HoBit Report on Gencot Development

Gunnar Teege

October 10, 2019

Chapter 1

Introduction

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

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

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

Roughly, Gencot supports the following tasks:

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

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

Chapter 2

Design

2.1 General Context

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

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

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

2.1.1 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. Sometimes 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 lowecase, 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). Moreover, Cogent field names must not begin with an underscore, so these names are mapped as well, by prepending `cogent_` (which results in two consecutive underscores).

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, 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 Type Names

There are cases where in Cogent a type name must be introduced for an unnamed C type (directly specified by a C type expression). Then the Cogent type name 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. Basically, an unnamed C type could be mapped to a corresponding Cogent type expression. However, this is not always possible or feasible.

Tagless enum types are always mapped to a primitive type in Cogent.

A tagless C struct could be mapped to a corresponding Cogent record type expression. However, the tagless struct can be used in several declarators and several different types can be derived from it. In this case the Cogent record expression would occur syntactically in several places, which is semantically correct, but may not be feasible for large C structs. Therefore, Gencot introduces a Cogent type name for every tagless C struct.

In the remaining cases no corresponding binary compatible type can be defined for a C type in Cogent (see Section 2.6). In these cases an abstract type is defined in Cogent which references the C type. The abstract type in Cogent is specified by its name only, hence a Cogent type name must be introduced.

To be able to process every source file independently from all other source files, Gencot uses a schema which generates a unique name for every C type expression.

Tagless structs and unions syntactically occur at only a single place in the source. The unique name is derived from that place, using the name of the corresponding source file and the line number where the struct/union begins in that file (this is the line where the struct or enum keyword occurs). The generated names have the forms

```
<kind><lnr>_x_h
<kind><lnr>_x_c
```

where the suffix is constructed from the name `x.h` or `x.c` of the source file. `<kind>` is one of `Struct` or `Union`, and `<lnr>` is the line number in the source file.

Derived types may syntactically occur at many places in a C program. The base type of a derived type is always either another derived type, a typedef

name, or a struct/union/enum type. Gencot maps all struct/union/enum types to a Cogent type name. Hence every derived type can be uniquely characterized by a sequence of derivation steps starting with a type name. The sequence of derivation steps is syntactically encoded in the generated name as follows.

A pointer derivation step is encoded by a single letter `"P"`.

An array derivation step without size specification is encoded by a single letter `"A"`. An array derivation step with a literal as size specification is encoded in the form

```
A<size>
```

where `<size>` is the size specification. If the size is specified by a single identifier the step is encoded in the form

```
AX<size>X
```

where `X` is a letter not occurring in the identifier. In all other cases an array derivation step is encoded by

```
AII
```

which may lead to name conflicts in Cogent and must be handled manually. As explained in Section 2.6.4 two different Cogent type names are used for every C array type. The second form is build by encoding the array derivation step with an additional letter `"U"` prepended.

A function derivation step is encoded in the form

```
F_X<P1>X<P2>X...X<Pn>X_
```

where the `<Pi>` are the parameter types and `X` is a letter not occurring in any of the parameter type names.

Function parameters of linear type my be readonly or not (see Section 2.6.8). For a parameter of linear type, the parameter type `<Pi>` is preceded by a letter `"R"` if it is readonly, otherwise it is preceded by a letter `"L"`. If the parameter type is not derived, the letter is separated by an underscore. Parameters of nonlinear type are not marked in either of this ways.

If parameters are not specified a step of the form `F_UnknownCogentParameters_` is used. If the function is variadic, the step ends with `XR_VariadicCogentParametersX_`.Both forms may lead to name conflicts in Cogent and must be handled manually.

If several derivation steps are applied to a base type, their encodings are concatenated, beginning with the last derivation step. The identifier for the base type is appended to the encoding for the first derivation step, separated by an underscore, if the first step is not a function derivation step. For a derived pointer or function type the base type can be the pseudo type `void`. In these cases the identifier `Void` is used for the base type in the Cogent type name.

Hence, for example for the C type

```
int (* [5])(int, const short*)
```

the generated name is

```
A5PF_XU32XRP_U16X_U32
```

Note that the generated Cogent 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 or a possible encoding of a derivation step.

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 produces again 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, no common include files are used.

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.

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. However, to inform the developer about the variable and its type, Gencot translates it to an abstract function from unit type to the type of the variable, without providing an implementation. It can be used to manually implement read access to the variable, write access must be handled completely manually. The name of the abstract function is the mapped name of the variable.

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.

It would be possible to translate only some include files used by the Cogent compilation unit to Cogent and access the content of the others through abstract Cogent types and functions which are mapped in antiquoted C to the original C definitions. However, for checking the refinement proof of the Cogent compilation unit, the Isabelle C parser reads the complete C program which results

from translating the Cogent code to C. The Isabelle C parser only supports a C subset and normally the original <package> include files will not be restricted to this subset. Therefore we do not use any of them for the Cogent compilation unit. Gencot always assumes that *all* include files used (transitively) by the Cogent compilation unit are also translated to Cogent and included in this form.

The only exception are the system include files. Their content typically needs specific treatment anyways. Gencot replaces every reference to a function declared in a system include file by an abstract function, whithout providing an implementation. It must be added manually by the developer. Referenced types defined in system include files are translated as usual.

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. As described above, all used include files which belong to the <package> must have been translated. If the origin file of a non-local reference 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.

A non-local reference is external relative to a given set of C source files, if its definition does not belong to the content of the set. For a name without linkage (mainly type names and struct/union/enum tags) its definition must be present for every reference, i.e. contained in the included origin file. Thus a reference to such a name is external, if its origin file does not belong to the set. For a name with linkage (mainly names of functions and variables) it depends on the kind of linkage. If it has internal linkage, its definition must also be present for every reference. Then the origin file is that containing the (single) definition, not a file containing a declaration.

If it has external linkage, however, the definition need not be present. In this case the origin file only contains a declaration of the name and even needs not be unique. Then it is not possible to decide whether a non-local reference is external by simply looking at the content of all included files. Instead, all the files in the given set must be inspected, whether they contain the name's definition. Note that this is only necessary for deciding whether a reference is external. The information necessary for processing it is always present as part of the declaration in the included origin file.

On the C level the information for external name references with internal or no linkage must always be provided manually, since they are not defined in the <package>. The information for variable references with external linkage must also be provided manually. The information for function references with external linkage is generated automatically by Gencot in the form of the exit

wrapper functions defined in antiquoted C.

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, the abstract access function is generated in the same form as when translating a variable defined in the Cogent compilation unit.

- 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. However, the Cogent preprocessor is executed separately for every file included with this `include` statement, thus preprocessor macros defined in an included file are not available in all other files. For this reason it cannot be used to reflect the file structure of a C program.

The second kind is the Cogent preprocessor `#include` directive, it works like the C preprocessor `#include` directive and is used by Gencot to integrate the separate Cogent source files. 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.cogent` containing definitions of the same functions. We map every C include file `x.h` to a Cogent source file `x-incl.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 `.cogent` and `-incl.cogent` using the Cogent preprocessor `#include` directive.

The file `x-incl.cogent` 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.

In some cases an `x.h` file contains function definitions, typically for inlined functions. They are translated to Cogent function defintions in the `x-incl.cogent` file in the usual way.

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

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>-externs.ac`.

External Name References

For external name references Gencot generates the information required for Cogent. All generated type and constant definitions are put in the file `<package>-exttypes.cogent`.

If a Cogent function in `x.cogent` invokes a function which is externally referenced and not defined in another file `y.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>-externs.cogent`. The corresponding exit wrappers are put in file `<package>-externs.ac` as described above.

For external variables Gencot creates abstract access functions and also puts them in `<package>-externs.cogent`. These functions are mostly intended as information for the developer, no implementation is provided by Gencot.

Derived Types

Gencot generates type definitions for all derived types used in the C source. For function and array types Gencot also generates abstract function definitions for working with values of these types, as described in Sections 2.6.5 and 2.6.4. All these definitions are put in the file `<package>-dvdtypes.cogent`.

Function types are used in C for defining or declaring functions or for constructing a function pointer type. In the former case the type is always translated to a Cogent function type expression, as described in Section 2.6.5. Only in the latter case it is translated to an abstract type with additional abstract functions.

For function pointer types Gencot also generates C type definitions, mapping the generated type name to a pointer to an incompletely defined function type returning void. The resulting type is always binary compatible with the original derived type.

For derived array types to be binary compatible, the element type must have the correct size and the element number must be correct. In simple cases Gencot can determine a corresponding C type using the name of the Cogent element type (which is used for the C type after translation by the Cogent compiler). In other cases a corresponding C type must be provided manually by the developer.

Gencot puts the corresponding C type definitions in the file `<package>-dvdtypes.h`.

For the abstract functions for function pointer and for array types Gencot provides implementations in antiquoted C in the file `<package>-dvdtypes.ac`.

Dummy Value Functions

Gencot does not translate C function bodies, instead, it uses a Cogent expression for a dummy result value (see Section 2.9). For linear and abstract Cogent types these expressions are constructed with the help of abstract functions. All these abstract function definitions are put in the file `<package>-dummies.cogent`.

Since the dummy result expressions are intended to be eliminated when the C bodies are manually translated, Gencot does not generate corresponding C function definitions for them.

Putting all abstract dummy function definitions in a single file provides an easy way to remove them from the Cogent compilation unit when the manual translation has been completed and they are not needed any more.

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.

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. They are not removed from `x-globals.c`, because they could be used in initializers of object definitions with external linkage.

The files `x-globals.c` do not become part of the Cogent compilation unit, because they use the original include files from the package and should not be read by the Isabelle C parser. Instead, each of them constitutes an own C compilation unit which complements the Cogent compilation unit. It must be separately compiled and linked with the Cogent compilation unit to provide the global variables for the remaining C compilation units.

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-incl.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>-externs.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.cogent` Implementation of all functions defined in `x.c`. Abstract access functions for all global variables defined in `x.c`. For each file `y.h` included by `x.c` the file `y-incl.cogent` is included.

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

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>-externs.cogent` Abstract function definitions for all external function and variable references.

`<package>-dvdtypes.cogent` Type and abstract function definitions for all used derived types.

`<package>-dummies.cogent` Abstract function definitions for dummy result values.

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

`<package>-dvdtypes.h` C type definitions for abstract types defined in `<package>-dvdtypes.cogent`.

`<package>-dvdtypes.ac` Implementations of abstract functions defined in `<package>-dvdtypes.cogent`.

Main Files

To put everything together we use the files `<package>.cogent` and `<package>.ac`.
The former includes all existing `x.cogent` files and the files `<package>-exttypes.cogent`, `<package>-externs.cogent`, `<package>-dvdtypes.cogent` and `<package>-dummies.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>-externs.ac`, `<package>-dvdtypes.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 code of the Cogent compilation unit. The file `<package>-dvdtypes.h` is `$esc`-included in `<package>.ac`, thus the corresponding normal include directive for it is present in `<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 atmost the before-unit of the first line and the after-unit of the last line is transferred to the target part. If the source code part is structured the same property holds for every embedded subpart. 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-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 between the comments preceding the definition and 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 comment associated with one of them is moved to the definition, the comments associated with the other declarations are lost.

Only before-units are handled this way, due to a technical problem with the C parser used. For declarations it does not provide the end position in a safe manner. For the intended application this is not a problem since API documentations are usually placed before the declaration and not after it.

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, character, and string constants, floating constant are not supported since they are not supported by Cogent.

2.3.1 Processing Direct Integer 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-incl.cogent` the identifiers are available in every Cogent file including `x-incl.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 `0xFFFFFFD6` 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 Direct Character and String Constant Definitions

A character constant definition has the form

```
#define CONST1 'x'
```

It is translated to a Cogent value definition similar as for integer constants. As type always `U8` is used, the constant is transferred literally.

A string constant definition has the form

```
#define CONST1 "abc"
```

It is translated to a Cogent value definition similar as for integer constants. As type always `String` is used.

In C it is also possible to specify a string constant by a sequence of string literals, which will be concatenated. A corresponding string constant definition has the form

```
#define CONST1 "abc" "def"
```

Since there is no string concatenation operator in Cogent, the concatenation is performed by Gencot and a single string literal is used in the Cogent value definition.

2.3.3 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
```

2.3.4 Processing Expression Constant Definitions

A constant definition in C may also specify its value by an expression. In this case the C preprocessor will replace the constant upon every occurrence by the expression, every expression according to the C syntax is admissible.

In this case Gencot also generates a Cogent value definition and transfers the expression. Gencot does not evaluate or translate the expression, however, it maps all contained names of other preprocessor defined constants to their Cogent form, so that they refer the corresponding Cogent value name. As type for an expression Gencot always assumes `int`, i.e. `U32` in Cogent.

If the expression is of type `int` and only uses operators which also exist in Cogent, positive integer constants and preprocessor defined constant names, the resulting expression will be a valid Cogent expression. In all other cases the Cogent compiler will probably detect a syntax error, these cases must be handled manually.

2.3.5 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 macro 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 the code, and re-insert the conditional directives into the generated Cogent code.

Only directives which belong to the <package> are handled this way, i.e., only directives which occur in source files belonging to the <package>. For directives in other included files, in particular in the system include files, this would not be adequate. First, normally there is no generated target code where they could be re-inserted. Second, configurations normally do not apply to the system include files.

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.

Retaining Conditional directives for certain configurations in the processed code makes only sense if the corresponding macro definitions which are tested in the directives are retained as well. Therefore also define directives can be retained. The approach in Gencot is to specify a list of regular expressions in the format used by awk. All directives which match one of these regular expressions are retained in the code to be interpreted before processing the code. The list is called the "Gencot directive retainment list" and may be specified for every invocation of Gencot.

The retained directives affect only the C code, the preprocessor directives are already selected for separate processing. To suppress directives which belong to a configuration not to be translated, macro definitions must be explicitly suppressed with the help of the Gencot manual macro list (see below) and include directives must be suppressed with the help of the Gencot include omission list (see below). Conditional directives are automatically omitted if their content is omitted.

2.4.2 Conditional Directives

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

Associating Conditional Directives to Target Code

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

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

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

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

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

Since the target code is generated without presence of the conditional directives structure, they must be transferred afterwards. This is done using the

same markers `#ORIGIN` and `#ENDORIG` as for the comments. Since every conditional directive occupies a whole line, the contents of every group consists of a sequence of lines not overlapping with other groups on the same level. If every target code part is marked with the begin and end line of the corresponding source code part, the corresponding group can always be determined from the markers.

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

2.4.3 Macro Definitions

Preprocessor macros are defined in a definition, which specifies the macro name and the replacement text. Optionally, a macro may have parameters. After the definition a macro can be used any number of times in "macro calls". A macro call for a parameterless macro has the form of a single identifier (the macro name). A macro call for a macro with parameters has the form of a C function call: the macro name is followed by actual parameter values in parenthesis separated by commas. However, the actual parameter values need not be C expressions, they can be arbitrary text, thus the macro call need not be a syntactically correct C function call.

Macro calls can occur in C code or in other preprocessor directives (macro definitions, conditional directives, and include directives). All macro calls occuring in C code must result in valid C code after full expansion by the preprocessor.

General Approach

Gencot tries to preserve macros in the translated target code instead of expanding them. In general this requires to implement two processing aspects: translating the macro definitions and translating the macro calls. Since Gencot processes the C code separately from the preprocessor directives, macro call processing can be further distinguished according to macro calls in C code and in preprocessor directives.

Gencot processes the C code by parsing it with a C parser. This implies that macro calls in C code must correspond to valid C syntax, or they must be preprocessed to convert them to valid C syntax. Note that it is always possible to do so by fully expanding the macro definition, however, then the macro calls cannot be preserved.

There are several special cases for the general approach of macro processing:

- if calls for a macro never occur in C code they need not be converted to valid C syntax and they need only be processed in preprocessor directives. This is typically the case for "flags", i.e., macros with an empty replacement text which are used as boolean flags in conditional directives.

- if for a parameterless macro all calls in C code occur at positions where an identifier is expected, the calls need not be converted to valid C syntax and can be processed in the C code. This approach applies to the "preprocessor defined constants" as described in Section 2.3.

- if for a macro with parameters all calls in C code are syntactically valid C function calls and always occur at positions where a C function call is expected, the calls need not be converted to valid C syntax and can be processed in the C code.
- if a macro need not be preserved by Gencot, its calls can be converted to valid C code by fully expanding them. Then the calls are not present anymore and the calls and the definition need not be processed at all. This approach is used for conflicting configurations as described in Section 2.4.1.

When Gencot preserves a macro, there are several ways how to translate the macro definition and the macro calls. An apparent way is to use again a macro in the target code. Then the macro definition is translated by translating the replacement text and optionally also the macro name. If the macro name is translated then also all macro calls must be translated, otherwise macro calls need only be translated if the macro has parameters and the actual parameter values must be translated.

How the macro replacement text is translated depends on the places where the macro is used. If it is only used in preprocessor directives, usually no translation is required. If it is used in C code parts which are translated to Cogent code, the replacement text must also be translated to Cogent. If it is used in C function bodies it must be translated in the same way as C function bodies, i.e., only the identifiers must be mapped and calls of other macros must be processed.

A macro may also be translated to a target code construct. Then the macro definition is typically translated to a target code definition (such as a type definition or a function definition) and the macro calls are translated to usages of that definition. This approach is used for the "preprocessor defined constants" as described in Section 2.3: the macro definitions are translated to Cogent constant definitions and the macro calls occurring in C code are translated to the corresponding Cogent constant names. Additionally, the original macro definitions are retained and used for all macro calls occurring in conditional directives, which are not translated. Macro calls in the replacement text of other preprocessor constant definitions are translated to the corresponding Cogent constant names.

Flag Translation

A flag is a parameterless macro with an empty replacement text. Its only use is in the conditions of conditional preprocessor directives, hence macro calls for flags only occur in preprocessor directives.

Gencot translates flags by directly transferring them to the target code. Neither their macro definitions nor their macro calls are further processed by Gencot.

The translation of a flag can be suppressed with the help of the Gencot manual macro list (see below).

Manual Macro Translation

Most of the aspects of macro processing cannot be determined and handled automatically by Gencot. Therefore a general approach is supported by Gencot

where macro processing is specified manually for specific macros used in the translated C program package.

The manual specification consists of the following parts:

- A specification of all parameterless macros which shall not be processed as preprocessor defined constants or flags. This specification consists of a list of macro names, it is called the "Gencot manual macro list". For all macros in this list a manual translation must be specified. Macros with parameters are never processed automatically, for them a manual translation must always be specified if they shall be preserved.

- For all manually processed macros for which macro calls may occur in C code a conversion to valid C code may be specified. This specification is itself a macro definition for the same macro, where the replacement text must be valid C code for all positions where macro calls occur. A set of such macro definitions is called a "Gencot macro call conversion". It is applied to all macro calls and the result is fed to the Gencot C code translation and is processed in the usual way, no further manual specification for the macro call translation can be provided. Since the conversion is applied after all preprocessor directives have been removed, it has no effect on macro calls in preprocessor directives.

- For all manually translated macros a translation of the macro definition may be specified. It has the form of arbitrary text marked with a specification of the position of the original macro definition in its source file. According to this position it is inserted in the corresponding target code file. A collection of such macro definition translations is always specifi for a single source file and is called the "Gencot macro translation" for the source file.

All four parts may be specified as additional input upon every invocation of Gencot.

The usual application for suppressing a flag definition with the Gencot manual macro list ist a parameterless macro which is conditionally defined by a replacement text or as empty. The second definition then looks like a flag but other than for flags the macro calls typically occur in C code. Usually in this case also the macro calls should be suppressed, this can be done by adding an empty definition for the macro to the Gencot macro call conversion.

Macro definitions are always translated at the position where they occurred in the source file. If the definition occurs in a file `x.h` it is transferred to file `x-incl.cogent` to a corresponding position, if it occurs in a file `x.c` it is transferred to file `x.cogent` to a corresponding position.

This implies that translated 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 macro calls occur in a conditional preprocessor directive which is transferred there), a manual solution is required.

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.

External Macro References

Whenever a macro call occurs in a source file, it may reference a macro definition which is external in the sense of Section 2.1.2. For such external references the (translated) definition must be made available in the Cogent compilation unit.

For all preprocessor defined constants (i.e. parameterless nonempty macros not listed in the Gencot manual macro list) Gencot adds the translated macro definition to the file `<package>-exttypes.cogent`. For manually translated macros a separate Gencot macro translation must be specified for external macro definitions. For them the position specification is omitted, they are simply appended to the file `<package>-exttypes.cogent`. If this is not sufficient, because macro calls already occur in `<package>-exttypes.cogent`, they must be inserted manually at the required position.

To avoid introducing additional external references, in the macro replacement text for preprocessor defined constants all macro calls are resolved to existing external reference names or until they are fully resolved. Manually translated macro definitions should handle external macro calls in a similar way.

2.4.4 Include Directives

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

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

and system includes of the form

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

Additionally, there can be include directives where the included file is specified by a preprocessor macro call, they have the form

