

HoBit Report on Gencot Development

Gunnar Teege

January 18, 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 performing 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 Mapping Names from C to Cogent

Names used in the C code shall be translated to similar names in the Cogent code, since they usually are descriptive for the programmer. Ideally, the same names would be used. However, this is not possible, since Cogent differentiates between uppercase and lowercase names and uses them for different purposes. Therefore, atleast the names in the “wrong” case need to be mapped.

Additionally, when the Cogent compiler translates a Cogent program to C code, it transfers the names without changes to the names for the corresponding C items. We will see below, that often these generated C items are needed additionally to the original C item which has been translated to Cogent by Gencot. If Gencot uses the same name in Cogent, this would cause a name conflict in the code generated by the Cogent compiler.

For this reason, Gencot uses name mapping schemas mapping all kinds of names which can cause such a conflict to a different, but similar name in Cogent. Generally, this is done by substituting a prefix of the name.

Often, a `<package>` uses one or more specific prefixes for its names, at least for names with external linkage. In this case Gencot should be able to substitute these prefixes by other prefixes specific for the Cogent translation of the `<package>`. Therefore, the Gencot name mapping is configurable. For every

<package> a set of prefix mappings can be provided which is used by Gencot. Two separate mappings are provided depending on whether the Cogent name must be uppercase or lowercase, so that the target prefixes can be specified in the correct case.

If a name must be mapped by Gencot which has neither of the prefixes in the provided mapping, it is mapped by prepending the prefix `cogent_` or `Cogent_`, depending on the target case.

Name Kinds in C

In C code the tags used for struct, union and enum declarations constitute an own namespace separate from the “regular” identifiers. These tags are mapped to Cogent type names by Gencot and could cause name conflicts with regular identifiers mapped to Cogent type names. To avoid these conflicts Gencot maps tags by prepending the prefixes `Struct_`, `Union_`, or `Enum_`, respectively, after the mapping described above. Since tags are always translated to Cogent type names, which must be uppercase, only one case variant is required.

Member names of C structs or unions are translated to Cogent record field names. Both in C and Cogent the scope of these names is restricted to the surrounding structure. Therefore, Gencot normally does not map these names and uses them unmodified in Cogent. However, since Cogent field names must be lowercase, Gencot applies the normal mapping for lowercase target names to all uppercase member names (which in practice are unusual in C).

C function parameter names are translated to Cogent variable names bound in the Cogent function body expression. Hence, both in C and Cogent the scope of these names is restricted to the function body. They are treated by Gencot in the same way as member names and are only mapped if they are uppercase in C, which is very unusual in practice.

The remaining names in C are type names, tags, function names, enum constant names, and names for global and local variables. Additionally, there may be C constant names defined by preprocessor macro directives. Local variables only occur in C function bodies which are not translated by Gencot. The other names are always mapped by Gencot, irrespective whether they have the correct case or not. The reasons are explained in Section 2.1.2 below.

Names with internal linkage

In C a name may have external or internal linkage. A name with internal linkage is local to the compilation unit in which it is defined, a name with external linkage denotes the same item in all compilation units. Since the result of Gencot’s translation is always a Cogent program which is translated to a single compilation unit by the Cogent compiler, names with internal linkage could cause conflicts when they origin in different C compilation units.

To avoid these conflicts, Gencot uses a name mapping scheme for names with internal linkage which is based on the compilation unit’s file name. Names with internal linkage are mapped by substituting a prefix by the prefix `local_x_` where `x` is the basename of the file which contains the definition, which is usually a file `x.c`. The default is to substitute the empty prefix, i.e., prepend the target prefix. The mapping can be configured by specifying prefixes to be substituted. This is motivated by the C programming practice to sometimes also

use a common prefix for names with internal linkage which can be removed in this way.

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 and must be handled manually.

Introducing Names for Abstract Types

For some C types no corresponding type can be defined in Cogent. In these cases an abstract type is defined in Cogent which references the C type. The abstract type in Cogent is specified by a name only. However, there need not exist a name for the C type, hence the name of the abstract type cannot be generated by mapping the C type name.

Unnamed C types are tagless struct/union/enum types and all derived types, i.e., array types, pointer types and function types. For tagless struct and enum types and for function types and some pointer types corresponding Cogent types can be defined. For the remaining cases new abstract type names must be generated.

