the same way. To retain them the first directive (`#if`, `#ifdef`, `#ifndef`) must match a regular expression in the list. For all other directives of a section (`#else`, `#elif`, `#endif`) the regular expressions are ignored.

The regular expressions are specified in the argument file line by line. An example file content is

```
^[[:blank:]]*#[[:blank:]]*if[[:blank:]]+!?[[:blank:]]*defined\(SUPPORT_X\)
^[[:blank:]]*#[[:blank:]]*define[[:blank:]]+SUPPORT_X
^[[:blank:]]*#[[:blank:]]*undef[[:blank:]]+SUPPORT_X
```

It retains all directives which define the macro `SUPPORT_X` or depend on its definition.

3.2.3 Processing Directives

Processing Constants Defined as Preprocessor Macros

We provide the script `convert-const.csh` for automating this task. If comments should be also be converted to Cogent the script can be used together with the script `convert-comment.csh`.

Processing Other Preprocessor Directives

The line numbers for positions count actual lines. Therefore the position of a preprocessor directive is specified by its starting line and its ending line.

3.2.4 Merging Directive Processing Results

3.2.5 Filter `gencot-mrgpp`