```
#include MACROCALL
```

for them it cannot be determined whether they are quoted or system includes.

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`.

Macro call includes are normally assumed to be quoted includes and are treated similar.

Quoted includes and macro call includes can be omitted from the translation by adding the file specification to the "Gencot include omission list".

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 preprocessor include directive

```
#include "x-incl.h"
```

If the original include directive occurs in file `y.c` the translated directive is put into the file `y.cogent`. If the original include directive occurs in file `y.h` the translated directive is put into the file `y-incl.h`.

Other quoted include directives and macro call includes are transferred to the Cogent source file without modifications, if necessary they must be processed manually.

2.4.5 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 about non-local name references, i.e. about names which are used in the source file but declared in an included file. An example is a non-local type name reference. To treat the type in Cogent in the correct way, it must be known whether it is mapped to a linear or non-liner type. To decide this, the definition for the type name must be inspected. Hence for C code processing Gencot always reads the source file content together with that of all included files.

The easiest way to do so would be to use the integrated preprocessor of the language-c parser. It is invoked by language-c to preprocess the input to the parser and it would expand all include directives as usual, thus providing access to all information in the included files.

However, the preprocessor would also *process* all directives in all included files. In particular, it would remove all C code in condition groups which do not belong to the current configuration. This is not intended by Gencot, its approach is to remove the directives which belong to the <package> and re-insert them in the target code.

There are two possible approaches how this can be done.

The first approach is to remove the preprocessor directives in advance, before feeding the source to language-c and its preprocessor. All include directives are retained and processed by the language-c preprocessor to include the required content. For this approach to work, the preprocessor directives must be removed in advance from *all* include files in the <package>. Additionally, the include directives must be modified to include the resulting files instead of the original include files.

The second approach is to first include all include files belonging to the <package> in the source, then removing the directives in this file, and finally feeding the result to the language-c preprocessor. This can be done for every

single source file when it is processed by the language-c parser, no processing of other files is necessary in advance.

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.

For simplicity, Gencot assumes that all files included by a quoted include directive belong to the <package>. Hence, the first include step is to simply process all quoted include directives and retain all system include directives in the code. The language-c preprocessor will expand the system includes as usual, thus providing the complete information needed for parsing and processing the C code.

If this is not adequate, Gencot could be extended by the possibility to specify file path patterns for the files to be included in advance for removing preprocessor 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 resulting types may require manual modification.

2.6.1 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.

In C all primitive types are numeric and are mapped by Gencot to a primitive type in Cogent. Note that in C the representation of numeric types may depend on the C version and target system architecture. However, the main goal of Gencot is only to generate Cogent types which are, after translation to C, binary compatible with the original C types. Hence it is sufficient for the numerical types to simply invert the Mapping used by the Cogent compiler.

Together we have the following mappings:

```
char, unsigned/signed char -> U8
short, unsigned/signed short -> U16
int, unsigned/signed int -> U32
long int, unsigned/signed long int -> U64
long long int, unsigned/signed long long int -> U64
```

The only mapping not determined by the Cogent compiler mapping is that for `long int`. For the gcc C version it depends on the architecture and is either the same as `int` (on 32 bit systems) or `lon long int` (on 64 bit systems). Gencot assumes a 64 bit system and maps it like `long long int`.

2.6.2 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 below).

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 Gencot always maps enumeration types to Cogent type `U32`.

If an enumeration type has a tag, Gencot preserves the tag information for the programmer and uses a type name of the form `Enum_tag`, as described in Section 2.1.1. For tagless enums no type names are introduced, they are directly mapped to type `U32`.

The rules for mapping enumeration types are

```
enum { ... } -> U32
enum e { ... } -> Enum_e
enum e -> Enum_e
```

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.

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.

2.6.3 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.

Mapping Struct and Union Types

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 unboxed abstract Cogent type.

Another semantic equivalent would be a Cogent record type, where always all fields but one are taken. However, this type is not binary compatible either, it is translated to a normal struct where every member has another offset. Even if the translated type in C is manually changed to a union, the Cogent take and put operations cannot be used since they respect the field offsets.

Together we have the mapping rules:

```
struct s -> #Struct_s
union s -> #Union_s
```

where `Struct_s` is the Cogent name of a record type corresponding to `struct s` and `Union_s` is the name of the abstract type introduced for `union s`.

As explained in Section 2.1.1, Gencot always introduces a Cogent type name for each struct and union, even if no tag is present in C. Since the tag name is either defined to name a Cogent record or an abstract type, it is always linear and names a boxed type which corresponds to a pointer. Hence, the type name generated for a struct is always used to refer to the type "pointer to struct", the struct type itself is translated to the type name with the unbox operator applied. The same holds for union types.

2.6.4 Array Types

A C array type `t[n]` has the semantics of a consecutive sequence of n instances of type `t`. A value of type `t[n]` is a pointer to the first element and therefore

compatible to type `t*`.

Basically, Cogent does not support accessing elements by an index value in an array. 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 includes 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.

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 the type cannot be mapped to a Cogent array, it must be mapped to an abstract type.

Since the Cogent array type is still under development, the current version of Gencot does not use it for mapping C array types. Instead, all C array types are mapped to Cogent with the help of generated names for abstract types.

Mapping Array Types

A C array type `t[n]` has two slightly different meanings. When it is used for allocating space in memory, it is used to determine the required space as `n * sizeof(t)`. When it is used as declared type for an identifier, it means that the identifier names a (constant) value of type `t*`, since C arrays are always represented by a pointer to the first element. In Cogent both cases must be supported. However, since the first case corresponds to a nonlinear (unboxed) type and the second case corresponds to a linear type, different Cogent types must be used. If the types are directly implemented by `t*` and `t[n]`, the linear type is not the derived pointer type of the nonlinear type, then the difference cannot be implemented by using a single type name with the unbox operator applied or not.

If the nonlinear case is implemented as a struct with the array as its only member, the linear case can be implemented by a pointer to this struct which corresponds in Cogent to the same type with the unbox operator omitted. The pointer to the struct is binary compatible to the pointer to its only member which is binary compatible with the pointer to the first array element. Together, the C array type can be mapped using a single Cogent abstract type.

Instead of using an abstract type for the wrapper struct, Gencot uses a Cogent record with a single field of the abstract type for the array. This makes it possible to use the Cogent take type operator for modelling an uninitialized array. The drawback is that two different type names are required.

Gencot constructs the type names as specified in Section 2.1.1. The array type `t[n]` is mapped using the two types

```
type UAn_T
type An_T = {arr: #UAn_T}
```

Type `UAn_T` is only needed for associating `An_T` with the array type in C, it is implemented in C as

```
typedef t UAn_T[n];
```

The unboxed type `#An_T` is binary compatible with `#UAn_T` and should be used for the unboxed case in all other places. This solution depends on the property, that a C struct with a single array as member has the same memory layout as the array itself. If the C implementation adds padding after the array (it cannot add it before the array according to the C specification), another solution must be used.

Cogent translates take and put operations for a record field to an assignment in C. Therefore, if field `arr` in a value of type `An_T` is put or taken, the C code generated by Cogent will be wrong, since arrays cannot be assigned in C. To avoid this, Gencot defines all abstract functions for array types on the type `An_T` and provides no abstract functions for `UAn_T`. Thus it is useless to access field `arr` in Cogent code and must not be done.

Note that the wrapped array type corresponding to `#An_T` *is* assignable in C, although it is binary compatible with `#UAn_T`. Thus, if a record field has type `#An_T`, corresponding to an embedded array, the take and put operations can be used for the field in the normal way.

If the element type `t` is again an array type, the meaning in C is a multidimensional array. The Cogent type name is constructed by concatenating several array derivation steps of the form `An` For example, the C type `t[2][7]` is mapped using the two Cogent types

```
type UA2A7_T
type A2A7_T = {arr: #UA2A7_T}
```

where `UA2A7_T` is implemented in C as

```
typedef UA7_T UA2A7_T[2];
typedef t UA7_T[7];
```

which in C is equivalent to `t[2][7]`.

A C array type cannot be used as result type of a function. Thus, the remaining possible uses of a C array type are

- as type of a global variable. This is translated using an access function which returns the variable value as a pointer of type `An_T`.
- as type of a function parameter. In C this is "adjusted" to the pointer-to-element type. Since this is binary compatible with type `An_T`, it is used

as parameter type in Cogent. Alternatively the adjusted type could be mapped, resulting in type `P_T` or `T` (in case of record elements). The type `An_T` is preferred here by Gencot since it preserves more information.

- as type of a member in a struct type. Here the size of the array is relevant, therefore the Cogent type `#An_T` is used as type for the corresponding Cogent record field.

- as defining type for a typedef name. Here the type to be used depends on the context where the typedef name is used. As described in Section 2.6.7, Gencot defines the mapped typedef name as an alias for the Cogent type `An_T`. Then applying the unbox operator to the typedef name is equivalent to applying it to `An_T`.

It would be nice to use two generic types for all C array types, instead of generating new names for every C array type. Then the element type need not be mapped to a type name, it may be mapped to an arbitrary Cogent type expression. A generic type for C arrays could have the form

```
type CArrayOf e
```

Then a single mapping in antiquoted C would be sufficient to map all instances of the generic type to their C equivalents. The Cogent compiler would be used to generate a monomorphic type for every usage in a program. The common antiquoted C typedef for the nonlinear case would need to have the form

```
typedef e $ty:(CArrayOf e) []
```

However, the Cogent compiler does not support antiquoted type definitions of this form. Therefore Gencot has to generate the monomorphic types on its own.

Moreover, when a C array type is used for a field in a record, after translation from Cogent to C a type must be used which includes the array size. To be able to specify a corresponding C typedef for the Cogent abstract type name, array types with different size specifications must be mapped to different Cogent abstract type names. This is the reason why the size specification is encoded into the type name, as described in Section 2.1.1.

Together we have the following mapping rule for C arrays with element type `el` and size specification `<size>`, depending on the context of the C array type:

```
el[<size>] -> A<size>_El , #A<size>_El
el[<complex size>] -> A_El, #A_El
el[] -> A_El , #A_El
el[*] -> A_El , #A_El
```

where El is the result of mapping the element type `el` to a Cogent type name. If the size specification is too complex (not a literal or single identifier) it is omitted and the type must be handled manually. If the element type is again a derived type the underscore is omitted.

If `el` is mapped to a type name with the unbox operator applied, which is not a function pointer type, a letter "`U`" is prepended to `El` in the form of an additional derivation step. This can happen for struct and union types. For example, the array type `struct s ... [5]` is mapped to

```
type UA5U_Struct_s
type A5U_Struct_s = {arr: #UA5U_Struct_s}
```

whereas the type nyme `A5_Struct_s` is the mapping for the type `(struct s ... *)[5]` for an array of struct pointers.

2.6.5 Function Types

C function types of the form `t (...)` are used in C only for declaring or defining functions or when a typedef name is defined for a function type. 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.

Mapping Function Pointers

In other places, however, Cogent does not translate its function types to C function pointers. Instead, it uses a C enumaration type where every known function has an associated enumeration constant. Whenever a function is bound to a variable, passed as a parameter or is invoked through a function pointer, it is represented by this enumeration constant in C, i.e., by an integer value. For function invocation Cogent generates dispatcher functions which receive the integer value as an argument and invoke the corresponding C function.

Binary compatibility is only relevant when a function is stored, then it is always represented by the enumeration constant in C generated from Cogent. Thus, a C function pointer type cannot be mapped by Gencot to a Cogent function type, since this will not be binary compatible. Instead, it must be mapped to a Cogent abstract type together with abstract functions which translate between the abstract type and the Cogent function type (needed when invoking the function in Cogent).

Together, Gencot treats C function types and C function pointer types in completely different ways. It maps C function types to Cogent function types and it maps C function pointer types to Cogent abstract types.

Mapping all C function pointer types to Cogent abstract types is the reason why names are generated for them, as described in Section 2.1.1. The goal is to use the same name for every occurrence of a C function pointer type. Thus it is infeasible to generate the name from the occurrence position (file name and line number). If an arbitrarily generated name would be used, Gencot could not separately compile different C files. If the parameter types would not be encoded in the name, it would not be possible to define abstract Cogent functions for translating between the abstract type and the Cogent function type.

Although the C function pointer is a pointer, the pointer target value (the machine code implementing the function) normally cannot be modified. Hence, semantically a function pointer type does not correspond to a linear type in Cogent, it could be represented by a readonly type or by an unboxed type. Gencot uses an unboxed type since all other types are also mapped to either boxed or unboxed types.

Together the rule for mapping a function pointer type is

```
t0 (*)(t1, ..., tn) -> #F_XT1X...XTnX_T0
```

where `Ti` is the name to which `ti` is mapped and `X` is a letter not occurring in the `Ti`. If a `ti` is mapped to a type name with the unbox operator applied, which is not a function pointer type, a letter "U" is prepended to `Ti` in the form of a derivation step. This can only happen for struct and union types since array and function types are adjusted to pointer types when used as parameter type, and are not allowed as function result types in C. In particular, if a parameter type `ti` is an array type, it is mapped to the name for the linear case. For example, the C function pointer type `int (*)(int, int[10])` is mapped to

```
#F_XU32XA10_U32X_U32
```

Mapping Function Parameters

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 a variable number of parameters is specified such as in `t (...)` ("variadic function type") must be treated manually in specific ways. Gencot maps variadic function types with an additional last parameter type `VariadicCogentParameters`. This pseudo type is intended to inform the developer that manual action is required. The bang operator is applied as a hint that no modifications are returned. For a function pointer, the corresponding Cogent type name has the form `F_X...XR_VariadicCogentParametersX_<result type>`.

C function types where the parameters are omitted, such as in `t ()` ("incomplete function type") 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. Note that incomplete function types cannot be used for function definitions, only for function pointers and for declarations of external functions. Gencot maps pointers to incomplete function types to type names of the form `F_UnknownCogentParameters_<result type>`. Gencot does not translate declarations of functions with incomplete types, these must be added manually. This behavior of Gencot is also exploited to handle manually translated macros, as described in Section 3.7.3.

All function pointers are represented by an integer in Cogent, hence a C function pointer type could be mapped (through the from/invk/to translation functions) to an arbitrary Cogent function type. Of course, to be useful the types of the parameters and result should be mapped as well. This is done in the same way as described above for function types.

Together the rules for mapping function types (and function pointer types through the from/invk/to translation functions) are

```
t(t1, ..., tn) -> (T1, ..., Tn) -> T
t(void) -> () -> T
t(t1,...,tn,...) -> (T1, ..., Tn, Variadic_Cogent_Parameters!) -> T
t(*)() -> #F_UnknownCogentParameters_T
```

Linear and Readonly Parameter Types

Like every other type, the type of a function parameter may be readonly because this property can be derived from the C type information, as described in Section 2.6.8. However, in a C program often a parameter is actually never modified, although this property cannot be derived from the C type information. Since this is an important property for working with linear types in Cogent, Gencot tries to capture these cases as well and make the parameter type readonly.

For a parameter with linear type, the function can only be defined in Cogent if the parameter is not discarded, i.e. it must be part of the result. Gencot assumes the most simple handling of this case, where the result is a tuple consisting of the original result of the C function together with a component for every parameter of linear type. To avoid these extra result components, Gencot tries to make all parameters with linear type readonly and add only the remaining parameters to the result type.

A parameter of linear type may not be discarded in Cogent, but it may be passed to an abstract function which discards it. In this case the parameter must not be returned, although it does not have readonly type.

Gencot determines the information about parameter modification and parameter discarding in a semi-automatic way as described in Section 2.10. It uses this information to make parameter types readonly or add parameters to the function result, according to the following rules:

- If the C type of a parameter is not linear according to Section 2.6.8, or if it is discarded according to Section 2.10, it is translated as described before.
- Otherwise, if the C type of the parameter is readonly according to Section 2.6.8, or is not modified according to Section 2.10 its translated type

is made readonly by applying the bang operator `!`. For a single such parameter of type `r` the translation rule becomes

```
t(t1, ... r, ... tn) -> (T1, ... R!, ... Tn) -> T
```

- Otherwise, if the parameter is already returned as result of the function, it is translated as described before.
- Otherwise, the function result is changed to a tuple and the parameter is added as component to that tuple. For a single such parameter of type `l` the translation rule becomes