All these generated type names belong to the same name space in Cogent. To be able to process every source file independently from all other source files, Gencot uses source file specific names when it generates new abstract type names. The generated names have the forms

```
<kind><nr>_x_h  
<kind><nr>_x_c
```

where the suffix is constructed from the name `x.h` or `x.c` of the source file. `<kind>` is one of `Union`, `Array`, or `Pointer`, according to the kind of the C type, and `<nr>` is a counter for all names of `<kind>` used in the corresponding source file, starting at 1.

Note that these new abstract type names could still cause conflicts with mapped type names. These conflicts can be avoided if no configured mapping prefix starts with one of the `<kind>` strings.

2.1.2 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 conflict with the name of the wrapper. This is guaranteed by the Gencot renaming scheme as described in Section 2.1.1.

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.

To use the exit wrapper from Cogent, a corresponding abstract function definition must be present in Cogent.

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 be 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).

Conflicts for names with internal linkage originating in different C compilation units are avoided by Gencot’s name mapping scheme as described in Section 2.1.1.

External Name References

To successfully compile the Cogent compilation unit all referenced identifiers must be declared in the C code. Those references which are used in the generated Cogent code must additionally be defined in Cogent. A non-local reference in a C source file is every identifier which is used in the file but not defined or declared in the same file.

In the original C source for every non-local reference there must be a declaration or definition present in one of the included files (its “origin file”). If the origin file of a non-local reference is a file which has already been translated by Gencot, the required information about the identifier is already present in the Cogent compilation unit. If the origin file of a non-local reference has not yet been translated, or is not a part of the `<package>` (which normally is the case for all system includes), we call it an “external reference”. For external references additional information must be created and made available in the Cogent compilation unit.

On the C level the information is provided by simply including the origin files of all external references. On the Cogent level the information is provided as follows.

- If the external reference is a type name or a struct/union/enum tag, a Cogent type definition is generated for the mapped name. The defined Cogent type is determined from the C type referenced by the type name as described in Section 2.6. The only difference is that all C type names used directly or indirectly by the C type are resolved, if they are not already external references. This is done to avoid introducing type names which are never referenced from any other place in the generated Cogent program.
- If the external reference is a function name, an exit wrapper and the corresponding Cogent abstract function definition is generated.
- If the external reference is the name of a global variable, no information is generated for Cogent, since Gencot does not support accessing global C variables from Cogent.
- If the external reference is the name of an enum constant or a preprocessor defined constant, a Cogent constant definition is generated.
- An external reference may be the name of a member in a struct or union. In this case also the struct or union tag must be externally referenced and the corresponding Cogent type definition is generated, as described above. Note that for a union member this will always be an abstract type which does not provide access to the member in Cogent.

2.1.3 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.

External Name References

To detect the external references, Gencot needs the information about all non-local references in every translated source file in the `<package>`. The non-local references are written to a separate file in a special format. For source file `x.c` it is written to file `x.c.refs`, for source file `x.h` it is written to file `x.h.refs`.

After processing all source file to be part of the Cogent compilation unit, Gencot processes all corresponding `.refs` files to detect the external references (where the origin file is not one of the processed `x.h`), and generate the required information. All generated type and constant definitions are put in the file `<package>-exttypes.cogent`.

Additionally, for all origin files used by at least one external reference, an include directive is put in the file `<package>-extincludes.c`.

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 C functions from Cogent are only created for the actual external references in a processing step for the whole `<package>`. They are implemented in antiquoted Cogent and put in the file `<package>-exit.ac`.

Abstract Functions as Interface to C Functions

If a Cogent function in `x-impl.cogent` invokes a function which is externally referenced and not defined in another file `y-impl.cogent`, this function must be declared as an abstract function in Cogent. These abstract function declarations are only created for the actual external references in a processing step for the whole `<package>`. They are put in the file `<package>-exit.cogent`.

The implementation for these functions is provided by the exit wrappers in `<package>-exit.ac`.

Abstract Types as Interface to C Types

A C type can be used in Cogent in two possible ways. Either 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. Since Gencot always maps a C type name to a different Cogent type name, both C type names do not conflict.

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 Cogent type name is different from the C type name, or has been generated if the C type has no name, Gencot generates 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.

Abstract type definitions referencing an existing C type may be generated in the files `x-types.cogent`, `x-impl.cogent`, and `<package>-exttypes.cogent`.

For file `x-types.cogent` we put the corresponding type mapping definitions in file `x-abstypes.h`. To make the information required for the referenced original C type definitions available in the Cogent compilation unit the file `x.h` must be included there as well. Note that this is possible without conflicts, since the type names generated by the Cogent compiler for non-abstract types are always mapped and thus different from all types in `x.h` or included files.