```
t(t1, ... l, ... tn) -> (T1, ... L, ... Tn) -> (T,L)
```

If the result is modified to a tuple, the first component is the original function result and the remaining components are the parameters of linear type in their order as they occur in the parameter tuple.

Linear and Readonly Parameter Types for Function Pointers

Since function pointer types shall be translated to function types through the from/invk/to translation functions, the information about linear and readonly parameter types must also be available for them. Moreover, the same C function pointer type may be mapped to two different Cogent types depending on whether a certain parameter is modified or not.

To support this the information about linear and readonly parameter types is coded into the Cogent type name as follows. If the parameter type is readonly or it is linear but the parameter is not modified by the function the mapped parameter type is prepended with a letter "`R`" in the form of an additional derivation step (i.e. separated by an underscore if the parameter type is not derived). Otherwise, if the parameter type is linear and the parameter is neither discarded nor returned as the function result, the mapped parameter type is prepended with a letter "`L`" in the same way. Otherwise, the mapped parameter type is unchanged.

Note that the result tuple used for linear parameter types is not coded directly into the type, it must be reconstructed from the parameter type information (all parameters with a prepended "`L`").

As an example consider the C function pointer type

```
int (*)(char[4], const char*)
```

where the first parameter may be modified by the function referenced by a pointer of that type. The type is translated to the Cogent type name

```
F_XLA4_U8XRP_U8X_U32
```

which is translated by the from/invk/to functions to the Cogent function type

```
(A4_U8,P_U8!) -> (U32,A4_U8)
```

If the first parameter is known to be never modified, the Cogent typename instead will be

```
F_XRA4_U8XRP_U8X_U32
```

and the corresponding Cogent function type will be

```
(A4_U8!,P_U8!) -> U32
```

2.6.6 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 this support can be employed by mapping the type to a Cogent record with a single field of the type referenced by the pointer.

In C, a pointer-to-array type is not the type of pointers to the array address, instead its values are array addresses. The difference from the array type is only that when applying the index operator `[]`, the whole array is selected instead of only the first element. In Cogent the corresponding type is a Cogent record with a single field of the mapped array type. Note, that this is equivalent to the mapping of the array type itself, as defined in Section 2.6.4.

Mapping Pointer Types

A pointer to a function type is mapped to an unboxed abstract type, as described in Section 2.6.5.

A pointer type `t*` to a struct is mapped to the corresponding boxed type, that means, it is mapped like the struct type `t`, but the unbox operator is omitted.

A pointer type `t*` where `t` is a union type is mapped in the same way to the corresponding boxed type, omitting the unbox operator from the mapped type `t`. Thus no support for accessing the referenced union is provided. The reason is, that access to the union as a whole is mostly useless and furthe access to the union mebers cannot be provided. For consistency, Gencot treats union types in the same way as struct types.

A pointer type `t*` where `t` is a primitive type, an enum type, or again a pointer type is mapped to a boxed record with a single field `cont` of the type to which `t` is mapped. For every such type a name is introduced as described in Section 2.1.1. This makes the Cogent program slightly more readable and the name is required as parameter type name in mapped function pointer types as described in Section 2.6.5. For the same reason Gencot does not use a single generic type for all such pointer types. Instead, like for arrays, it creates a monomorphic type for every pointer type used in the program.

The resulting type is defined in the form

```
type P_<type name> = { cont: <type name> }
```

If the `<type name>` is a mapped derived type the underscore is omitted.

Values of such types are binary compatible to the C pointer type and they can be dereferenced with the help of the Cogent take and put operations, thus supporting the full functionality of the C pointer.

A pointer type `t*` where `t` is an array type is mapped to a boxed record with a single field `cont` of the unboxed array type:

```
type PAn_<type name> = { cont: #An_<type name> }
```

where `An_<type name>` is the mapping of the array type. This pointer type is binary compatible with the mapped array type `An_<type name>`, however it is not equivalent as a Cogent type. This is intended since other abstract functions should be available for the pointer than for the array. In particular, since the field `cont` has type `#An_<type name>` instead of `#UAn_<type name>` it can be accessed using take and put operations in Cogent, as for the other mapped pointer types.

Finally, a pointer to `void` is mapped to the abstract type

```
type P_Void
```

Here the type of the referenced data is unknown and no support for dereferencing it can be provided in Cogent, all processing must be implemented by specific abstract functions.

Together we have the mapping rules for pointer types:

```
void * -> P_Void
r (*)(...) -> #F_X...X_R
struct s * -> Struct_s = { ... }
union s * -> Union_s
(*el)[...] -> PA..._El = { cont: #A..._El }
otherwise: t * -> P_T = { cont: T }
```

where `Struct_s` and `Union_s` are the names introduced for the struct or union types and `R`, `El` and `T` are the Cogent type names to which `r`, `el` and `t` are mapped, respectively. If `El` and `T` are derived, the underscore is omitted.

2.6.7 Defined Type Names

In C a typedef can be used to define a name for every possible type. In principle, it would be possible to map a typedef name by resolving it to its type and then mapping this type as described above. However, the typedef name often bears information for the programmer, hence the goal for Gencot is to preserve this information and map the typedef name to the corresponding Cogent type name which is defined by translating the typedef to a Cogent type definition.

A typedef name can be used in C to derive a pointer type from it. The mapping of pointer types depends on the kind of base type, as described above. If the base type is a struct, union, or array type, the pointer type is mapped to the mapped typedef name omitting the unbox operator. If the base type is a function type, the pointer type is mapped to the mapped typedef name with the unbox operator applied (as for all function pointer types). Otherwise, a type name of the form `P_T` is used where `T` is the mapped typedef name.

Always mapping pointer types derived from typedef names with the help of `P_T` would result in the following situation for the case where a typedef name is defined for a struct:

```
typedef struct s snam
mapping: struct s -> #Struct_s
mapping: struct s * -> Struct_s
mapping: snam -> Cogent_snam
mapping: snam * -> P_Cogent_snam
```

where `Struct_s` is the name of the Cogent record type corresponding to `struct s`. The problem here is that `snam *` is mapped using `P_T`, resolving to the Cogent type `P_#Struct_s` instead of `Struct_s`, preventing access to the record in Cogent.

Therefore Gencot treats every typedef name resolving to a struct, union, or array type as if it would resolve to the corresponding pointer type. The plain name is mapped with the unbox operator applied, the pointer type derived from it is mapped without unbox operator applied.

A typedef name resolving to a function type is also treated as if it would resolve to the corresponding function pointer type. The pointer type derived from it is mapped to the mapped typedef name with the unbox operator applied (since it is a function pointer type). This implies that the plain name cannot be mapped. However, that is no restriction, since a C typedef name for a function type can only be used for constructing the corresponding C function pointer type, either explicitly or implicitly by adjustment. In particular, it cannot be used for defining a function of that type, since a definition must always include the parameter names.

The resulting mapping rules are for function type names:

```
tn -> #TN
tn* -> #TN
```

for names of function pointer types:

```
tn -> #TN
```

for names of a struct or union type:

```
tn -> #TN
tn* -> TN
```

for names of an array type:

```
tn -> TN, #TN
```

depending on its usage context, and for all other type names:

```
tn -> TN
```

where `TN` is the name mapping of `tn`.

This implies, that also the Cogent type definitions generated from a C typedef have to be modified, if the target type is a struct or union type. In this case Gencot translates the typedef to a Cogent type definition which defines the mapped typedef name as a synonym for the corresponding boxed type in Cogent.

If the target type is a function type Gencot translates typedef to a Cogent type definition which defines the mapped typedef name as a synonym for the translated function pointer type. Here, the unbox operator can be applied either to the righthand side of the type definition or to every occurrence of the mapped typedef name or both. Gencot applies it to both to make it apparent that in both cases the type is not linear.

If the target type in a typedef is another typedef name Gencot resolves it to the final target type before applying the rules above.

2.6.8 Linear and Readonly Types

C types can be qualified as `const`. This means, the values of the type are immutable and could be stored in a readonly memory. A variable declared with a readonly type is initialized with a value and cannot be modified afterwards. The immutability of an aggregate type also implies that values cannot be modified by modifying parts: for a struct the fields cannot be modified and for an array the elements cannot be modified. This behavior corresponds to the behavior of all primitive and unboxed types in Cogent.

If a C type is not qualified as `const`, stored values of the type may be modified. This may have non-local effects if the stored value is shared (part of several other values). In Cogent, values of primitive and unboxed types cannot be shared (only copies can be part of other values). Therefore a modification of the C value always corresponds to replacing the value bound to a variable in Cogent. This can be represented by binding the new value to a variable of the same name which will shadow the previous binding. Together, this means that a `const` qualifier is irrelevant whenever a C type is translated to a primitive or unboxed type in Cogent.

The situation is different for C pointer types which are translated to linear types in Cogent. Values of linear types may be modified using put and take operations in Cogent, but they are restricted in their use. Put and take operations correspond to modifications of the value referenced by the pointer. Thus, a `const` qualification of the pointer type is still irrelevant for them, however, a `const` qualification of the pointer's *base type* means that put and take operations are not possible. This case is supported by Cogent as readonly types, which are not restricted in their use in the same way as linear types.

Note, however, that Cogent does not separate between pointers and their referenced values: the referenced value is treated as part of the linear value. If the referenced value itself contains references, the values referenced by them are also treated as part of the overall value. This implies, that a readonly type in Cogent corresponds to a C pointer type with `const` qualified base type where all components with a pointer type recursively have the same property.

It further implies, that a C type also corresponds to a linear type in Cogent, if it directly or indirectly *contains* pointers where the base type is not `const` qualified. This may be the case for struct or union types (members may have such pointer types) or for array types (the elements may have such a pointer type).

An exception are C pointers to functions. It is assumed that the function code cannot be modified, hence a C pointer to function is treated like a primitive type in Cogent.

Gencot tests every C type for being a pointer or containing a pointer. If this is the case, the translated Cogent type is known to be linear. The C type is then further tested whether all pointers have a `const` qualified base type. If this is the case, the type is translated to a Cogent readonly type, by applying the bang operator `!` to the type after translating it as described in Section 2.6.6.

In particular, the readonly property is valid for all pointer types where the base type contains no pointers, such as `const char*`.

2.7 Basic Operations for Datatypes

For working with the mapped C datatypes, in particular for those mapped using abstract Cogent types, Gencot provides support by defining and implementing polymorphic Cogent functions, some of which are abstract and some of which are implemented in Cogent.

Although the operations provided for different kinds of datatypes have different semantics, they have common properties and most are represented by common polymorphic functions. Here the operations are first introduced conceptually, together with the polymorphic functions, then they are presented for specific kinds of data types. Gencot provides the polymorphic function definitions in separate Cogent files which can be included in a Cogent source.

2.7.1 Dummy Expressions

For every mapped type Gencot defines a dummy expression of that type. It is used as a replacement for the actual body when translating C functions (see Section 2.9).

The dummy expression for all mapped numerical and enumeration types is the literal `0`. The dummy expression for the unit type (used for functions which have `void` as result type) is the unit value `()`.

For all other types Gencot provides in `include/gencot/DummyExpr.cogent` the polymorphic abstract function

```
gencotDummy: all(a). () -> a
```

from the unit type to every possible Cogent type. Since the dummy expressions are intended to be eliminated before compiling the C code generated by the Cogent compiler, Gencot does not provide C definitions for this abstract function.

If a function modifies parameters of linear type, it is translated by Gencot to return a tuple consisting of the original result and all such parameters (see Section 2.10). The dummy result expression is then built as a tuple with a dummy expression for the original result as the first component and the unmodified parameters as the remaining components.

2.7.2 Default Values

Conceptually, Gencot provides a default value for every regular non-function Cogent type. For the primitive numeric types it is `0`, for type `String` it is `""`, and for type `()` it is the unit value `()`.

For a regular tuple (which has no component of linear or readonly type) it is the tuple of default values. For a regular unboxed record it is the record where all fields have their default value. For a regular variant type it is the default value of the first variant.

Gencot defines the polymorphic abstract function

```
defaultVal: all(out:<DSE). () -> out
```

in `include/gencot/Default.cogent`.

Gencot provides instances for the numeric types `U8,U16,U32,U64` which return 0, for the unit type `()` which returns `()`, and for type `String` which returns `""`.

Other instances must currently be defined manually, Gencot could be extended to provide automatic support for them as well.

2.7.3 Creating and Disposing Boxed Values

Since all pointer types are mapped to Cogent linear types, Cogent does not provide support for creating values of these types ("boxed values"). In C a pointer can be created using the address operator `&` or by allocating data on the heap using a C standard function such as `malloc`. The address operator is supported by Gencot only for data on the heap, as explained in Section 4.5.6. Therefore, the basic functionality for pointer creation is allocation on the heap. This must be provided as an abstract Cogent function.

Since values of a linear type cannot be discarded in Cogent, another abstract Cogent function is required for disposing such values, implemented by using the C standard function `free`.

Gencot provides the polymorphic abstract functions defined in `include/gencot/Memory.cogent`:

```
create : all(evt). Heap -> Result (evt,Heap) Heap
dispose : all(evt). (evt,Heap) -> Heap
```

for creating and disposing values of linear types. Gencot provides instance implementations for all linear types, i.e. all (boxed) record types and abstract types. `Heap` is an abstract data type for modelling the C heap where the data structures are allocated, it is defined in `include/gencot/Memory.cogent`. `Result` is a variant type defined in the Cogent standard library (usually abbreviated as `R`) with variants for the success and error cases. In the success case `create` returns the newly allocated boxed value and the modified heap, in the error case it only returns the heap.

The `create` instances only allocate space on the heap but do not initialize it. To model this property in Cogent, Gencot uses two different Cogent types for every linear type to represent uninitialized ("empty") and initialized ("valid") values. Conceptually, the instances of `create` and `dispose` are only defined for the empty-value types. The function `create` returns empty values, the function `dispose` expects empty values. In particular, empty values cannot contain linear parts (pointers to other memory regions) and can thus be safely discarded by deallocating their memory space.

Note that according to their definition, the functions `create` and `dispose` are defined for arbitrary types `evt`. However, Gencot only provides instances for (empty-value) linear types. Since it is not possible to express this type property in Cogent, the use of unsupported instances of both functions are not detected by the Cogent compiler.

When Gencot maps a C type to a linear Cogent type, it always uses a Cogent record type (see Section 2.6). For these types the empty-value type is implemented by the record with all fields taken. Gencot defines in `include/gencot/Memory.cogent` the preprocessor macro

```
EVTYPE(vvt)
```

which expands to the empty-value type `vvt take (..)` corresponding to the valid-value type `vvt`. It can be used to specify the correct instances of `create` and `dispose` for all types implemented as Cogent record. For example, the `create` instance for a mapped array type `A5_U32` can be specified as

```
create[EVTYPE(A5_U32)]
```

2.7.4 Modifying Boxed Values

Boxed records can be modified in Cogent by the put and take operations which in the simplest case have the form

```
let r' = r { f = v } in ...
let r' { f = p } = r in ...
```

In the put operation `r` is the old record value, `f` is the field to be written, `r'` is the new ("modified") record value and `v` is the new field value. The type of `r'` is either the same as for `r` or it differs, if field `f` was taken in `r`. In the take operation `r` is the old record value, `f` is the field to be taken, `r'` is the record without `f` and `p` is a pattern for binding the old field value. The type of `r'` is always different from that of `r`.

Note that in the C translation of this code the boxed records correspond to pointers to structs and for both operations the pointers `r` and `r'` are the same.

We generalize this idea by defining operations for modifying boxed values in the form of Cogent functions. To provide a common framework for modification operations, Gencot defines function types for such modification functions in `include/gencot/ModFun.cogent`. The basic function types are

```
type ModTypeFun obj res arg out = (obj,arg) -> (res,out)
type ModFun obj arg out = ModTypeFun obj obj arg out
```

Here `obj` is the type of the value to be modified and `res` is its type after the modification. Type `ModTypeFun` covers the general case where the type of the new value is different from that of the old value (although the pointer values in C are the same!), `ModFun` covers the more special case where the type is the same before and after the modification. In both cases a value of type `arg` is passed to the modification function, it may provide information about how to modify the value, and the modification function additionally returns a value of type `out`, e.g., an error code. The functions `fst` and `snd` defined in the Cogent standard library can be used to retrieve the modified value and the additional output.

Note, that the additional property of a "modification function", that the result pointer must be the same as the argument pointer, cannot be expressed in Cogent (or any other functional language). The type `ModTypeFun` is used by Gencot as an informal marker for such functions. A function implemented in Cogent is a modification function, if it only applies take and put operations to its first argument or other modification functions. An abstract Cogent function is a modification function is a modification function if the C implementation always returns the same pointer, without deallocating it in between.

The put operation can now be represented as a function of type `ModTypeFun R (R put f) V ()` where `R` is the type of `r` and `V` that of field `f` and value `v`. If the field is not taken in `R` the type is `ModFun R V ()`. The take operation

can be represented as a function of type `ModTypeFun R (R take f) () V`, it returns a pair of the remaining record and the taken field value.

A typical pattern for modifying a record field `f` is a combination of a take and a put operation of the form

```
let r' { f = h } = r
and r'' = r' { f = chg(h) }
```

where a function `chg` is used to determine the new field value from the old field value. We can generalize function `chg` to a function of type `(fld,arg)->(fld,out)` where `fld` is the type of field `f`. It takes additional input of type `arg` and returns additional output of type `out`. However, it need not be a modification function, since type `fld` need not be linear and `chg` may map its first argument to an arbitrary other value of the same type. Gencot defines the function type

```
type ChgFun obj arg out = (obj,arg) -> (obj,out)
```

for this kind of "change functions". Its meaning is the plain Cogent semantics of the type definition, no additional property is included.

Now we can define a higher order function for the combined field modification as

```
modify (r,(chg,x)) =
  let r' { f = h } = r
  and (v,y) = chg(h,x)
  in (r' { f = v }, y)
```

It has the function type `ModFun R (ChgFun V arg out, arg) out`, accepts as additional input a pair of a change function for the field value and its additional input, and returns the modified record and the additional result of the field change function. Note that `modify` also respects the type constraints if the field type `V` is linear. The field value is neither duplicated nor discarded.

For this kind of modifying a part Gencot defines the corresponding generalized function type

```
type ChgPartFun obj prt arg out =
  ModFun obj (ChgFun prt arg out, arg) out
```

where `obj` is the type of the modified object, `prt` is the type of the part to be changed, `arg` is the type of the information passed to the part change function, and `out` is the type of additional output of both functions.

Since every `ModFun` is also a `ChgFun`, modification functions of this type can be chained to modify parts arbitrarily deep embedded in other parts.

As an example, to change a part `p` of type `P` in a part `q` of type `Q` in a value `r` of type `R` an expression of the form