For file `x-impl.cogent` we put the corresponding type mapping definitions in the file `x-abstypes.c`. However, to make the information required for the referenced original C type definition available, it is not possible to include `x.c`, since the C function definitions would conflict with their entry wrappers in the Cogent compilation unit. Instead, the file `x-globals.c` is used, which is described in the next section.

For file `<package>-exttypes.cogent` we put the corresponding type mapping definitions in the file `<package>-exttypes.c`. The information required

for the referenced original C type definitions is always available, since all origin files for external references are included in the Cogent compilation unit.

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. Note, that this approach also makes all type definitions available which are needed by `x-abstypes.c`.

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 must be 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 `<package>-exit.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-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-abstypes.h C definitions for abstract Cogent types defined in **x-types.cogent** used to reference existing C types.

x-abstypes.c C definitions for abstract Cogent types defined in **x-impl.cogent** used to reference existing C types.

x-globals.c Content of **x.c** with all function definitions removed.

x.h.refs List of all non-local references used in **x.h**.

x.c.refs List of all non-local references used in **x.c**.

For the Cogent compilation unit the following common files are used:

<package>-exttypes.cogent Type and constant definitions for all external type and constant references.

<package>-exit.cogent Abstract function definitions for all external function references.

<package>-exit.ac Exit wrapper definitions for all external function references.

<package>-exttypes.c C type definitions for abstract types defined in **<package>-exttypes.cogent**.

<package>-extincludes.c Include directives for the origin files of all external references.

Main Files

To put everything together we use the files `<package>.cogent` and `<package>.ac`. The former includes all existing `x-impl.cogent` files and the files `<package>-exttypes.cogent` and `<package>-exit.cogent`. 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`, and the files `<package>-exit.ac` and `<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-abstypes.h` and `x-abstypes.c` and the files `<package>-exttypes.c` and `<package>-extincludes.c` 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 and all type definitions 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.

If a Cogent block comment end sequence `-}` occurs in a C block comment, the translated Cogent block comment will end prematurely. This will normally cause syntax errors in Cogent and must be handled manually. It is not detected by Gencot.

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.

If there is no main source code before the first subpart (e.g., a declaration starting with a struct definition), the before-group of the first line is nevertheless associated with the main part and not with the first subpart. The after-group at the end of a part is treated in the analogous way.

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 at most 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. If no target code is generated for the main part but for subparts, the before-unit of the main part immediately precedes the before-unit of the first subpart, if both exist, and analogously for the after-units.

Target code for a part may be generated in several separated places. If no code is generated for the main part, it must be defined to which group of subpart targets the comments associated with the main part is associated.

2.2.4 Declaration Comments

Since toplevel 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 or global variable is associated with its declaration instead of the definition.

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

Gencot assumes, that only one declaration exists for each 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 names defined in this way may have arbitrary C constants as their value. Gencot only handles integer and character constants, floating constant are not supported since they are not supported by Cogent.

2.3.1 Processing Direct Constant Definitions

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: U8
const1 = CONST1
```

preserving the original constant definition or directly in the shorter form

```
const1: U8
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 preprocessor 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 `0xFFFFFDD6` which is `4294967254` in decimal notation.

As described in Section 2.1.1, names for preprocessor defined constants are always mapped to a different name for the use in Cogent. This is not strictly necessary, if a preprocessor name is lowercase. By convention, C preprocessor constant definitions use uppercase identifiers, thus they normally must be mapped anyways.

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

2.3.2 Processing Indirect Constant Definitions

A constant definition may also reference a previously defined constant in the form

```
#define CONST2 CONST1
```

In this case the Cogent constant definition uses the same type as that for `CONST1` and also references the defined Cogent constant and has the form

```
#define CONST2 CONST1
const2: U8
const2 = const1
```

2.3.3 External Constant References

If the constant `CONST1` is an external reference in the sense of Section 2.1.2, a corresponding Cogent constant definition is generated in the file `<package>-exttypes.cogent`. It has the same form

```
#define CONST1 123
const1: U8
const1 = CONST1
```

as for a non-external reference. Thus we define the original preprocessor constant name `CONST1` here, although it is already defined in the external origin file. The reason for this approach is that the define directive here is intended to be processed by the Cogent preprocessor. Therefore we cannot include the origin file to make the name available, since that would also include the C code in the origin file.