```
modifyQInR (r, (modifyPInQ, (chg, arg)))
```

modifies `r` by replacing `p` by `chg (p,arg)`. To make the modification functions generic for the type of the additional input to the part modification function and for the type of its additional output they can be defined as polymorphic:

```
modifyQInR: all(arg,out). ChgPartFun R Q arg out
modifyPInQ: all(arg,out). ChgPartFun Q P arg out
chg: ChgFun P A O
```

If the change function `chg` needs as input information of nonlinear type from other linear parts of `q` or `r` it is not possible to pass these parts to `chg`. For a record type `R` either they must be taken from `r` and put back in after the modification, then the type of `r` is `R take (...)` and `modifyQInR` cannot be applied because of type incompatibility. Or the parts are accessed as readonly in a banged context for `r`, then the readonly parts cannot escape from the banged context to be passed to the modification operation (which must be outside of the banged context since it modifies `r`). Instead, the required nonlinear information must be retrieved from the linear parts in a banged context for `r`. Since it is nonlinear it may escape from the context and can be passed to the modification operation.

In C it is a common pattern to pass pointers to other parts of a data structure around for efficiency and access values through these pointers only when needed to modify parts of the structure. In the Cogent translation the values must be accessed separately in a banged context and then passed to the modification operation as copies.

If type `prt` of the part to be modified is not linear, function `modify` removes the part's value from the structure, passes it to the part change function and puts the result back into the structure. This is inefficient for large parts when the part is only *modified* by changing a small subpart. In C the typical way of dealing with this situation is to pass a pointer to the part change function instead.

The same approach can be used in Cogent defining a function `modref` which works like `modify`, but uses a part modification function of type `ModFun` and passes a pointer to it instead of a copy of the part. Since it returns the same pointer, the part value can be modified "in-place". In Cogent the pointer corresponds to a value of linear type, so the part modification function can usually be implemented in Cogent. Function `modref`, instead, must be abstract and implemented in C with the help of the address operator `&`.

Gencot defines the function type

```
type ModPartFun obj pprt arg out =
  ModFun obj (ModFun pprt arg out, arg) out
```

which can be used to specify the type of `modref`:

```
modref: ModPartFun obj pprt arg out
```

Here `pprt` is the pointer type corresponding to the part's type `prt`. Note that Gencot's type mapping scheme supports for every mapped C type a corresponding pointer type.

Function `modref` can be chained with other `modref` functions and with `modify` functions in the same way as described for `modify` functions.

Of course, using a pointer to the in-place part of the boxed value introduces sharing between both. However, the part modification function has no access to the boxed value other than by the pointer to the part. It could only get access if the boxed value or a part of it would be passed to it using the additional input of type `arg`. But since that is passed to `modref` together with the boxed value itself, the Cogent type checking rules prevent this as a double use of the boxed value. Thus the approach is safe and the part modification function can work with the pointer according to the usual Cogent rules, as long as it always

returns the same pointer as result value. This means it cannot dispose it and it cannot store it in another structured value and return a different pointer of the same type. Function `modref` then simply discards the returned pointer so that the sharing ends when it completes.

2.7.5 Initializing and Clearing Boxed Values

Gencot uses the terms "initialization" and "clearing" for the conversions between empty and valid boxed values. After a boxed value has been created it must be initialized to be used, before it is disposed it must be cleared.

Initialization must set every part of a structured value to a valid value. For parts of nonlinear type this is straightforward, since values of nonlinear types can be directly denoted in Cogent programs in most cases. Parts of unboxed record and abstract types are made valid by passing a pointer to the part to an initialization function for the corresponding boxed part value (as described for function `modref` in Section 2.7.4).

For parts of linear type there are two possible approaches: they can be created during initialization or they can be passed as arguments. If they are created, the initialization function needs the heap as additional in- and output. Otherwise it takes all parts of linear type as additional input. Of course, if several parts of linear type exist, some of them can be created and some passed as arguments.

Parts of readonly type `S!` cannot be initialized by creating a value for them. If a value is created in the initialization function it must be banged there but then it may not leave the banged context. The readonly value must either be created by a function which returns a value of type `S!`, or it must be passed as argument to the initialization function.

Clearing must convert every part of a structured value to an empty value. For parts of nonlinear type nothing needs to be done, or the value can be explicitly set to a default value to overwrite the stored information for security reason. Parts of unboxed record and abstract types are made empty by passing a pointer to a clearing function for the boxed part value.

Parts of linear type, dually to initializing them, can be disposed during clearing, or they can be returned as result, so that they are not discarded. If they are disposed, the clearing function takes the heap as additional in- and output. Otherwise it returns all parts of linear type as output. If several parts of linear type exist, some of them can be disposed and some returned as results.

Parts of readonly type need no specific treatment during clearing, since they can be discarded in the same way as parts of nonlinear type.

Initialization and clearing functions are modification functions in the sense of Section 2.7.4. Translated to C they always return the same pointer they received as input. Gencot defines the following function types in `Memory.cogent`:

```
type IniFun evt vvt arg out = ModTypeFun evt (Result vvt evt) arg out
type ClrFun vvt evt arg out = ModTypeFun vvt evt arg out
```

They can be used by the developer to mark functions as initialization or clearing functions. This is purely informal since no constraint between the type parameters can be enforced by Cogent, so a correct pair of empty-value type and the corresponding valid-value type must be specified by the developer.

The type for initialization functions returns a result of type `Result vvt evt`, using the generic type `Result` from the Cogent standard library. The reason is that we assume that initialization may fail. This is typically the case when initializing a part of the value includes allocating space on the heap. Whenever there is any problem during initialization the initialization is rolled back and the original empty value of type `evt` is returned. For clearing we always assume that no error can occur, then the rollback is always possible without causing another error.

Initialization and clearing functions are defined by Gencot to always expect additional input and output values of arbitrary types `arg` and `out`. The input value can be used to specify default values to be used, the heap for creating and disposing parts of linear type, and initialization and clearing functions for parts of the value. If a function only uses fixed default values and functions for parts the unit type `()` is used as type `arg`. The output value can be used to return an error code, the modified heap, or parts of linear type which have not been disposed.

An initialization or clearing function for a type with several parts must handle all parts together because it must transform from the empty-value type, where all parts are taken, to the valid-value type, where all parts are present. If an initialization or clearing function handles only one part, its type must respect which other parts are taken and which are not. This is not feasible for types with many parts. However, if all parts must be handled together, there are many ways how to do so, especially if there are parts of linear and/or readonly type.

Gencot provides the following polymorphic abstract initialization and clearing functions which are defined for only some of these cases:

```
initFull : all(evt,vvt). IniFun evt vvt #vvt ()
clearFull : all(vvt,evt). ClrFun vvt evt () #vvt
initHeap : all(evt,vvt:<E). IniFun evt vvt Heap Heap
clearHeap : all(vvt:<E,evt). ClrFun vvt evt Heap Heap
initSimp : all(evt,vvt:<DSE). IniFun evt vvt () ()
clearSimp : all(vvt:<DSE,evt). ClrFun vvt evt () ()
```

Generally, these functions are only defined if `evt` is the empty-value type corresponding to the valid-value type `vvt`. Since this cannot be expressed by constraints in Cogent, Gencot defines the preprocessor macros:

```
INIT(<k>,vvt) -> init<k>[EVTYPE(vvt),vvt]
CLEAR(<k>,vvt) -> clear<k>[vvt,EVTYPE(vvt)]
```

They can be used to specify valid instances of the functions with the correct types so that the constraints between them are fulfilled and the Cogent typechecker can be used to test for correct application. For example, `INIT(Full,vvt)` expands to `initFull[EVTYPE(vvt),vvt]`.

The first two functions pass the full content as argument: the additional input type for initialization and the additional result type for clearing is the unboxed type `#vvt`. It is used to pass all content for the referenced memory region to `initFull` which copies it there. For `clearFull` it is used to return all content, in particular, all content of linear type, so that it is not discarded.

The second pair passes the heap as additional input and result. Additionally, `vvt` must be escapeable (no readonly parts), because these cannot be initialized

internally. Function `initHeap` allocates all parts of linear type on the heap and `clearHeap` disposes them. Parts of type `MayNull a` (see Section 2.7.10) are initialzed to `null` and are disposed only if they are not `null`. All parts of non-primitive type are initialized or cleared using function `initHeap` or `clearHeap`, respectively. All parts of primitive type are initialized to their default value `defaultVal ()` and are cleared by doing nothing.

The third pair passes no additional information. Moreover, `vvt` must be regular (no linear or readonly parts). Functions `initSimp` and `clearSimp` work as `initHeap` or `clearHeap`, but the heap is not required since there are no linear parts which must be allocated or disposed.

The first pair is the most general, it is applicable to all kinds of valid-value types `vvt`. However, it bears the most overhead, since all content must be passed as argument and copied to the memory to be initialized. The other two pairs support "in-place" operation, however, they are more restrictive.

Although all initialization functions return a value of type `Result vvt evt`, the functions `initFull` and `initSimp` can never fail and will always return a value `Success v` where `v` is of type `vvt`. The functions are defined in this way for consistency, so that all have a compatible type and can be passed as argument to a common function parameter. It is assumed that the overhead for the additional unnecessary checks are not crucial for the performance, since initialization and clearing are no frequent operations.

Functions `clearFull`, `clearHeap`, and `clearSimp` do not overwrite the referenced memory with a "clearing value". If this is required, a manually defined clearing function must be used instead.

For specific types `vvt` custom initialization and clearing functions can be defined manually. Typically, they pass values for some parts as parameters (in particular those of readonly types), and use default values or heap allocation for the others.

2.7.6 Accessing Parts of Structured Values

For working with a structured value it is often necessary to access its parts for reading or modifying them. Gencot supports the following conceptual operations on structured values:

`get` access the value of a part for reading,

`set` replace the value of a part by a given value, discarding the old value,

`exchng` replace the value of a part by a given value, returning the old value,

`modify` apply a change function to a part.

Every function accesses only a single part of the structured value, if several parts must be accessed at the same time custom operations must be defined manually.

Depending on the type of structured value and the kind of specifying the part, the part may safely exist or not. For example, for a record field specified by its name it can be statically determined whether it exists, for an array element specified by a calculated index value this is not the case.

The actual function types of the operations differ depending on the way how the part is specified for them and whether it safely exists. However, if the

structured value has type `T` and the part has type `S` the conceptual functionalities are as follows:

The function `get` has functionality `T! -> S!`, if the part safely exists. It expects a readonly value as input and returns a readonly copy of the part's value. This operation can be defined in Cogent for arbitrary types `S` because the returned value can be shared with the value remaining in the structure since both are readonly.

If the part does not safely exists there are two possible functionalities: either the function returns a variant value, or it returns a default value if the part does not exists. The variant value is the simpler and "cleaner" form, possible variant types are `Option` and `Result` from the Cogent standard library (the former treating the part semantically as "optional", the latter treating its nonexistence as an "error"). However, the use of the variant value introduces an overhead, since Cogent implements it as a record. It must be constructed, passed on the stack as function result, and then tested and deconstructed; for very frequent accesses to a part this may cause a relevant performance reduction. Therefore an alternative form should be provided which always passes the part's value directly, using a default if the part does not exist. It should only be used if it is clear from the context that the part exists. A possible choice for the default value is `defaultVal` (defined in Section 2.7.2), however that is restricted to parts of regular type, therefore other solutions are required for other types.

An alternative to the function `get` would be a function which takes a modifyable structure and returns the structure together with the element. However, this would be cumbersome in many applications and misleading for proofs, since the function never modifies the structure.

The function `set` has functionality `(T,S) -> (T,())` and is a modification function in the sense of Section 2.7.4, hence its type can be denoted as `ModFun T S ()`. It expects a structure and the new value as input and returns the structure where only the value of the part has been replaced by the new value. The result value is structured as a pair, so that the function has the form of a modification function. Since the old value of the part is discarded, function `set` is only defined if type `S` is discardable. If the part does not exist, the function returns the unmodified structure.

The function `exchng` has functionality `(T,S) -> (T,S)` and is also a modification function, so its type can be denoted as `ModFun T S S`. It works like `set`, but instead of discarding the old value of the part it returns it in the result. This can be done for arbitrary types `S` since values of this type are neither shared nor discarded. If the part does not exist, the function returns the unmodified structure together with the input value.

The function `modify` is a modification function and has functionality `ChgPartFun T S A O` where `A` and `O` are arbitrary types. According to the definition of `ChgPartFun` `modify` applies a part change function of type `ChgFun S A O` to change the part. All types may be linear, since the corresponding values are only passed through to the part change function and back. The part change function determines the new part value from the old part value. If the part's type `S` is linear it can be implemented by actually modifying the part's value. In particular, this can be done by again using `modify` as part change function changing a part's part, as it has been described for the `modify` function in Section 2.7.4.

If the part does not exist, function `modify` returns the unmodified structure.

However, it must also return a value of type `O`, although the part change function is never executed. Similar as for function `get` there are two possibilities: returning a variant value or a default value. A third possibility here is to use the same type `A` as additional input and result to the part change function. Then, if the part change function is not executed, its additional input can be returned as result. This works even for linear types `A` since this way the value is not discarded.

If the part's type `S` is not linear, it can be efficiently accessed using a pointer to it. Gencot supports this with the following two operations:

`getref` return a pointer to a part,

`modref` apply a modification function in-place to a part.

The function `getref` has functionality `T! -> PS!` where `PS` is the mapped type used by Gencot for pointer to `S`. If `S` is an unboxed record or abstract type `#B` then `PS` is the corresponding boxed type `B`. Otherwise `PS` is the type `P_S` generated by Gencot for pointers to values of type `S`. The function cannot be implemented in Cogent and is usually implemented in C using the address operator `&`. The operation is safe since both the structure and the result type are readonly. The shared memory used by both can neither be modified through the structure nor through the resulting pointer.

If the part does not safely exist, the situation is similat to function `get`. However, now the result type `PS` is always linear, so `defaultVal` cannot be used to provide a default value.

The function `modref` has functionality `ModPartFun T PS A O` where `A` and `O` are arbitrary types. It behaves as described in Section 2.7.4. If the part does not safely exist, the possible solutions are the same as for function `modify`.

Note that all part access functions are defined in a way that they never allocate or deallocate memory on the heap. therefore they never need the heap as additional in- and output.

An alternative to the function `modref` would be a pair of functions `ref` and `deref` where `ref` returns the pointer together with the structure converted to a type which marks the part as removed (for a record this corresponds to the type with a field taken), and `deref` converts the type back to normal. Since the structure has the converted type the part cannot be accessed through it as long as the type has not been changed back which is done by `deref`, consuming the pointer. However, it is possible to apply `deref` to another pointer of the same type, causing sharing between the structure and the original pointer. To prevent this it must be proven for the Cogent program that the `deref` operation is always applied to a pointer retrieved by a `ref` operation before any other `deref` is applied to the structure. This implies that for proving the type safety properties an arbitrary complex part of the Cogent program must be taken into account. Therefore Gencot does not support such functions.

To make the other parts of the structure available in the modification operation the `modref` operation could pass the structure to it together with the additional input, with the type converted to a readonly type where the modified part is marked as removed. This is safe because in the modification function the part cannot be accessed through the structure and the structure cannot be modified by inserting another value for the part since it is readonly in the modification operation. Note however, that instead of passing the structure to the

modification operation, all (nonlinear) values required from it can also be retrieved outside of `modref` and passed to it as part of the additional information of type `A`. Therefore Gencot does not support this approach.

2.7.7 Primitive Types

The primitive Cogent types are the numerical types, `Bool` and `String`.

Creating and Disposing Values

Values of primitive types cannot be created and disposed. If instances such as `create[U32]` are used in a program this is not detected as error by the Cogent compiler. However, since Gencot does not provide implementations for such instances the resulting C program will not compile.

Modifying Values

Values of primitive types cannot be modified, they can only be replaced. The type constructor `ChgFun` can be used to define such replacement functions for primitive types. The type constructors for modification functions defined in Section `design-operations-modify` can also be applied to primitive types, however the resulting function type does not have the intended semantics of modification functions. Therefore such types should not be used.

Initializing and Clearing Values

Values of primitive types cannot be initialized or cleared. For primitive types there is no corresponding empty-value type. Like for the modification function types the type constructors `IniFun` and `ClrFun` can be applied to primitive types with useful results, but not with the intended semantics. Therefore they should not be used.

Accessing Parts of Values

Primitive values have no parts, therefore the part access operations are not provided for them.

Although the type `String` has a structure consisting of a sequence of characters, access to characters is not supported by Gencot because there is no known size for `String` values.

2.7.8 Pointer Types

Here we denote as "Gencot pointer types" all Cogent types of the form

```
type P... = { cont: ... }
```

generated by Gencot for several C pointer types (see Section 2.6.6).

Gencot pointer types always point to a value of primitive type, of an abstract type representing an unboxed array, or again a pointer (which may also be a function pointer, or a pointer used to represent an array or a boxed record).

Gencot pointer types do not include the type `P_Void`. For `P_Void` no information about the referenced data structure is available. Therefore Gencot

cannot support any operations for it. Values of this type are fully opaque, they can be passed around but neither created, nor manipulated or disposed.

It is possible for the developer to manually define additional Gencot pointer types by defining types using the same definition schema. Gencot provides the operation support described in the following sections also for such types.

Creating and Disposing Pointers

The empty-value type corresponding to a Gencot pointer type `P...` is the type `P... take cont` which is equivalent to the expansion of the macro call `EVTYPE(P...)` (see Section 2.7.3).

For every Gencot pointer type `P...` used in the Cogent program Gencot automatically generates instances of the functions `create` and `dispose` of the form:

```
create[EVTYPE(P...)]
dispose[EVTYPE(P...)]
```

Modifying Pointers

Since Gencot pointer types are specific Cogent record types, modifications of pointer values can be done with the Cogent take and put functions.

The only modification function applicable to a pointer consists of replacing the referenced value. Such modifications correspond either to initialization and clearing operations or to dereferencing operations, described in the next two sections.

Initializing and Clearing Pointers

Before a pointer returned by `create` can be used it must be initialized by storing a value into the referenced memory region. Dually, before disposing a pointer the memory region may be cleared.

All instances of the functions `initFull/Heap/Simp` and `clearFull/Heap/Simp` described in Section 2.7.5 are available for Gencot pointer types. Functions `initFull` and `clearFull` additionally pass the value referenced by the pointer (wrapped in an unboxed record). Functions `initSimp` and `clearSimp` set the referenced value to its default value, discarding it upon clearing. Functions `initHeap` and `clearHeap` are required, if the referenced value is linear, for allocating or deallocating it on the heap.

Functions `initHeap` and `clearHeap` must also be used if the referenced value is of type `MayNull a` (see Section 2.7.10) since it has linear type. Note that function `initHeap` here actaully does not use the heap since it sets the referenced value to null. However, we assume that this is feasible and do not provide alternative support for it, since it can also be implemented in Cogent using the put operation.

For example a pointer `p` of type `EVTYPE(P_U32)` can be initialized to the value `5` with typechecks by the Cogent expression