If the external definition is indirect, the value used in the define directive is always resolved to the final literal or to an existing external reference. This is

done for determining the Cogent type for the constant and avoids introducing unnecessary intermediate constant names.

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:

- Preprocessor constant definitions
- 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 resolve all macro definitions as long as they are defined by another single macro name. If the result is a C integer constant (possibly negative) or a C character constant the macro is assumed to be a constant definition. 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 External References

Preprocessor constant names and other macro names can be referenced in normal C code where they are substituted by the corresponding definition. Additionally, defined preprocessor names can be referenced in the conditions of conditional directives. In both cases the reference may be external in the sense of Section 2.1.2.

For such external references the definition must be made available in the Cogent compilation unit. Since the references shall be processed by the Cogent preprocessor this cannot be done by including the origin files using Cogent include statements, since the origin files normally also contain C code. Instead, Gencot generates define directives for all such references in the file `<package>-exttypes.cogent`.

To avoid introducing additional external references, in the macro substitution text all macro names are resolved to existing external reference names or until they are fully resolved.

2.4.3 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.

2.4.4 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 file `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.

2.4.5 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 always external to the translated `<package>`, therefore system include directives are discarded in the Cogent code. The information required by external references from system includes is always fully contained in the file `<package>-exttypes.cogent`.

Translating Quoted Include Directives

Quoted include directives for a file `x.h` which belongs to the Cogent compilation unit 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 file `y-impl.cogent`. If the original include directive occurs in file `y.h` the translated statement is put into the file `y-types.h`.

Otherwise the quoted include includes an external file and is discarded in the Cogent source file for the same reason why the system includes are discarded.

2.4.6 Other Directives

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

2.5 Parsing and Processing C Code

After comments and preprocessor directives have been removed from a C source file, it is parsed and the C language constructs are processed to yield Cogent language constructs.

2.5.1 Including Files for C Code 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 comments and preprocessor directives have been 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.

In the included original files the comments are still present and must be removed as well. This could be done by the language-c preprocessor immediately before parsing. However, it is easier to remove the preprocessor directives when the comments are not present anymore. Therefore, Gencot removes the comments immediately after processing the quoted include directives.

2.5.2 Processing the C Code

2.6 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 both types are binary compatible as a whole.

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 binary compatible 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 mapped to Cogent names as described in Section 2.1.1.