```
INIT(Full,P\_U32) (p,#{cont=5})
```

and a pointer `p` of type `PP_U32` can be cleared with typechecks by the Cogent expression

```
CLEAR(Heap,PP\_U32) (p,heap)
```

which clears and disposes the referenced pointer.

Alternatively values of Gencot pointer type can be initialized and cleared using the Cogent put and take operations. Then the code for the initialization example above is

```
p{cont=5}
```

and for the clearing example is

```
let p{cont=h}
and h{cont}
and heap = dispose(h,heap)
in (p,heap)
```

where clearing the value referenced by `h` is omitted since it is of primitive type.

Dereferencing Pointers

If a pointer is seen as a structured value, it has the referenced value as a single part. Then the operations for accessing parts of a structured value can be defined for a Gencot pointer type as follows The operation `getref` corresponds to the identity, the operation `modref` is equivalent to applying the part modification function directly to the pointer. Both are not provided separately for pointers. The other operations all dereference the pointer in some way. Gencot provides in `inlude/gencot/CPointer.cogent` the polymorphic functions

```
getPtr: all(ptr,ref). ptr! -> ref!
setPtr: all(ptr,ref:<D). ModFun ptr ref ()
exchngPtr: all(ptr,ref). ModFun ptr ref ref
modifyPtr: all(ptr,ref,arg,out). ChgPartFun ptr ref arg out
```

Instances of these functions are only defined if `ptr` is a Gencot pointer type and `ref` is the corresponding type of the referenced values.

The function `getPtr` dereferences a readonly pointer and returns the result as readonly. The function `setPtr` replaces the referenced value by its second argument, discarding the old value. The function `exchngPtr` works like `setPtr` but returns the old referenced value as additional result. The function `modifyPtr` applies a change function to the referenced value, replacing or modifying it.

Since a Gencot pointer type corresponds to a pointer which is guaranteed to be not null, the value referenced by the pointer safely exists and the functions need not handle the case where it does not exist.

2.7.9 Function Pointer Types

As described in Section 2.6.5, function pointer types are mapped by Gencot to abstract types of the form `#F...`. They are not covered by Gencot pointer types since they behave differently.

Creating and Disposing Function Pointers

Function pointers cannot be created and disposed. Gencot does not provide instances of `create` and `dispose` for function pointer types.

Modifying Values

Function pointers cannot be modified. As for primitive types the type constructors for modification functions should not be used for them.

Initializing and Clearing Values

Function pointers cannot be initialized or cleared, there is no corresponding empty-value type. Like for the modification function types the type constructors `IniFun` and `ClrFun` can be applied to function pointer types with useful results, but not with the intended semantics. Therefore they should not be used.

Accessing Parts of Values

Function pointers have no parts, therefore the part access operations are not provided for them.

Converting between Functions and Function Pointers

For the values of the abstract type for function pointers in Cogent there are two relevant operations: invoking it as a function and converting a Cogent function to a value of that type. Both are supported by Gencot by providing polymorphic abstract functions in `inlude/gencot/CPointer.cogent` for the task.

The latter operation is supported by the polymorphic abstract function

```
toFunPtr: all(fun, fptr). fun -> fptr
```

where `code` is a Cogent function type and `fptr` is the corresponding function pointer type. Although `fptr` is uniquely determined by `fun` Gencot does not provide support for this constraint. However, Gencot provides implementations of instances only for valid pairs of function and function pointer types.

Invoking a function is supported by the polymorphic abstract function

```
fromFunPtr: all(fptr, fun). fptr -> fun
```

which translates the function pointer to the Cogent function (equivalent to the enumeration value) which then can be invoked in the usual way in Cogent.

The Cogent compiler cannot derive the type `fptr` when instances of these functions are used. It must me xplicitly specified such as in

```
toFunPtr[_,#F_...]
fromFunPtr[#F_...,_]
```

All functions for which a function pointer is accepted or returned by these functions must be known to Cogent so that there is an enumeration constant for it. It is not possible to pass a pointer to an arbitrary C function to Cogent as a parameter or as a field in a record. The function must either be defined in Cogent or it must be defined as an abstract function in Cogent. Moreover, if no function of type `fun` is defined in the Cogent program, the generated C code from an invocation of the result of `fromFunPtr` in Cogent is incomplete, since it invokes the dispatcher function which does not exist.

An alternative approach for invoking a function pointer would be a polymorphic abstract function

```
invkFunPtr: all(ptr,args,res). (ptr, args) -> res
```

where `args` is the type of the single argument of the Cogent function (possibly a tuple) and `res` is the result type of the Cogent function. In its C implementation `invkFunPtr` applies the function pointer to the argument and returns its result. This approach always causes correct C code to be generated by the Cogent compiler. However, since the Isabelle C parser does not support function pointer invocations, no refinement proof can be processed for the resulting C program.

Therefore Gencot does not support this alternative approach.

Hence for example for the C type

```
int *()(int, short)
```

the following function instances are provided

```
toFunPtr[(U32,U16) -> P_U32, #F_XU32XU16X_P_U32]
fromFunPtr[#F_XU32XU16X_P_U32, (U32,U16) -> P_U32]
```

2.7.10 The NULL Pointer

The type safety of Cogent relies on the fact that the pointers representing values of linear types are never `NULL`. If null pointers are used in the C source, there is no immediate translation. The way how to translate C code which uses null pointers in a binary compatible way depends on the way how the null pointers are used.

A null pointer can be used as struct member `f` to mark the corresponding part as "uninitialized". It is set when the struct is created and later replaced by a valid pointer. It remains valid until the struct is disposed. If additionally the struct is used only in places during the "uninitialized state" which are different from those afterwards, the "uninitialized state" can be represented by marking the part `f` as not present in the type used for the struct. Setting the pointer to a non-null value changes the struct type to the normal type used for it in Cogent. In a similar way NULL pointers can be used in array elements and in referenced values while the array or pointer has its empty-value type.

If the field `f` is initialized "on demand", i.e., not at a statical point in the program, or if not all elements of an array are initialized together, this solution is not possible. A null pointer can also be used as an "error" or "escape" value for function parameters or results.

The main problem in both cases is that it must be possible to determine at runtime whether a value is null or not. So simply allocating a "dummy" to get a valid non-null pointer is not sufficient, it must also be possible to recognize the dummy pointer.

One possibility for this is if there is a value referenced by the pointer which never occurs in normal execution, it can be used to mark the pointer as dummy.

Another possibility is to use a single dummy pointer for all values of a specific linear type which can be null and store it in a seperate place for comparing it. However, this cannot be done in Cogent since the dummy pointer would be a shared linear value. Even if implemened in C through abstract functions, to prove memory safety every access to such a value must be guarded with a test for the dummy pointer. Thus it is easier to actually use the null pointer instead of the dummy pointer, in combination with a guard testing for the null pointer

Moreover using a dummy pointer is not binary compatible if the pointer is also accessed in external existing C code where it is set to `NULL` or tested for being `NULL`.

A straightforward approach for working with the null pointer uses polymorphic abstract functions

```
null: all(a). () -> a
ronull: all(a). () -> a!
isNull: all(a). a! -> Bool
```

To dispose the null pointer the function `dispose` must also check for the null pointer and ignore invocations for it. The function `ronull` is required to create null values of readonly types. This cannot be done by applying the bang operator to the result of `null` since then the readonly null value cannot escape the banged context.

However, the polymorphic functions cannot be restricted to specific types `T` by Cogent, therefore all values of all types must be assumed to be null and their use must be guarded by `isNull`.

If Gencot generates monomorphic functions `null_T`, `ronull_T`, `isNull_T` only for linear types or a specific subset of them, this approach is also not safe. The existence of the function `null_T` makes it necessary to guard all accesses to values of type `T` and `T!` with the help of `isNull_T`. The existence of the function `ronull_T` makes it necessary to guard all accesses to values of type `T!`, the values of type `T` need not be guarded since a readonly value can never be made modifyable again. Cogent cannot detect cases where values which may be null are accessed without guard. Therefore Gencot does not automatically define the functions for linear types `T`.

The abstract data type `MayNull`

A safer approach replaces type `T` by an abstract type and uses type `T` only in places where a value has been checked for not being null. The abstract type must be specific for `T` and it must be complemented with abstract functions for generating and testing the null pointer. Gencot provides the following generic abstract data type in `include/gencot/MayNull.cogent`. It is available for all linear Cogent types, not only for types generated by Gencot by mapping a C pointer type.

```
type MayNull a
null:      all(a). () -> MayNull a
roNull:    all(a). () -> (MayNull a)!
mayNull:   all(a). a -> MayNull a
roMayNull: all(a). a! -> (MayNull a)!
notNull:   all(a). MayNull a -> Option a
roNotNull: all(a). (MayNull a)! -> (Option a)!
```

The function `null` returns the null pointer, the function `mayNull` casts a non-null pointer of type `a` to type `MayNull a`. The operations `roNull` and `roMayNull` do the same for readonly pointers.

The type `Option` is used from the Cogent standard library. It is preferred over type `Result` because being null is not interpreted as an error here. The

function `notNull` returns `None` if the argument is null and `Some x` if the argument `x` is not null. The function `roNotNull` does the same for a readonly argument. Since `notNull` and `roNotNull` are the only functions which make the value available as a value of type `a` it is guaranteed by the Cogent type constraints that all accesses to the value are guarded by one of these two functions (if no other abstract functions are introduced which convert from `Maynull a` to `a`).

As usual, the type parameter `a` cannot be restricted by Cogent to linear types. However, Gencot provides instances of the functions only for linear types `a`.

Note that since the type `MayNull a` is not denoted by a single type name it is not possible to construct Cogent mappings of derived types for it, such as a function type with a parameter of type `MayNull a`. This is correct, since the function would be polymorphic. If a Cogent type name is defined for a specific instance of `MayNull`, mappings of derived types for the instance can be constructed using this type name.

Based on the abstract functions the function

```
isNull: all(a). (MayNull a)! -> Bool
isNull mn = roNotNull mn | None -> True | Some x -> False
```

is defined in Cogent for an explicit test for the null pointer.

General Operations

Values of type `MayNull a` cannot be created, disposed, initialized, or cleared. Either they are null, or they are pointers of a valid-value pointer type, for which the type-specific functions for creating, initializing, cearing, and disposing are available.

The type constructors `ModFun`, `ModTypeFun`, `ChgPartFun`, and `ModPartFun` can be applied to `MayNull a` with the usual intended semantics.

Part Access Operations

Conceptually, the type `MayNull a` can be seen as a structured value with the non-null pointer of type `a` being an optional "part". Then the operations for accessing parts of a structured value can be defined for `MayNull a` as follows The operation `get` corresponds to `roNotNull`. The operation `set` cannot be defined, since the type `a` of the "part" is always linear and thus not discardable. The operations `getref` and `modref` cannot be defined, since the "part" is identical to the container and need not be stored somewhere on the heap. All four operations are not provided by Gencot. The other two operations are provided by the polymorphic functions

```
exchngNull: all(a).
   ModFun (MayNull a) a a
modifyNull: all(a,arg).
   ModPartFun (MayNull a) a arg arg
modifyNullDflt: all(a,arg:<D,out:<DSE).
   ModPartFun (MayNull a) a arg out
```

Again, instances of the functions are only defined if `a` is a linear type.

The function `exchngNull (mn, p)` takes as input two pointers, the first of which may be null. If `mn` is not null the result is the pair of `mayNull p` and the non-null pointer corresponding to `mn`.

The functions `modifyNull (mn, (modfun,addinput))` and `modifyNullDflt (mn, (modfun,addinput))` are alternatives for operation `modify` as described in Section 2.7.6. If `mn` is not null it modifies the object referenced by `mn` by applying the part modification function `modfun` of type `ModFun a arg out` to `mn` and returns `mn` and the additional result of type `out`. Note that since here the "part" is identical with the "whole", to modify the whole a modification function must be applied to the part instead of a change function. Therefore the functions have type `ModPartFun` instead of `ChgPartFun`.

The case where the `MayNull a` value is null corresponds to the case where the part does not exist. According to the description in Section 2.7.6 the function `roNotNull` corresponds to function `get` with a variant type as result. Here a more efficient alternative passing the result value immediately is not possible, since in the null case a valid pointer must be passed, which is not available. Therefore the typical C pattern of dereferencing a pointer without testing for null, because we know from the context that it is not null, cannot be transferred directly to Cogent. However, a similar effect can be achieved using type `a` for which it is guaranteed by the Cogent type system that it is a non-null pointer and which is always dereferenced without testing it for null.

Function `exchngNull` behaves in the null case as described in Section 2.7.6, it returns its input `(mn,p)`. Function `modifyNull` corresponds to the third case described in Section 2.7.6. It uses the same type for the additional input and result and returns the input to the part modification function as output when the part modification function is not used. This is the most general solution, therefore it is preferred over the other two. Since it restricts the function to be used as part modification functions, the alternative `modifyNullDflt` is provided which discards the additional argument and returns the default value `defaultVal[out]()` as additional result, however it is restricted in its types `arg` and `out` as usual.

2.7.11 Record Types

As described in Section 2.6.3 C struct types are always mapped to Cogent record types. Additional record types may be introduced manually in the translated Cogent program.

Creating and Disposing Records

The empty-value type corresponding to a record type `R` is the type `R take (..)` where all fields are taken, which may be denoted by `EVTYPE(R)` (see Section 2.7.3).

For every record type `R` used in the Cogent program (not only those generated by mapping a C type) Gencot automatically generates instances of the functions `create` and `dispose` of the form:

```
create[EVTYPE(R)]
dispose[EVTYPE(R)]
```

Modifying Records

The type constructors `ModFun`, `ModTypeFun`, `ChgPartFun`, and `ModPartFun` can be applied to record types with the usual intended semantics.

A specific kind of modification functions for records are functions which put or take a record field. Gencot defines macros in `include/gencot/ModFun.cogent` to generate the type of simple put and take operations for record types.

A macro call of the form

```
PUTFUN(R,f,A,O)
```

expands to the type

```
ModTypeFun (R take f) R A O
```

for a function which puts field `f` in a record of type `R`. A macro call of the form

```
TAKEFUN(R,f,A,O)
```

expands to the type

```
ModTypeFun R (R take f) A O
```

for a function which takes field `f` in a record of type `R` (without returning the taken value).

A macro call of the form

```
PUTFUN<n>(R,(f1,...,fn),f,A,O)
```

expands to the type

```
ModTypeFun (R take (f,f1,...,fn)) (R take (f1,...,fn)) A O
```

for a function which puts field `f` while the fields `fi` are already taken. A macro call of the form

```
TAKEFUN<n>(R,(f1,...,fn),f,A,O)
```

expands to the type

```
ModTypeFun (R take (f1,...,fn)) (R take (f,f1,...,fn)) A O
```

for a function which takes field `f` while the fields `fi` are already taken.

Gencot does not automatically define modification functions for record types, since a record can be modified using the Cogent get and put operations for its fields. Only if a field of unboxed record type is modified in-place through a modification function for its context, the record fields must be manipulated using modification functions for the record. Then such operations must be defined manually.

Initializing and Clearing Records

Before a record returned by `create` can be used it must be initialized by putting values for all fields. Dually, before disposing a record it must be cleared by taking all fields.

All instances of the functions `initFull/Heap/Simp` and `clearFull/Heap/Simp` described in Section 2.7.5 are available for record types. Functions `initFull` and `clearFull` pass a value of the corresponding unboxed record type. Functions `initSimp` and `clearSimp` set all fields to their default value, ignoring them upon clearing. Functions `initHeap` and `clearHeap` are required if the record contains fields of linear type, for allocating or deallocating values for them on the heap.

Functions for initializing and clearing a record `r` of type `R` by treating the fields differently can be manually defined in Cogent by putting or taking values into/from all fields.

If fields of linear type or of unboxed record or abstract type should be initialized or cleared using specific initialization or clearing functions, these functions can either be specified explicitly in the code or they can be passed as additional parameter to the initialization or clearing function.

A field `f` with an unboxed record type `#S` corresponds to an embedded struct in C. The space for this struct is allocated together with the space for `r` by function `create`, so it needs only be initialized. The `initFull` instance initializes it by writing the value in one assignment together with alll other fields. The `initHeap/Simp` instances instead pass a pointer to the embedded struct to corresponding `initHeap/Simp` instances for the part.

When an initialization function for `R` is implemented manually, it should be possible to do the same with an arbitrary initialization function of type `IniFun EVTYPE(S) S arg out` for field `f`. This is the same approach as for the operation `modref` described in Section 2.7.4. However, `modref` takes as argument a value of type `R` where no field is taken, so it cannot be used to put the taken field `f` by initializing it. Separate modification functions are required for the record with taken fields. If the record contains several embedded structs every modification function initializes one field and the result type has one field less taken. Thus, defining such modification functions imposes an order in which the embedded structs must be initialized and whether they are initialized before or after the other fields. A corresponding modification function for field `f` which must be initialized before field `g` but after all other fields would be

```
putFInR : all(arg,out).
  PUTFUN1(R,(g),f,(IniFun EVTYPE(S) S arg out,arg),out)
```

It is invoked with the initialization function for the embedded record and its argument as additional information. In a similar way a field of unboxed record type `#S` can be cleared in-place using an abstract function of the form

```
takeFInR : all(arg,out).
  TAKEFUN1(R,(g),f,(ClrFun S EVTYPE(S) arg out,arg),out)
```

which applies a clearing function of type `ClrFun S EVTYPE(S) arg out` and takes the field `f` while field `g` is already taken. Of course, instead of polymorphic functions for all types `arg` and `out` the specific forms corresponding to `initFull`, `initHeap`, or `initSimp` can be defined.

Another approach to record initialization could have been that the function `create[EVTYPE(R)]` returns a value where only the fields of primitive and linear type are taken and the embedded structs are present, but again with all primitive and linear fields taken. Then the initialization function for field `f` could be directly applied to a pointer to field `f`. However, it is still necessary to retrieve the pointer to `f` with the help of an abstract function which now has to respect the fact that other embedded record fields have types with some fields taken. So the same number of additional abstract functions is needed as in the approach above and their argument types are of similar complexity.

Initializing a field `f` with a function pointer type `#F_...` can always be done using the Cogent put operation. The value to be put must be constructed using the function `toFunPtr` (see Section 2.7.9). To avoid null pointers to functions, as described in Section 4.5.4, a dummy function should be defined in Cogent and passed to the `toFunPtr` function.

Accessing Record Fields

The parts of a record are its fields. The conceptual operations for accessing parts of a structured value can be defined for a record type `R` as follows.

The field to be accessed by the operation must be specified by its name. This can only be done as part of the function name. Therefore Gencot does not define polymorphic functions for accessing arbitrary fields of arbitrary record types. For every record type `R` a set of differently named functions must be defined for every field `f`. Since only some of the functions are needed, Gencot does not automatically generate such functions, they must be defined manually, if required.

The access functions can be defined as polymorphic functions in respect to the record type. Then, for every field name `f` there is a polymorphic function which can be used for all records with a field named `f`. Additionally, the function must be polymorphic in the field type, so that it supports fields of different type. The resulting functions have the following forms

```
getFldF : all(rec,fld). rec! -> fld!
setFldF : all(rec,fld:<D). ModFun rec fld ()
exchngFldF : all(rec,fld). ModFun rec fld fld
modifyFldF : all(rec,fld,arg,out). ChgPartFun rec fld arg out
```

where `rec` is the type of the record and `fld` is the type of the field.

The operation `getFldF` for a field `f` corresponds to the Cogent member access operation `r.f`. Since the `get` operation is not passed as argument to other functions, it need not be defined for record fields, it is always possible to use the Cogent member access operation instead.

The operation `setFldF` for a field `f` and a value `v` corresponds to the Cogent put operation rf = v, if the field has a discardable type. Otherwise it is not supported.

The operation `exchngFldF` for a field `f` and a value `v` can be implemented in Cogent by first taking the old value of `f`, then putting `v` and finally returning the record together with the old value.

The operation `modifyFldF` for a field `f` and a field change function can be implemented in Cogent by first taking the value of `f`, then applying the change function to it, and finally putting the result back into the field.

All three operations may be passed as argument to other modification functions, then they must be defined and cannot be replaced by a direct inline implementation in Cogent.

The operations `getref` and `modref` cannot be implemented in Cogent, they must be defined as abstract function which are implemented in C with the help of the address operator `&`. For a field `f` the corresponding function definitions have the form

```
getrefFldF : all(rec,pfld). rec! -> pfld!
modrefFldF : all(rec,pfld,arg,out). ModPartFun rec pfld arg out
```

where `pfld` is the Gencot mapping of the type of pointer to the field type.

Since whenever an instance of these functions is defined, the field is safely existing, the case of the nonexisting part needs not be respected.

Remember that according to the definition of `ModPartFun` the function `modrefFldF` is invoked in the form

```
(r',o) = modrefFldF(r, (m,a))
```

where `r` is a boxed record value which is not readonly, `m` is a modification function of type `ModFun pfld arg out` which has the form `(pfld,arg) -> (pfld,out)`, `a` is the argument of type `arg` passed to `m`, `r'` is the record value `r` after the modification, and `o` is the additional result of type `out` returned by `m`.

2.7.12 Array Types

Gencot represents arrays of known size in Cogent always by a (boxed) record of a type `An...` (where `n` is the array size) with a single field of (unboxed) abstract type `#UAn...` which in C is defined to be the array type (see Section 2.6.4). We call these types "Gencot array types" here.

Gencot array types are complemented by abstract polymorphic functions for working with arrays. For these functions Gencot automatically generates the instance implementations in C. All these implementations involve the array size in some way. The developer may manually introduce additional Gencot array types by adhering to the Gencot naming schema for array types (see Section 2.6.4). Then Gencot also supports these types and generates instances of all polymorphic functions for them.

If the array size is unknown (types of the form `A_El`) no instances of the polymorphic functions are provided by Gencot. However, implementations may be provided manually. Often this is possible by the developer determining the array size from the context. If several array types with different unknown sizes have been mapped to the same abstract type, they must be disambiguished manually.

Gencot only supports arrays which are allocated on the heap. In C, arrays can also be introduced by defining them as global or local variable, but Cogent has no language constructs which support this.

Creating and Disposing Arrays

Cogent does not provide a language construct to create or dispose values for Gencot array types.

The empty-value type corresponding to type `An...` is the type generated by the macro call `EVTYPE(An...)` as usual. It statically marks the array as uninitialized.

For every Gencot array type `An...` used in the Cogent program Gencot automatically generates instances of the functions `create` and `dispose` of the form:

```
create[EVTYPE(An...)]
dispose[EVTYPE(An...)]
```

Both functions also support multidimensional array types of the form `An1An2...`.

Modifying Arrays

If an array is seen as a structured value, its elements are its parts.

Since the actual array type is always abstract, all accesses to array elements must be performed using abstract functions.

The type constructors `ModFun`, `ModTypeFun`, `ChgPartFun`, and `ModPartFun` can be applied to Gencot array types with the usual intended semantics.

Gencot provides no equivalent for single Cogent take and put operations for array elements. It would be necessary to statically encode the set of array indices for which the elements have been taken in the type expression, this is not feasible.

Initializing and Clearing Arrays

An array initialization function sets a valid value for every element. An array clearing function clears every element, clearing and disposing all linear values contained in elements. Since all elements are of the same type, they can all be treated in the same way by an initialization or clearing function for the element type. Array elements are similar to an embedded struct: the element value is stored in-place in the memory region used for the array. Hence, initializing and clearing an element can also be done in-place by passing a pointer to the element to the element initialization or clearing function. As usual, an additional input given to the array function is passed through to every invocation of the element function.

All instances of the functions `initFull/Heap/Simp` and `clearFull/Heap/Simp` described in Section 2.7.5 are available for Gencot array types. Functions `initFull` and `clearFull` take or return the complete array content as an unboxed value of type `#An...`. Functions `initSimp` and `clearSimp` set all elements to their default value, ignoring them upon clearing. Functions `initHeap` and `clearHeap` are required, if the element type is linear, for allocating or deallocating it on the heap.

For example an array `a` of type `EVTYPE(A16_U32)` can be initialized by setting all elements to `0` (the `defaultVal[U32]()`) with typechecks by the Cogent expression

```
INIT(Simp,A16\_U32) (a,())
```

and an array `a` of pointers to integers with Cogent type `A16P_U32` can be cleared with typechecks by the Cogent expression

```
CLEAR(Heap,A16P\_U32) (a,heap)
```

which will clear and dispose all elements.

Additionally, Gencot provides operations where the user can specify how to initialize or clear a single element, which is then applied to all elements. The specification for a single element is done by passing an initialization or clearing function which can be applied to a pointer to the element. Gencot defines the corresponding polymorphic abstract functions

```
initArrPar: all(evt,vvt,epe,vpe,arg:<S,out).
   IniFun evt vvt (IniFun epe vpe arg out, arg, (out,out)->out) out
clearArrPar: all(vvt,evt,vpe,epe,arg:<S,out).
   ClrFun vvt evt (ClrFun vpe epe arg out, arg, (out,out)->out) out
initArrSimp: all(evt,vvt,epe,vpe,arg:<S).
   IniFun evt vvt (IniFun epe vpe arg (), arg) ()
clearArrSimp: all(vvt,evt,vpe,epe,arg:<S).
   ClrFun vvt evt (ClrFun vpe epe arg (), arg) ()
initArrSeq: all(evt,vvt,epe,vpe,arg).
   IniFun evt vvt (IniFun epe vpe arg arg, arg) arg
clearArrSeq: all(evt,vvt,epe,vpe,arg).
   IniFun evt vvt (IniFun epe vpe arg arg, arg) arg
```

where `epe` is the empty-value type for a pointer to an element and `vpe` is the corresponding valid-value type. The first pair of functions passes the additional input of type `arg` to every invocation of the element function, therefore it must be sharable. The outputs of type `out` are combined using the function passed as third part of the additional input to `initArrPar` or `clearArrPar`. It is applied by "folding", starting at the first element. The second pair uses an element function which does not return a result and needs no combination function. The third pair of functions passes the additional result of every invocation of the element function as additional input to the invocation for the next element. Thus it must have a common type, which may be linear, since it is neither shared nor discarded. In particular, it is possible to use `initHeap` or `clearHeap` as element function (which has exactly the same effect as using `initHeap` or `clearHeap` for the array as a whole).

For every array type `AnEl` (or `An_El`) Gencot provides the instances

```
initArrPar[EVTYPE(AnEl),AnEl,EVTYPE(El),El,A,O]
clearArrPar[AnEl,EVTYPE(AnEl),El,EVTYPE(El),A,O]
initArrSimp[EVTYPE(AnEl),AnEl,EVTYPE(El),El,A]
clearArrSimp[AnEl,EVTYPE(AnEl),El,EVTYPE(El),A]
initArrSeq[EVTYPE(AnEl),AnEl,EVTYPE(El),El,A]
clearArrSeq[AnEl,EVTYPE(AnEl),El,EVTYPE(El),A]
```

for all types `A` and `O`.

In these instances all types besides `A` and `O` are uniquely determined by the array type `AnEl`, although this cannot be expressed by Cogent type constraints. Therefore Gencot defines in `include/gencot/Memory.cogent` the preprocessor macros

```
INITARR(<k>,n,ek,El,A,O)
CLEARARR(<k>,n,ek,El,A,O)
```

which expand to the corresponding instance specifications shown above. Parameter <k> is either Par, Simp or Seq. The additional parameter ek is required for technical reasons and specifies the kind of the element type. It may be empty for a primitive type, F for a function pointer type, R for a boxed record type, U for an unboxed record type, A for a Gencot array type, and P for all Gencot pointer types. If <k> is not Par the last parameter O is ignored and may be empty.

For example an array a of type EVTYPE(A16_U32) can be initialized by setting all elements to 5 with typechecks by the Cogent expression

```
INITARR(Simp,16,,U32,U32,) (a,(INIT(Full,P\_U32),5))
```

It could be useful to also know the element index in the element function. This could be supported by passing the index as additional input to the element functions. However, this implies that the normal initialization and clearing functions for the element type cannot be used as element functions here, since they do not expect the index as additional input. Therefore the functions automatically supported by Gencot for initializing and clearing arrays do not pass the index to the element functions. However, using initArrSeq or clearArrSeq, the index can be calculated by passing it from one invocation of the element function to the next, counting it up in the element function.

Accessing Array Elements

All operations for accessing parts of a structured value are nontrivial for the elements of an array. Since elements are specified by an index value a specified element need not exist, this case must be handled by the part access operations.

Gencot provides polymorphic functions for all six operations with alternatives for the case where the element does not exist:

```
getArr : all(arr,idx,el). (arr!,idx) -> el!
getArrChk : all(arr,idx,el). (arr!,idx) -> Result el! ()
setArr : all(arr,idx,el:<D). ModFun arr (idx,el) ()
exchngArr : all(arr,idx,el). ModFun arr (idx,el) el
modifyArr : all(arr,idx,el,arg).
  ModFun arr (idx, ChgFun el arg arg, arg) arg
modifyArrDflt : all(arr,idx,el,arg:<D,out:<DSE).
  ModFun arr (idx, ChgFun el arg out, arg) out
getrefArr : all(arr,idx,pel). (arr!,idx) -> pel!
getrefArrChk : all(arr,idx,pel). (arr!,idx) -> Result pel! ()
modrefArr : all(arr,idx,pel,arg).
  ModFun arr (idx, ModFun pel arg arg, arg) arg
modrefArrDflt : all(arr,idx,pel,arg:<D,out:<DSE).
  ModFun arr (idx, ModFun pel arg out, arg) out
```

The type variable arr denotes the array type, idx denotes the index type, el denotes the array element type, and pel denotes the type of pointers to elements.

Instances of these functions are only defined if arr is a Gencot array type. Type idx must be one of U8, U16, U32 according to the <size> of the array. Types el and pel must be as determined by arr. arg and out may be arbitrary types only constrained as specified above.

Function `getArr` retrieves the indexed element as readonly. If the element does not exist because the specified index is not in the range `0..<size>-1` the function returns the element with index `0` which always exists since Gencot array types must have atleast one element. Note that this solution is more general than returning `defaultVal` since it works for arbitrary element types. As alternative, as described in Section 2.7.6, the function `getArrChk` returns a variant value using the value `Error()` if the element does not exist. This function should be used whenever it cannot be proven that the index is valid.

Function `setArr`, as usual, is only defined for a discardable element type. It simply discards the old value of the indexed element and sets it to the specified value.

Function `exchngArr` replaces the element at the specified position by the element passed as parameter and returns the old element in the result.

If the specified index is not in the range `0..<size>-1` both functions work as described in Section 2.7.6: `setArr` returns the unmodified array and `exchngArr` returns its unmodified input.

Function `modifyArr` changes the element at the specified position by applying the element change function to it. If the specified index is not in the range `0..<size>-1` the function returns the unmodified array together with the additional input value for the element change function. This corresponds to the third solution described in Section 2.7.6 and requires that the types of additional input and result are the same. Since this may prevent the use of some element change functions an alternative is provided by the function `modifyArrDflt` which discards the additional input and returns the default value `defaultVal[out] ()`. This allows different types for input and result, however, the required constraints apply to these types.

Function `getrefArr` returns a pointer to the element in the array without copying the element. This is safe since both the array and the result type are readonly. If the element does not exist it behaves like `getArr`, returning a pointer to the element with index `0`. The alternative function `getrefArrChk` is provided and returns the corresponding variant value.

Functions `modrefArr` and `modrefArrDflt` work like `modifyArr` and `modifyArrDflt` but use an element modification function and pass a pointer to the element to it, so that the element is modified in-place.

The types of `modifyArr`, `modifyArrDflt`, `modrefArr`, and `modrefArrDflt` are similar to a `ChgPartFun` or `ModPartFun`, but differ because in addition to the element function and its argument the element index must be passed as argument.

Note that it is not possible to pass the array or parts of it as additional information in the parameter of type `arg` of the element functions, since that would always be a second use of the array of linear type. If several elements must be modified together, a specific modification function must be defined and used instead of `modifyArr` or `modrefArr`.

For multidimensional array types of the form `An1An2...` the element type is again an array type and the access functions work accordingly.

2.8 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 2.9.

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 2.9). 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.

2.8.1 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, however, it may be used in several declarators. Therefore Gencot also generates a toplevel type definition for it, with an introduced type name as described in Section 2.1.1.

2.8.2 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 Sectiom 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.

2.8.3 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.

2.8.4 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.

2.8.5 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 2.6.3.

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.

2.8.6 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 2.6.2.

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.

2.9 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 2.6.5. In a C function definition the types for all parameters must be specified in the parameter list, if old-style function definitions are ignored.

2.9.1 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 restrictions 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.

This does not include the mapping of derived types. Type derivation in C is done in declarators which refer a common type specification in a declaration. In Cogent there is no similar concept, every declarator must be translated to a separate declaration. This is not done due to the reasons described above. As result, a translated local declaration may have the form