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 the distinction between function types and function pointer types 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 using a tuple or record for the function parameters 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 is 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 a Cogent function type 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
```

2.7 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. A typedef can only occur on toplevel or in function bodies in C. For every declarator in a toplevel typedef Gencot generates a Cogent type definition at the corresponding position. Hence all these Cogent type definitions are on toplevel, as required in Cogent. Typedefs in function bodies are not processed by Gencot, as described in Section 2.8.

A C object declaration may occur

- on toplevel (called an “external declaration” in C),
- in a struct or union specifier for declaring members,
- in a parameter list of a function type for declaring a parameter,
- in a compound statement for declaring local variables.

External declarations are simply discarded by Gencot. In Cogent there is no corresponding concept, it is not needed since the scope of a toplevel Cogent definition is always the whole program.

Compound statements in C only occur in the body of a function definition, which is not translated by Gencot (see Section 2.8). Thus, declarations embedded in a body are not processed by Gencot.

Union specifiers are always translated to abstract types by Gencot, hence declarations for union members are never processed by Gencot.

The remaining cases are struct member declarations and function parameter declarations. For every declarator in an object declaration, Gencot generates a Cogent record field definition, if the C declaration declares struct members, or it generates a tuple field definition, if the C declaration declares a function parameter.

2.7.1 Target Code for struct/union/enum Specifiers

Additionally, whenever a struct-or-union-specifier or enum-specifier occurring in the C declaration has a body and a tag, a Cogent type definition is generated for the corresponding type, since it may be referred in C by its tag from other places. 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.

Since Cogent type definitions must be on toplevel, Gencot defers it to the next possible toplevel position after the target code generated from the context of the struct/union/enum declaration. If the context is a typedef, it is placed immediately after the corresponding Cogent type definition. If the typedef contains several full specifiers (which may be nested), all corresponding Cogent type definitions are positioned on toplevel in the order of the beginnings of the full specifiers in C (which corresponds to a depth-first traversal of all full specifiers).

If the context is a member declaration in a struct-or-union-specifier, the Cogent type definition is placed after that generated for its context.

If the context is a parameter declaration it may either be embedded in a function definition or in a declarator of another declaration. Function definitions in C always occur on toplevel, the Cogent type definitions for all struct/union/enum declarations in the parameter list are placed after the target code for the function definition (which may be unusual for manually written Cogent code, but it is easier to generate for Gencot). In all other cases the Cogent type definitions for struct/union/enum declarations in a parameter list are treated in the same way as if they directly occur in the surrounding declaration.

Note, that a struct/union/enum tag declared in a parameter list has only “prototype scope” or “block scope” which ends after the function type or definition. Gencot nevertheless generates a toplevel type definition for it, since the tag may be used several times in the parameter list or in the corresponding body of a function definition. Note that this may introduce name conflicts, if the same tag is declared in different parameter lists. Since declaring tags in a parameter list is very unusual in C, Gencot does not try to solve these conflicts, they will be detected by the Cogent compiler and must be handled manually.

A full specifier without a tag can only be used at the place where it statically occurs in the C code. Gencot does not generate a toplevel type definition for it, it directly embeds the generated type in the target code generated for the specifier context. If the context is a declaration for struct members, the generated type is used as the type for the corresponding Cogent record field. If the context is a function parameter declaration, the generated type is used as the type for the corresponding Cogent tuple field. Note, that in both cases the tagless specifier can have nested full specifiers with tags, which are treated in the normal way.

2.7.2 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 comma or semicolon, thus there is no main part code between or after the declarators. The specifiers may consist of a single full specifier, then there is no main part code at all.

The target code part generated for a declaration consists of the sequence of target code parts generated for the declarators, and of the sequence of target code parts generated for the full specifier, if present. No target code is generated for the main part itself. In both sequences the subparts are positioned consecutively, but the two sequences may be separated by other code, since the second sequence consists of Cogent type definitions which must always be on toplevel.

According to the rules defined in Section 2.2.3, the before-unit of the declaration is put before the target of the first subpart, which is that for the full specifier, if present, otherwise it is the target for the first declarator. In the first case the comments will be moved to the type definition for the full specifier. The rationale is that often a comment describing the struct/union/enum declaration is put before the declaration which contains it.

The after-unit of the declaration is always put behind the target of the last declaration.

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. The parentheses around the parameter-type-list belong to the main part, thus a comment is only associated with a parameter if it occurs inside the parentheses.

In all other cases a declarator is an unstructured source code part.

2.7.3 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 2.6, either a Cogent type expression is determined from this C type, or the Cogent type is decided to be abstract.

The defined type name is generated from the C type name according to the mapping described in Section 2.1.1. Type names used in the C type specification are mapped to Cogent type names in the Cogent type expression in the same way.

2.7.4 Object Declarations

C object declarations are processed if they declare struct members or function parameters.

For such a C object 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 2.6.

In the case of a named record field the Cogent field name is determined from the name in the C declarator as described in Section 2.1.1. In the case of an unnamed tuple field a name specified in the C parameter declaration is always discarded.

2.7.5 Struct or Union Specifiers

For a full specifier with a tag Gencot generates a Cogent type definition. The name of the defined type is generated from the tag as described in Section 2.1.1. For a union specifier the type is abstract, no defining type expression is generated. For a struct specifier a (boxed) Cogent record type expression is generated, which has a field for every declared struct member which is not a bitfield. Bitfield members are aggregated as described in Section 2.6.

A specifier without a body must always have a tag and is used in C to reference the full specifier with the same tag. Gencot translates it to the Cogent type name defined in the type definition for the full specifier.

Note that the Cogent type defined for the full specifier corresponds to the C type of a pointer to the struct or union, whereas the unboxed Cogent type corresponds to the C struct or union itself. This is adapted by Gencot when translating the C specifier embedded in a context to the corresponding Cogent type reference.

2.7.6 Enum Specifiers

For a full enum specifier with a tag Gencot generates a Cogent type definition immediately followed by Cogent object definitions for all enum constants. The name of the defined type is generated from the tag as described in Section 2.1.1. The defining Cogent type is always U32, as described in Section 2.6.

A specifier without a body must always have a tag and is used in C to reference the full specifier with the same tag. Gencot translates it to the Cogent type name defined in the type definition for the full specifier.

2.8 Processing C Function Definitions

A C function definition is translated by Gencot to a Cogent function definition. Old-style C function definitions where the parameter types are specified by separate declarations between the parameter list and the function body are not supported by Gencot because of the additional complexity of comment association.

The Cogent function name is generated from the C function name as described in Section 2.1.1.

The Cogent function type is generated from the C function result type and from all C parameter types as described in Section 2.6. In a C function definition the types for all parameters must be specified in the parameter list, if old-style function definitions are ignored.

2.8.1 Function Bodies

In C the function body consists of a compound statement which is specified in imperative programming style. In Cogent the function body consists of an expression which is specified in functional programming style with additional considerations which are crucial for proving properties of the Cogent program. Therefore Gencot does not try to translate function bodies, this must be done by a human programmer.

It would be possible, however, to translate C declarations embedded in the body. These may be type definitions and definitions for local variables. However, there are no good choices for the generated target code. Type definitions cannot be local in an expression in Cogent, they must be moved to the toplevel where they may cause conflicts. Local variable definitions could be translated to Cogent variable bindings in let-expressions, however, C assignments cannot be translated for them. Also, the resulting mixture of C code and Cogent code is expected to be quite confusing to the programmer who has to do the manual translation. Therefore, no declarations in function bodies are processed by Gencot.

The only processing done for function bodies is the substitution of names occurring free in the body. These may be names with global scope (for types, functions, tags, global variables, enum constants or preprocessor constants) or parameter names. For all names with global scope Gencot has generated a Cogent definition using a mapped name. These names are substituted in the C code of the function body by the corresponding mapped names so that the mapping need not be done manually by the programmer.

Additionally, the function parameter names usually occur free in the function body. To make them apparent to the programmer, Gencot generates a Cogent pattern for the (single) parameter of the Cogent function which consists of a tuple of variables with the names generated from the C parameter names. As described in Section 2.1.1 the C parameter names are only mapped if they are uppercase, otherwise they are translated to Cogent unmodified. If they are mapped they are substituted in the body. Since it is very unusual to use uppercase parameter names in C, the Cogent function will normally use the original C parameter names.

The generated Cogent function definition has the form

```
<name> :: (<ptype1>, ..., <ptypen>) -> <restype>
<name> (<pname1>, ..., <pnamen>) =
<compound statement>
```

where the `<compound statement>` is plain C code with substituted names.

Of course this code cannot be syntax checked by the Cogent compiler. It would be possible to include the C function body in a comment to improve that. However, then still a dummy result expression would be needed for the Cogent function to be valid Cogent code. It is not trivial to generate that for arbitrary result types, hence Gencot does not try to separate the C code from the Cogent code other than putting it in a new line after the Cogent function header.

2.8.2 Comments in Function Definitions

A C function definition which is not old-style syntactically consists of a declaration with a single declarator and the compound statement for the body. It is treated by Gencot as a structured source code part with the declaration and the body as subparts without any main part code. According to the structures of declarations the declaration has the single declarator as subpart and optionally a full specifier, if present. The declarator has the parameter declarations as subparts.

The target code part for the declaration and for its single declarator is the header of the Cogent function definition (first two lines in the schema in the previous section). The target code part for the full specifiers with tags in the declaration (which may be present for the result type and for each parameter) is a sequence of corresponding type definitions, as described for declarations in Section 2.7.1, which is placed after the Cogent function definition. The target code part for full specifiers without tags is the generated type expression embedded in the Cogent type for the corresponding parameter or the result.

All parameter declarations consist of a single declarator and the optional full specifier. The target code part for a parameter declaration and its declarator is the corresponding parameter type in the Cogent function type expression.

Hence, comments associated with parameter declarations in C are moved to the parameter type expression in Cogent.

To preserve comments embedded in the C function body it is also considered as a structured source code part. Its subparts are the declarations (structured as usual) and the statements. A statement is structured if it contains declarations or statements, these are its subparts. Since in the target code only identifier are substituted, the target code structure is the same as that of the source code. The structure is only used for identifying and re-inserting the transferrable comments and preprocessor directives. Note that this works only if the conditional directive structure is compatible with the statement structure, i.e., a group must always contain a sequence of complete statements and/or declarations, which is the usual case in C code.

An alternative approach would be to treat all nonempty source code lines as subparts of a function body, resulting in a flat structure of single lines. The advantage is that it is always compatible to the conditional directive structure and all comment units would be transferred. However, generating the corresponding origin markers in an abstract syntax tree is much more complex than generating them for syntactical units for which the origin information is present in the syntax tree. Since the Gencot implementation generates the target code as an abstract syntax tree, the syntactical statement structure is preferred.

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.

3.1 Origin Positions

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.

For re-inserting it must be possible to relate comments and preprocessor directives to the generated target code parts. As described in Sections 2.2 and 2.4, comments and preprocessor directives are associated to the C source code via line numbers. Whenever a target code part is generated, it is annotated with the line numbers of its corresponding source code part. Based on these line number annotations the comments and preprocessor directives can be positioned at the correct places.

The line number annotations are markers of one of the following forms, each in a single separate line:

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

where `<bline>` and `<aline>` are line numbers.

An `#ORIGIN` marker specifies that the next code line starts a target code part which was generated from a source code part starting in line `<bline>`. An `#ENDORIG` marker specifies that the previous code line ends a target code part which was generated from a source code part ending in line `<aline>`. Thus, by surrounding a target code part with an `#ORIGIN` and `#ENDORIG` marker the position and extension of the corresponding source code part can be derived.

In the case of a structured source code parts the origin marker pairs are nested, if the target code part generated from a subpart is nested in the target code part generated from the main part. If there is no code generated for the main part, the `#ORIGIN` marker for the first subpart immediately follows the `#ORIGIN` marker for the main part and the `#ENDORIG` marker for the last subpart is immediately followed by the `#ENDORIG` marker for the main part.

If no target code is generated from a source code part, the origin markers are not present. This implies, that an `#ORIGIN` marker is never immediately followed by an `#ENDORIG` marker.

It may be the case that several source code parts follow each other on the same line, but the corresponding target code parts are positioned on different lines. Or from a single source code part several target code parts on different lines are generated. In both cases there are several origin markers with the same line number. Conditional preprocessor directives associated with that line must be duplicated to all these target code parts. For comments, however, duplication is not adequate, they should only be associated to one of the target code parts. This is implemented by appending an additional “+” sign to an origin marker, as in

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

Comments are only associated with markers where the “+” sign is present, all other markers are ignored. In this way, the target code generation can decide where to associate comments, if a position is not unique.

3.2 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 declarations to definitions.

The filter `gencot-selcomments` selects all comments from the input, translates them to Cogent comments 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.

3.2.1 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` statements 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.

3.2.2 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. It translates C comments to Cogent comments in the output. The translation is done here since the filter must identify the start and end sequences of comments, so it can translate them specifically. Start and end sequences which occur as part of comment content are not translated.

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.

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.

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 translated 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.

The filter never deletes lines, hence in its output the original line numbers can still be determined by counting lines, if the newlines belonging to the separators are ignored.

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, replacing C comment start and end sequences by their Cogent counterpart. 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); new-
    line; 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

3.2.3 Filter gencot-mrgcomments

The filter for merging comments into the target code is implemented as an awk script. It consists of a BEGIN rule, a line rule, and an END 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 (original) line number of the separator between before- and after-unit.

The line rule uses a buffer for its output. It is used to process all **#ORIGIN** and **#ENDORIG** markers belonging to a line and collect the associated comment units and content. The END rule is used to flush the buffer.

For every consecutive sequence of **#ORIGIN** markers, the before units associated with the line numbers of all markers with a “+” sign are collected in a buffer. Every single code line is put in a second buffer. For every consecutive sequence of **#ENDORIG** markers, the after units associated with the line numbers of all markers with a “+” sign are collected in a third buffer. Whenever a code line or an **#ENDORIG** marker is followed by a line which is no **#ENDORIG** marker, the content of all three buffers is output and the buffers are reset.

In the buffer, before-units are concatenated without any separator. After-units are separated by a newline to end possibly trailing line comments.

3.2.4 Declaration Comments

To safely detect C declarations and C definitions Gencot uses the language-c parser.

Only comments associated with declarations with external linkage are transferred to their definitions. For declarations with internal linkage the approach for transferring the comments does not work, since the declared names need not be unique in the <package>.

Processing the declaration comments is implemented by the following filter steps.

Filter gencot-deccomments

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

#DECL <name> <bline> <aline>

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

Filter `gencot-movcomments`

The filter `gencot-movcomments` `<file>` processes the output of `gencot-deccomments` as input. For every 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-<name>.comment` and `after-<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 `#DECL`. 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 definition, Gencot uses the markers