```
Struct_s1 a, *b, c[5];
```

although the Cogent types for `a, b, c` would be `#Struct_s1`, `Struct_s1`, and `#A5_Struct_s1`, respectively. This translation for the derived types must be done manually.

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.

If the function is variadic an additional last tuple component is added with a variable named `variadicCogentParameters`, mainly to inform the developer that manual action is required.

If the C function body is directly written into the Cogent source file, it cannot be syntax checked by the Cogent compiler. Therefore it is wrapped as a Cogent comment by adding a - sign immediately after the opening brace and before the closing brace.

To yield a correct Cogent function, a dummy result expression is generated. Gencot generates the expression depending on the result type, as described in Section 2.6.

The generated Cogent function definition has the form

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

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

2.9.2 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.

Function Header

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 2.8.1, 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.

Function Body

To preserve comments embedded in the C function body it is also considered as a structured source code part. Its subpart structure corresponds to the syntactic structure of the C AST. Since in the target code only identifiers 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 syntactic structure, i.e., a group must always contain a complete syntactical unit such as a statement, expression or declaration, which is the usual case in C code in practice.

An alternative approach would be to treat all nonempty source code lines as subparts of a function body, resulting in a flat sequence 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.

2.10 Function Parameter Modifications

Gencot uses the information, whether a parameter value may be modified by a function, when it translates the function, as described in Section 2.6.5.

A parameter is modified, when a part of it is changed, which is referenced through a pointer. Only then the modification is shared with the caller and persists after the function call.

2.10.1 Detecting Parameter Modifications

A parameter may be modified by an assignment to a referenced part. Such an assignment is easy to detect, if it directly involves the parameter name. However, the assignment can also be indirect, where the parameter or part of it

is first assigned to a local variable or another structure and then the assignment is applied to this variable or structure. This implies that a full data flow analysis would be necessary to detect all parameter modifications.

Gencot uses a simpler semiautomatic approach. It detects all direct parameter modifications which involve the parameter name automatically. It then generates a JSON description which lists all functions with their parameters and the information about the detected modification cases. The developer has to check this description and manually add additional cases of other parameter modifications. The resulting description is then read by Gencot and used during translation.

Gencot also treats the special cases, where a parameter is discarded (by directly or indirectly invoking the C standard function `free`) or is modified but already returned as the (single) result of the C function. These cases must also be detected by the developer.

Parameter modifications are only relevant if the parameter is a pointer or if a pointer can be reached by the parameter and if at least one such pointer is not declared to have a `const` qualified referenced type. Gencot detects this information from the C type and adds it to the JSON description, so that the developer can easily identify the relevant cases where to look for modifications.

In the case of a variadic function, the number of its parameters cannot be determined from the function definition. Here Gencot uses the invocation with the most arguments to determine the number of parameter descriptions added to the JSON description.

A parameter may be modified by a C function locally, but it can also be modified by passing it or a part of it as parameter to an invoked function which modifies its corresponding parameter. Gencot supports these dependent parameter modifications, by detecting dependencies on parameter modifications by invoked functions and adding them to the JSON description. Again, only dependencies which involve the parameter name are detected automatically, other dependencies must be added by the developer.

When Gencot reads the JSON template it calculates the transitive closure of all dependencies and uses it to determine the parameter modification information. Thus the developer has only to look at every function locally, it is not necessary to take into account the effects of invoked functions.

2.10.2 Required Invocations

To support the incremental translation of single C source files, Gencot determines the parameter modification description for single C source files. It processes all function definitions in the file by analysing their function bodies. However, for an invoked function the definition may not be available, it may be defined externally in another C source file. This situation must be handled manually: the developer has to process additional C source files where the invoked functions are defined, to add their descriptions.

Gencot specifically selects only the relevant invocations, which are required because a parameter modification depends on it. If a parameter is already modified locally, additional dependencies are ignored and the corresponding invocations are not selected. In this way Gencot keeps the JSON descriptions minimal.

An invoked function may also be specified using a function pointer. In this case the possible function bodies cannot be determined from the program. Again, it is left to the developer to determine, whether such invocations may modify their parameters. Gencot supports this by also adding parameter description templates for function pointers defined in the source file, to be filled by the developer.

In contrast to functions, function pointers can also be defined locally in a function[1] (as local variable or as parameter). Gencot automatically adds description templates for all invoked local function pointers.

2.10.3 External Invocations

Finally, a required invocation may be external to the package which is processed by Gencot, such as an invocation of a function in the C standard library. When all remaining required invocations are external in this sense, Gencot can be told to "close" the JSON description. Then it uses the declarations of all required invocations to generate a description template for each of them. Since no bodies are available, the developer must fill in the information for all these external functions (and function pointers).

Invoked function pointers may also be members of a struct or union type. For them no definition exists, they are also handled by Gencot when closing a JSON description: for all required invocations of a struct or union member a description template is added which is identified by the member name and the tag of the struct or union type. Invocations of function pointer members in anonymous struct or union types are ignored by the current Gencot version.

2.10.4 Described Entities

Gencot uses the information about parameter modifications when it maps a C function type or function pointer type to Cogent, as described in Section 2.6.5.

C function types are only used when functions are declared or defined. Function definitions are translated to Cogent for all functions defined in the Cogent compilation unit. Function declarations are translated to Cogent for all functions invoked but not defined in the Cogent compilation unit. In both cases Gencot automatically determines the corresponding definitions or declarations and generates the JSON descriptions for them, to be filled by the developer and read when translating the C code to Cogent.

C function pointer types can be used as type for several entities: for global and local variables, for members of struct and union types, for parameters and the result of declared and defined functions, and as base type or parameter types in derived types.

As described in Section 2.9.1, local variable declarations are not translated by Gencot, hence their types need not be mapped and no JSON description is provided if it is a function pointer type. In all other cases, however, the declaration is translated to Cogent and the type must be mapped. For every C source file (also `.h` files), the current version of Gencot provides JSON descriptions of all function pointer types which are used

[1] The current version of Gencot does not support C extensions for nested function definitions.

- for a global variable defined in the file (including those with internal linkage, see Section 2.1.3),
- for a member of a tagged struct type defined in the file,
- for a parameter of a function defined in the file,
- as element type of an array type used for one of the kinds of entities specified above.

These are the cases where it is easy to create a unique identifier for the function pointer entities, which can be understood by the developer and used to associate the description to the entity when translating it to Cogent. The description is provided independent of whether the entity has parameters of linear type or not. If not, it is not needed by Gencot for translating the entity definition, however it may be needed as required invocation for calculating information for parameters depending on it.

In all other cases, such as a parameter type of another function pointer type, or a member of a tagless struct, Gencot does not provide or use JSON descriptions when mapping the type. For parameters of linear type it always assumes that they may be modified and are not discarded and maps the type to a corresponding Cogent type name.

2.10.5 Function Pointer Type Names

In C, in the definition of an entity of function pointer type, the type may be specified directly as a derived type, or it may be specified by a typedef name. In the latter case, several entities may share a common function pointer type which is specified in the type name definition.

The parameter modification descriptions are needed for all single function pointer entities, since these are used in invocations and may be required for calculating dependencies for parameters of other functions. If several entities use a common typedef name, they must agree on the parameter modification description and the common description must be known to Gencot when translating the type name definition.

The situation can be even more complex if first a typedef name for a function type is defined and then several different typedef names for function pointer types or function pointer array types are derived from it. All these typedef names and the corresponding entities must agree on the parameter modification description.

However, entities using a common typedef name may be defined in different C source files and even in different C compilation units, some of which belong to the Cogent compilation unit and others are external to it, but may be added later. This makes it impossible to guarantee that the descriptions for all entities are available and agree, if an incremental translation to Cogent shall be supported.

For this reason Gencot does not relate entities with a common typedef name and does not test whether their descriptions agree, they are processed completely independently. For translating the type name definition, Gencot uses another parameter modification description, which is also independent from those of the entities. It is the task of the developer to take care that all these descriptions agree. If they do not, this will cause inconsistent uses of the Cogent type to which the common type name is mapped in the generated Cogent code.

Note that, in contrast to the entities, the defined type may directly be a function type. As described in Section 2.6.7 such a type definition is not translated to a definition for a Cogent function type, instead it is translated in the same way as for a function pointer type, mapping to a generated Cogent abstract type name. This makes it possible to translate derived pointer types and pointer array types to Cogent using the mapped type name only, the parameter modification description is only needed for translating the original function type definition. For example the C type name definitions

```
typedef int fun(int,char*);
typedef fun *pfun;
```

where the second parameter may be modified, are translated to the Cogent type definitions

```
type Cogent_fun = #F_XU32XLP_U8X_U32
type Cogent_pfun = #Cogent_fun
```

so that the parameter modification description is only needed for the first type definition translation.

The additional parameter modification descriptions are associated with the common type name of the definition which directly contains the derived function type. For every C source file (also `.h` files) Gencot provides JSON descriptions for all such type name definition in the file, where the defined type is a function type, a function pointer type, or an array type with a function pointer type as element type. The description is provided independent of whether the function type has parameters of linear type or not. If not, the description is not used at all by Gencot, but it may be useful as additional information for the developer.

More complex types involving a derived function type are ignored. As for the corresponding entities Gencot assumes for their translation that all parameters of linear type may be modified.

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`.

Many steps are implemented as Unix filters, 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 predefined scripts for filter combinations.

3.1 Origin Positions

Since the `language-c` parser does not support parsing preprocessor directives and C comments, 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 part 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.

Gencot uses the filter `gencot-reporigs` for removing repeated origin markers from the generated target code, as described in Section 3.5.8.

3.2 Parameter Modification Descriptions

As described in Section 2.10 Gencot uses a parameter modification description in JSON format to be filled in collaboration with the developer to determine which function parameters may be modified during a function call.

3.2.1 Description Structure

This description is structured as follows. It is a list of JSON objects, where each object is an entry which describes a function using the following attributes:

```
{ "f_name" : <string>
, "f_comments" : <string>
, "f_def_loc" : <string>
, "f_num_params" : <int> or <string>
, "f_result" : <string>
   <parameter descriptions>
, "f_num_invocations" : <int>
, "f_invocations" : <list of invocation descriptions>
}
```

The attribute `f_name` specifies a unique identifier for the function, as described in Section 3.2.2. The attribute `f_comments` is not used by Gencot, it

can be used by the developer to add arbitrary textual descriptions to the function entry. The attribute `f_def_loc` specifies the name of the C source file containing the definition of the function or functionn pointer (or its declaration, when the function entry has been generated for closing the JSON description).

The attribute `f_num_params` specifies the number of parameters. In the case of a variadic function or a function with incomplete type (which may be the case if the entry has been generated from a declaration), it is specified as `"variadic"` or `"unknown"`, respectively. The attribute `f_result` specifies the identifier of the parameter which is returned as function result (typically afer modifying it). If the result is not one of the function parameters, the attribute is not present. This attribute is never generated by Gencot, it must be added manually by the developer.

All known parameters of the function are described in the `<parameter descriptions>`. Every parameter description consists of a single attribute where the parameter identifier (see Section 3.2.2) is the attribute name. The value is one of the following strings:

`"nonlinear"` According to its type the parameter is not a pointer and its value does not contain pointers directly or indirectly.

`"readonly"` The parameter is not `"nonlinear"` but according to its type all pointers in the parameter have a `const` qualified referenced type.

`"yes"` The parameter is neither `"nonlinear"` nor `"readonly"` and it is directly modified by the function.

`"discarded"` The parameter is neither `"nonlinear"` nor `"readonly"` and it is directly discarded ("freed") by the function.

`"depends"` The parameter is neither `"nonlinear"` nor `"readonly"`, it is neither modified or discarded directly, but it may be modified by an invoked function.

`"no"` None of the previous cases applies to the parameter.

`"?"` The parameter is neither `"nonlinear"` nor `"readonly"`, but the remaining properties are unconfirmed.

`"?depends"` The parameter is neither `"nonlinear"` nor `"readonly"` and it may be modified by an invoked function, but the remaining properties are unconfirmed.

Gencot only generates parameter descriptions with the values `"nonlinear"`, `"readonly"`, `"yes"`, `"?"`, and `"?depends"`. The first two can be safely determined from the C type. If Gencot finds a direct modification, it sets the description to `"yes"`. Otherwise, if it finds a dependency on an invocation, it sets the description to `"?depends"`. Otherwise it sets it to `"?"`.

The task for the developer is to check all unconfirmed parameter descriptions by inspecting the source code. If a local modification is found, the value must be changed to `"yes"`. Otherwise, if the description was `"?depends"` it must be confirmed by changing it to `"depends"`. Otherwise, if a dependency is found, the value `"?"` must be chaged to `"depends"`, otherwise it must be set to `"no"`.

Discarding a parameter is normally only possible by invoking the function "free" in the C standard library. No other function can directly discard one of its parameters. However, a parameter may be discarded by invoking an external function which indirectly invokes free. The task for the developer is to identify all these cases where an external function (where the entry has been generated during closing the JSON description) discards one of its parameters and set its parameter description to `"discarded"`.

It may be the case that a C function modifies a value referenced by a parameter and returns the parameter as its result, such as the C standard function `memcpy`. In this case the parameter need not be added to the Cogent result tuple, since it already is a part of it. To inform Gencot about this case the developer has to add a `f_result` attribute to the function description. As an example, the description for the C function `memcpy` should be

```
{ "f_name" : "memcpy"
, "f_comments" : ""
, "f_def_loc" : "string.h"
, "f_num_params" :3
, "f_result" :"1-__dest"
, "1-__dest" :"yes"
, "2-__src" :"readonly"
, "3-__n" :"nonlinear"
}
```

since it always returns its first parameter as result.

The attribute `f_num_invocations` specifies the number of function invocations found in the body of the described function. If the same function is invoked several times, it is only counted once. The attribute `f_invocations` specifies a list of JSON objects where each object describes an invocation. If the function entry describes a function pointer or an external function, no body is available, so no invocations can be found and both attributes are omitted. If no parameter depends on any invocation, the second attribute (invocation list) is omitted, only the number of invocations is specified.

An entry in the list of invocations describes an invocation using the following attributes:

```
{ "name" : <string>
, "num_params" : <int> or <string>
  <argument descriptions>
}
```

The attribute `"name"` specifies the function identifier of the invoked function. The attribute `"num_params"` specifies the number of parameters according to the type of the invoked function, in the same way as the attribute `"f_num_params`. If an invoked function has no parameters, no entry for it is added to the invocation list.

The `<argument descriptions>` describe all known arguments of invocations of the function. When Gencot creates an invocation description, it inserts argument descriptions according to the maximal number of arguments found in an invocation of this function. Thus, also for invocations of incompletely defined or variadic functions, an argument description is present for every actual argument used in an invocation.

Every argument description consists of a single attribute where the attribute name is the parameter identifier of the parameter corresponding to the argument. The value is one of the strings `"nonlinear"` or `"readonly"` (according to the type of the parameter of the invoked function), or a list of parameter identifiers.

If the value is `"nonlinear"` or `"readonly"` parameter dependencies on this argument are irrelevant, since the invoked function cannot modify or discard it. Otherwise, the list specifies parameters of the *invoking* function for which the modification or discarding depends on whether the invoked function modifies or discards this argument.

The task for the developer is for all unconfirmed parameter descriptions of the invoking function to check whether there are (additional) dependencies on arguments of invoked functions and add these dependencies to the argument descriptions.

3.2.2 Identifiers for Functions and Parameters

Function Identifiers

In the JSON description unique identifiers are needed for all described functions, so that they can be referenced by invocations.

Functions can always be identified by their name. However, for functions with internal linkage the name is only unique per compilation unit, whereas the JSON description may span several compilation units. Therefore, functions with external linkage are identified by their plain name (which is an unstructured C identifier), functions with internal linkage are identified in the form

```
<source file name> : <function name>
```

If the source file name has extension `".c"` the extension is omitted, otherwise (typically in the case of `".h"`) it is not omitted, resulting in identifiers such as `"app.h:myfunction"`.

As source file name only the base name is used. C packages with source files of the same name in different directories are not supported by the current Gencot version.

Function pointers can be identified by the pointer name. Note that several different functions may be invoked through a function pointer at different times. However, for the analysis of parameter modifications *all* functions invokable through the pointer must be taken into account and together be described. Therefore a single id for the function pointer is sufficient. To inform the developer that it is a function pointer, we prepend a star to the pointer name:

```
* <function pointer name>
```

Gencot also supports parameter modification descriptions for arrays of function pointers. Here all array elements are described together by a single description. To inform the developer that the description is for a function pointer array, the identifier has the form

```
*[] <function pointer name>
```

No other types derived from a function pointer are supported by parameter modification descriptions.

A function pointer (array) may be globally defined, then it may have external or internal linkage. In the latter case we prepend the source file name to make it unique, as described above:

```
<source file name> : <function pointer (array)>
```

A function pointer (array) may also be locally defined in another function. To get a unique identifier we prepend the identifier of the containing function:

```
<function identifier> / <function pointer (array)>
```

The JSON description contains only entries for functions which are defined or invoked. Function invocations can only occur in function definitions which have a body. Therefore local function pointers are only relevant in defined functions which cannot be function pointers themselves. This implies that a local function pointer id is always one-level consisting of a global function id and a function pointer name.

A function pointer (array) may also be a member of a struct or union type. Here we do not differentiate between different instances of this type, we describe the members of all such instances together using the member name as function pointer name in

```
<tag> . <function pointer (array)>
```

where `<tag>` is the tag name of the struct or union type.

Finally, as described in Section 2.10.5, Gencot uses parameter modification descriptions for typedef names where the type involves a derived function type. In this case the id has the form

```
typedef | <function (pointer (array))>
```

where the defined typedef name is used as function pointer name. Here, the defined type may also be a function type, in this case the id has the form `typedef|<typedef name>` omitting the star before the `<typedef name>`. Thus for the type name definition

```
typedef int fun(int,char*);
```

a description with id `typedef|fun` is used whereas for

```
typedef int (*fun)(int,char*);
```

a description with id `typedef|*fun` is used.

There are other cases of function pointers, such as function pointer members in anonymous struct types which are not named by a tag, or function pointers which are parameters of another function pointer type. Gencot does not generate identifiers for them, ignores invocations of such function pointers, and does not use parameter modification descriptions for their mapping.

However, if such function pointers are needed as required invocations in other parameter modification descriptions, it is possible to add them manually. Then it is the task of the developer to identify such invocations, determine whether they are relevant for parameter modification dependencies, create a unique identifier for them and add them manually to the JSON description.

Whenever Gencot reads and processes a JSON parameter modification description it treats all function identifiers as opaque unique strings, hence manually created function identifiers need not conform to any schema.

Parameter Identifiers

In the JSON description unique identifiers are also needed for all described function parameters, so that they can be referenced by argument descriptions. Since these references are always local in a function description, parameter identifiers need only be unique per function. Therefore the C parameter name is usually sufficient as id in the JSON description.

However, the parameter name is not always available: If the function has an incomplete type no parameter names are specified, if the function is variadic, only the names of the non-variadic parameters are specified. Therefore Gencot always uses the position number as parameter id, where the first parameter has position 1. To make the JSON description more readable for the developer, Gencot appends the parameter name whenever it is available. Together, a parameter id is a string with one of the forms

```
<pos>
<pos>-<name>
```

where `<pos>` is the position number and `<name>` is the declared parameter name.

Since all parameter identifiers are strings they can be used as JSON attribute names and since they always start with a digit they can be recognized and do not conflict with other JSON attribute names.

When Gencot reads and processes a JSON parameter modification description it removes the optional name from all parameter identifiers and uses only the position. Hence parameter identifiers are considered equal if they begin with the same position.

3.2.3 Example

The following example illustrates the format of the parameter modification descriptions. It consists of two function descriptions, one for a defined function with internal linkage and one for a function pointer parameter invoked by that function.