```
#DEF before-<name>
#DEF after-<name>
```

The markers must be inserted by the filter which generates the 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 `#DEF` it is replaced by the content of the corresponding file in the output. All other lines are copied to the output without modification.

3.3 Preprocessor Directives

3.3.1 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-rempp` `<file>` removes all preprocessor directives from its input, replacing them by empty lines. All other lines are copied to the output without modification. 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, for each kind Gencot provides the merging filter `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.

3.3.2 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-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 intended 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 in an initial step, 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.

3.3.3 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.

3.3.4 Filter `gencot-prcconst`

3.3.5 Merging Directive Processing Results

3.3.6 Filter `gencot-mrgcond`

3.4 Parsing and Processing C Code

3.4.1 Including Files

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 included files.

Execution Related to Other Filters

The include processing must be applied as the first step on the original C source files when doing C code processing. Only then the following steps are applied to all code which is finally fed to the language-c parser. The include filter step is followed by application of `gencot-remcomments`, then `gencot-rempp`.

The filters for selecting comments and preprocessor directives, however, are still applied to the single original source files, since they do not require additional information from the included files.

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 `fff` 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 multiple times. Usually, C include files use an `ifdef` directive around all content to prevent multiple includes. The `gencot-include` filter does not interpret `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 `dir/` 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 `fff` 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.

3.4.2 Preprocessing

The language-c parser supports an integrated invocation of an external preprocessor, the default is to use the `gcc` preprocessor. However, the integrated invocation always reads the C code from a file (and checks its file name extension) and not from standard input.

To implement C code processing as a filter, Gencot does not use the integrated preprocessor integration, it invokes the preprocessor as an additional separate step. For consistency reasons it is wrapped in the minimal filter script `gencot-cpp`.

The preprocessor step only has the following purpose:

- process all system include directives by including the file contents,
- process retained conditional directives to prevent conflicts in the C code.

All other preprocessing has already been done by previous steps.

3.4.3 Reading the Input

Parsing

To apply the parser to the standard input we invoke it using function `parseC`. It needs an `InputStream` and an initial `Position` as arguments.

The language-c parser defines `InputStream` to be the standard type `Data.ByteString`. To get the standard input as a `ByteString` the function `ByteString.getContents` can be used.

The language-c parser uses type `Position` to describe a character position in a named file. It provides the function `initPos` to create an initial position at the beginning of a file, taking a `FilePath` as argument, which is a `String` containing the file name. Since Gencot and the C preprocessor create line directives with the file name `<stdin>` for the standard input, this string is the correct argument for `initPos`.

The result of `parseC` is of type `(Either ParseError CTranslUnit)`. Hence it should be checked whether an error occurred during parsing. If not, the value of type `CTranslUnit` is the abstract syntax tree for the parsed C code.

Both `parseC` and `initPos` are exported by module `Language.C`. The function `ByteString.getContents` is exported by the module `Data.ByteString`. Hence to use the parser we need the following imports:

```
import Data.ByteString (getContents)
import Language.C (parseC,initPos)
```

Then the abstract syntax tree can be bound to variable `ast` with

```
do
  input_stream <- Data.ByteString.getContents
  ast <- either (error . show) return $ parseC input_stream (initPos "<stdin>")
```

Analysis

The language-c parser supports an additional “analysis” module for further processing the abstract syntax tree. Although it is not complete and only processes toplevel declarations (including typedefs), and object definitions, it is very useful for implementing Gencot translation. Function definition bodies are not covered by analysis, but they are not covered by Gencot either.

The result of the analysis module is a map for all toplevel declarations and object definition, mapping the identifier to its semantics, which is mainly its declared type. Whereas in the abstract syntax tree there may be several declarators in a declaration, declaring identifiers with different types derived from a common type, the map maps every identifier to its fully derived type.

Also, tags for structs, unions and enums are contained in the map. In C their definitions can be embedded in other declarations. The analysis module collects all these possibly embedded declarations in the map. The map also gives for every defined type name the definition.

Together, the information in the map is much more appropriate for creating Cogent code, where all type definitions are on toplevel. Therefore, Gencot uses the map resulting from the analysis step as starting point for its translation.

To use the analysis module, the following import is needed:

```
import Language.C.Analysis
```

Then, if the abstract syntax tree has been bound to variable `ast`, it can be analysed by

```
globals <- either (error . show) (return . fst) $ runTrav_ $ analyseAST ast
```

which binds the resulting map to variable `globals`. `runTrav_` returns a result of type `Either [CError] (GlobalDecls, [CError])`, where `GlobalDecls` is the semantics map. The error list in the first alternative contains fatal errors which made the analysis fail. The error list in the second alternative contains warnings about semantic inconsistencies, such as unknown identifiers, which are returned together with the map.

Source Code Origin

3.4.4 Generating Cogent Code

Cogent Surface Syntax Tree

Extending the Cogent Surface Syntax