```
[
    { "f_name" :"app:somefun"
    , "f_comments" :""
    , "f_def_loc" :"app.c"
    , "f_num_params" :5
    , "1-f_sort" :"nonlinear"
    , "2-desc" :"readonly"
    , "3-input" :"?depends"
    , "4-size" :"nonlinear"
    , "5-result" :"yes"
    , "f_num_invocations" :2
    , "f_invocations" :
        [
            { "name" :"memcpy"
            , "num_params" :3
            , "1-__dest" :[ "3-input" ]
            , "2-__src" :"readonly"
```

```
            , "3-__n" :"nonlinear"
            }
        ,
            { "name" :"app:somefun/*f_sort"
            , "num_params" :3
            , "1-arr" :[ "3-input" ]
            , "2-h" :[]
            , "3-len" :"nonlinear"
            }
        ]
    }
,
    { "f_name" :"app:somefun/*f_sort"
    , "f_comments" :""
    , "f_def_loc" :"app.c"
    , "f_num_params" :3
    , "1-arr" :"?"
    , "2-h" :"?"
    , "3-len" :"nonlinear"
    }
]
```

The description is not closed, since it does not contain an entry for the invocation `memcopy`. This invocation is required, since parameter `input` of `somefun` depends on its first argument which is neither nonlinear nor readonly. If the description of parameter `input` is changed to `"yes"`, the description is closed, since now the invocation `memcopy` is not required anymore.

3.2.4 Reading and Writing Json

Input and output of JSON data is done using the package `Text.Json`. There is another package `Data.Aeson` for Json processing which supports a much more flexible conversion between JSON and Haskell types, but this is not needed here.

The package `Text.Json` uses the type `JSValue` to represent an arbitrary JSON value. It provides the functions `encode` and `decode` to convert between `JSValue` and the JSON string representation. Depending on the kind of actual value, a `JSValue` can be converted from and to corresponding Haskell types using the functions `readJSON` and `showJSON`. A JSON object is converted to the type `JSObject a` where `a` is the type of attribute values, usually this is again `JSValue`. A `JSObject a` can be further processed by converting it from and to an attribute-value list of type `[(String,a)]` using the functions `fromJSobject` and `toJSObject`.

The Gencot parameter modification description is represented as a list of Json objects of type `[JSObject JSValue]`. The same representatiom is used for the contained invocation description lists. For internal processing the objects are converted to attribute-value lists. All processing of JSON data is directly performed on these lists.

A parameter modification description is read by applying the `decode` function to the input to yield a list of `JSObject JSValue`. It is output by applying the `encode` function to such a list.

Additionally, the resulting string representation is formatted using the general prettyprint package `Text.Pretty.Simple`. The function `pStringNoColor` is used since it does not insert control sequences for colored represenation and produces a plainly formatted text representation. Its result is of type `Text` and must be converted to a string using the function `unpack` from package `Data.Text`.

3.2.5 Evaluating a Description

A parameter modification description is complete, if all paremeter descriptions are confirmed and the description is "closed", i.e., has no required invocations. This implies that whenever a parameter is dependent on an invocation argument, there is a function description present for the invoked function where the corresponding parameter is described. This allows to eliminate all parameter dependencies by following them until an independent parameter description is found. This process is called "evaluation" of the parameter modification description.

The result of evaluation is a simplified parameter modification description where the value `"depends"` does not occur anymore as parameter description. Additionally, all information about function invocations is removed, since it is no more needed.

Evaluation only terminates, if there are no cyclic parameter dependencies. This property is checked by Gencot. If there are cyclic parameter dependencies in the C code they must be eliminated manually by the developer by removing enough dependencies to break all cycles.

A parameter may have several dependencies, which may result in different description values. This cannot be the case for the values `"nonlinear"` and `"readonly"`: If the parameter of the invoked function has nonlinear type, this also holds for the passed parameter, no dependency can exist in this case. If the parameter of the invoked function has a readonly type, the passed parameter can still have a linear type which is not readonly. But it cannot be modified by the function invocation, hence no dependency can exist in this case either.

Thus, after evaluation, a parameter may have any subset of the values `"yes"`, `"discarded"`, and `"no"`, meaning that it may be modified by some invocations, discarded by some invocations, and not modified by others. This is reduced to a single value as follows. The value `"no"` is only used if none of the other two is present. If both `"yes"` and `"discarded"` are present, Gencot cannot decide whether the parameter is always discarded (perhaps after modification), or always modified (perhaps after discarding and reallocating it). In this case it always assumes a modification and uses the single value `"yes"`. If this is not correct it must be handled manually by the developer.

The reason for this treatment is that before evaluation Gencot gives a local modification a higher priority than a dependency, to reduce the number of required invocations. This may hide a dependency which results in discarding the parameter. To be consistent, Gencot also prefers modifications resulting from dependencies after evaluation over discarding.

Since in evaluated parameter modification descriptions all parameter descriptions are still confirmed, the only values possible for a parameter description are `"nonlinear"`, `"readonly"`, `"yes"`, `"discarded"`, and `"no"`. This information

is used by Gencot when translating function parameter and result types, as described in Section 2.10.

3.2.6 Haskell Modules

Parameter modification descriptions are implemented in the following three Haskell modules.

Module `Gencot.Json.Parmod` defines the following types for representing parameter modification descriptions:

```
type Parmod = JSObject JSValue
type Parmods = [Parmod]
```

They are used for constructing and processing the JSON representation. Type `Parmod` represents a single function description as a JSON object with arbitrary JSON values. Type `Parmods` represents a full parameter modification description consisting of a sequence of function descriptions.

Module `Gencot.Traversal` defines the type `ParmodMap` as follows:

```
type ParmodMap = Map String [String]
```

It is used to represent the information from an evaluated parameter modification description for applying it. It is a map from function identifiers to the list of parameter description values according to the function's parameters. Additionally, if a function description contains an attribute `f_result` and the specified parameter has description value `"yes"`, its value in the `ParmodMap` is changed to `"result"`.

Module `Gencot.Json.Translate` defines the translation from C source code to parameter modification descriptions in JSON format. The monadic action

```
transGlobals :: [DeclEvent] -> CTrav Parmods
```

translates a sequence of global declarations and definitions (see Section 3.5.3) to a sequence of function descriptions. It translates all definitions and declarations of functions and of objects with function pointer type or function pointer array type. It translates all type name definitions for a function type, a function pointer type, or a function pointer array type. For every definition of a compound struct or union type it translates all members with function pointer type or function pointer array type. All other `DeclEvent`s are ignored.

Module `Gencot.Json.Process` defines functions for reading and processing parameter modification descriptions in JSON format. The main functions are

```
showRemainingPars :: Parmods -> [String]
getRequired :: Parmods -> [String]
filterParmods :: Parmods -> [String] -> Parmods
mergeParmods :: Parmods -> Parmods -> Parmods
sortParmods :: Parmods -> [String] -> Parmods
addParsFromInvokes :: Parmods -> Parmods
evaluateParmods :: Parmods -> Parmods
convertParmods :: Parmods -> ParmodMap
```

The first function retrieves the list of all unconfirmed parameters in the form

```
<function identifier> / <parameter identifier>
```

The function `getRequired` retrieves the function identifiers of all invocations on which at least one parameter depends. The function `filterParmods` restricts a parameter modification description to the descriptions of all function where the function identifier belongs to the string list. The function `mergeParmods` merges two parameter modification description. If a function is described in both, the description with less unconfirmed parameter descriptions is used. If the same number of parameter descriptions are confirmed always the description in the first sequence is selected. The function `sortParmods` sorts the function descriptions in a parameter modification description according to the order specified by a list of function identifiers. The parameter modification description is reduced to the functions specified in the list. If a function occurs in the list more than once, the position of its first occurrence is used for the ordering. The function `evaluateParmods` evaluates a parameter modification description as described in Section 3.2.5. The function `convertParmods` converts the JSON representation to the compact form `ParmodMap` (also interpreting `f_result` attributes) for applying it when generating Cogent code.

3.3 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.

3.3.1 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` statenments but it looks unusual in the Cogent code.

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

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

3.3.2 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); newline; goto nocode

code-inblock If next is

- **end-of-line** collect(`after`); output(`before` + `after`); newline; goto aftercode-inblock
- **block-comment-end** collect(`after`); goto code-afterblock

code-afterblock If next is

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

aftercode-inblock If next is

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

3.3.3 Filter `gencot-mrgcomments`

The filter for merging comments into the target code is implemented as an awk script. It consists of a function `flushbufs`, 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 around a nonempty 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 (i.e., separated by a single line containing only whitespace), the before units associated with the line numbers of all markers with a "+" sign are collected in a buffer. The following nonempty 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 processed by the function `flushbufs`.

Normally, the buffers are output and reset. In the case that the third buffer is empty (there are no after units for the code) the code line is retained in a separate buffer. If in the next group the first buffer is empty (there are no before units for the code) the code lines of both groups are concatenated without a newline in between (also removing the indentation for the second line). This has the effect of completely removing all markers which are not used for inserting comments at their position, thus removing all unnecessary line breaks.

Markers are also completely removed, if a unit exists but contains only whitespace (such as empty lines for formatting). This is done because the target code is already formatted during generation and inserting additional whitespace for formatting is not useful.

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

3.3.4 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 decalaration 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>
```

where <name> is the name of the declared item, and <bline> is the original source line number where the declaration begins.

The filter implementation is described in Section 3.7.6.

Filter `gencot-movcomments`

The filter `gencot-movcomments <file>` processes the output of `gencot-deccomments` as input. For every line as above, it retrieves the before-unit of `<bline>` from `<file>` and stores it in the file `<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 unit to the comment file. A comment file is even written if the comment unit is empty.

Filter `gencot-defcomments`

For inserting the comments before the target code parts generated from a definition, Gencot uses the marker

```
#DEF <name>
```

The marker 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.

3.4 Preprocessor Directives

3.4.1 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-selppconst <file>` selects from the result of `gencot-selpp` all macro definitions which shall be processed as preprocessor defined constant. If `<file>` is specified it must contain the Gencot manual macro list (see Section 2.4.3).

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 the Gencot

directive retainment list (see Section 2.4.1) of regular expressions for directives which shall be retained.

The filter `gencot-unline` is a utility filter. It expects line directives in its input. It removes all included content (detected by the file specification in the line directives) and expands line directives in the content from `<stdin>` to sequences of empty lines.

How the directives are processed depends on the kind of directives (see Section 2.4). Gencot provides the processing filters `gencot-prcppflags <file>`, `gencot-prcppconst` and `gencot-prcppincl <file>`. Conditional directives are not processed, they are inserted without changes. Other macro definitions are processed manually, the result is simply inserted in the target code. Flags are not translated, but they must be selected and Origin markers must be added.

Conditional directives are merged into the target code in a different way, therefore Gencot provides the specific merging filter `gencot-mrgppcond <file>` for them. It also separates the conditional directives from other directives and adds Origin markers. All other directives are merged into the target code by filter `gencot-mrgpp <file>`. Both filters merge the content in `<file>` into the input and write the merged code to the output. `<file>` must contain the directives to be merged, for `gencot-mrgpp` they must have been marked with Origin markers.

Constant definitions must be preprocessed to map constants names in the replacement bodies (see below). This is done by filter `gencot-preppconst`.

Configuration files typically consist of preprocessor flag definitions, where some of them are commented out. Gencot supports translating configuration files by providing two additional filters. The filter `gencot-preconfig` uncomments all commented macro definitions. The filter `gencot-postconfig <file>` re-comments the generated target code, it takes the original configuration file as its argument.

3.4.2 Separating Directives

The Gencot directive retainment list is used to retain some directives in the output of `gencot-rempp`. These directives are still selected by `gencot-selpp` and re-inserted by `gencot-mrgcond`, if they are not suppressed during directive processing. For conditional directives it is intended to always re-insert them. Preprocessor defined constants are never suppressed. Other macros can be suppressed by not specifying a manual macro definition translation.

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 (including line 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.

Copied directives are normalized in the following sense: All whitespace before and after the leading hash sign are removed (for line directives a single

blank is retained after the hash sign). This is done to simplify further directive processing.

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

Filter `gencot-selppconst`

The filter for selecting constant definitions is implemented as an awk script. It is intended to be applied to the output of `gencot-selpp`, hence it expects normalized preprocessor directives in its input.

Roughly, preprocessor constant definitions are macro definitions without macro parameters but with a nonempty replacement body (i.e., not a flag). However, parameterless macros can be used for defining and inserting all other kinds of C code fragments, such as statements or declarations. So, not all parameterless macro definitions can be translated to Cogent value definitions.

Parameterless macro definitions can be recognized syntactically, since for a macro with parameters an opening parenthesis must follow the macro name without separating whitespace.

Parameterless macro definitions to be translated to Cogent value definitions can in general not be recognized without parsing the replacement body as C code, and even that would not be safe since it can consist of arbitrary fragments with other macro calls embedded. Hence Gencot supports the Gencot manual macro list (see Section 2.4.3) for cases which cannot be recognized automatically. Since usually these complex cases are seldom, the Gencot manual macro list consists of a file specifying names of parameterless macros which should *not* be processed as a preprocessor defined constant. The name of this file is passed to the filter `gencot-selppconst` as an additional argument.

The filter selects and transfers all constant definitions to its output without modifications. Line directives are also transferred. All lines belonging to other directives are replaced by empty lines.

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. It is assumed that all quoted include directives have already been processed in an initial step, however, there may also be include directives where the file is specified by a macro call, which should normally not be retained. Therefore system include directives are retained and all other include directives are removed.
- directives which match a regular expression from the Gencot directive retainment list are not removed, they are intended to be interpreted by the language-c preprocessor to suppress information which causes conflicts during code processing or for other reasons.

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. Before a directive is matched with a regular expression, it is normalized by removing all whitespace around the leading hash sign, hence the regular expressions can be written without considering such whitespace. An example file content is

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

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

Filter `gencot-unline`

This filter is implemented as an awk script.

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 subsequent filters.

All content which does not origin in file `<stdin>` is removed together with its line directives.

3.4.3 Processing Directives

Processing Constants Defined as Preprocessor Macros

The replacement body of a preprocessor constant definition may reference the names of other preprocessor constant definitions. They must be replaced by the mapped Cogent name which corresponds to the name of the Cogent constant. In general, that would also require at least a lexical analysis of the replacement bodies to find the preprocessor constant names. Gencot uses a simpler approach and implements this as follows.

A file is generated which contains for every preprocessor constant definition a macro definition which maps the original name to the Cogent name. This file is prepended to the original file where all constant definitions are masked in a way that they are not recognized by the C preprocessor. The resulting file is processed by the C preprocessor, this will substitute all constant names in the replacement bodies by their mapped form. Together, these steps are implemented by the preprocessing filter `gencot-preppconst`.

Since constant names used in replacement bodies may have been defined in included files, the filter `gencot-preppconst` is intended to be applied to the file after all include directives have been expanded, as described for C code in Section 3.5.1. In the input line directives are expected which specify the origin file for all its content. They are transferred to the output.

The output of `gencot-preppconst` is then processed by filter `gencot-prcppconst` to translate the constant definitions to Cogent value definitions.

Filter `gencot-preppconst` The preprocessing filter is implemented as an awk script. It is intended to be applied to the result of `gencot-selppconst`.

The selected macro definitions in the input are processed twice. In the first phase for every macro definition of the form

```
#define CONST1 replacement-body
```

a definition of the following form is generated:

```
#define CONST1 cogent_CONST1
```

where `cogent_CONST1` is the result of mapping the name `CONST1` to a Cogent variable name according to Section 2.1.1.

In the second phase a masked definition of the following form is generated:

```
_#define XCONST1 replacement-body
```

All backslashes marking continuation lines are duplicated, because the C preprocessor removes them.

The sequence of all definitions from the first phase is prepended to the sequence of definitions from the second phase. Then the resulting file is piped through the C preprocessor which applies the definitions from the first phase to the replacement-bodies in the definitions from the second phase.

The definitions from the first phase are preceded by a line directive for the dummy file name `<mappings>` so that their lines do not count to the content of `<stdin>`.

Filter `gencot-prcppconst` The processing filter is implemented as an awk script. It is intended to be applied to the result of `gencot-preppconst`. It contains the masked definitions to be processed, both from the original source file and from all included files. The definitions from included files are not translated to Cogent value definitions, but they are needed to determine the type and value of externally defined constants.

The type and value needed for the Cogent value definition are determined from the replacement body as follows.

If the body is in the definition line and does not contain any whitespace it may be a C literal. First, enclosing parentheses are removed to also recognize replacement bodies of the form `(127)`. Then the following cases are recognized.

- if the body starts with a decimal digit and only contains digits and letters it is assumed to be an integer literal in decimal, octal (leading zero), or hexadecimal (leading `0x`) notation. Its type is determined to be `U8, U16, U32,` or `U64` depending on the integer value.
- if the body starts with a minus sign, followed by an integer literal as above, its type is determined as `U32` and the unsigned value is constructed by calculating the 2-complement.
- if the body is enclosed in single quotes it is assumed to be a character constant. Its type is determined as `U8`.
- if the body is enclosed in double quotes it is assumed to be a string constant, its type is determined as `String`.

- if the body is a single identifier it is resolved using the previous constant definitions. If successful, the type is determined from that definition. As value the identifier is used.

If the body is in the definition line and contains whitespace, it is cecked whether it consists of a sequence of words which are either string literals or constant names which resolves to type string. In this case it is assumed that the value is a sequence of string literals to be concatenated. Its type is determined as `String`, its value is constructed by concatenating all string values.

In all other cases (also for all bodies using continuation lines) Gencot assumes an expression of type int and determines the type as `U32`. As value the unmodified C expression is used.

Cogent value definitions are only generated from constant definitions belonging to the content of `<stdin>` Every generated Cogent value definition is wrapped in `#ORIGIN` markers according to the line offset of the original constant definition in `<stdin>`.

Processing Flags

Processing flags consists only in selecting the flag definitions from all preprocessor directives and adding `#ORIGIN` markers.

A flag definition is a parameterless macro definition with an empty replacement text. The macro name must be followed by optional whitespace and a newline which is not masked by a backslash. Thus every flag definition occupies exactly one line.

The processing filter `gencot-prcppflags` is implemented as an awk script. It is intended to be applied to the result of `gencot-selpp` after applying `gencot-unline`. The latter removes all directives not contained in the current source file and expands line directives.

The processing filter wraps every selected flag definition line in a pair of `#ORIGIN` and `#ENDORIG` markers for the corresponding line number.

The Gencot manual macro list (see Section 2.4.3) also applies to flags. For flags named in this list their definitions are removed. The name of the list file is passed to the filter `gencot-prcppflags` as an additional argument.

Processing Other Preprocessor Macros

All other preprocessor macros must be processed manually by providing a Gencot macro call conversion and a Gencot macro translation (see Section 2.4.3). Each is provided by a manually prepared file.

The Gencot macro call conversion file is passed as additional input to the language-c preprocessor when processing the result of `gencot-rempp`. The preprocessor prepends the file to its main input thus the contained macro definitions are applied to all macro calls for converting them to valid C syntax.

The Gencot macro translation file is passed as <file> argument to a separate execution of `gencot-mrgpp` before the execution of `gencot-mrgppcond`. It must contain `#ORIGIN` and `#ENDORIG` markers which have been manually inserted according to the origin line range of the translated macro definition. Note that, although `gencot-mrgpp` only processes `#ORIGIN` markers, the `#ENDORIG` markers are required for subsequent steps such as inserting conditional directives and comments.

Processing Include Directives

The preprocessing filter `gencot-prcppincl` is implemented as an awk script. It is intended to be applied to the result of `gencot-selpp` after applying `gencot-unline`. The latter removes all directives not contained in the current source file and expands line directives.

The argument file contains the Gencot include omission list (see Section 2.4.4).

3.4.4 Merging Directive Processing Results

When the conditional directives are merged into the target code, the other directives must have already been merged in, since the conditional directives are inserted depending on the content of groups and their positions. Therefore, first all other directives must be merged using the filter `gencot-mrgpp`, then the conditional directives must be merged using the filter `gencot-mrgppcond`.

Filter `gencot-mrgppcond`

The filter for merging the conditional directives into the target code is implemented as an awk script. As argument it takes the name of a file containing the directives to be merged. Since conditional directives need not be processed it is intended that this file contains the output of filter `gencot-selpp`, i.e., all directives selected from the source, processed by `gencot-unline` to remove included input and line directives. `gencot-mrgppcond` selects only the conditional directives and merges them into the filter input, additionally generating origin markers for every merged directive.

The filter input must contain the generated Cogent target code and the other preprocessor directives. All content in the input must have been marked by origin markers.

In its BEGIN rule the filter reads the conditional directives from the argument `<file>` and associates them with their line numbers, building the list of all sections, ordered according to the line number of their first directive (`#if, #ifdef, #ifndef`). Every section is represented by its list of directives.

While processing the input the program maintains a stack of active sections and for every section in the stack the active group. A section is active if some of its directives have been output but not all. The active group is that corresponding to the last directive which has been output for the section, i.e. it is the group which will contain all target code which is currently output.

The filter only uses the `#ORIGIN` markers to position conditional directives. Gencot assumes that for every `#ENDORIG` marker a previous `#ORIGIN` marker exists for the same line number. Since a condition group always contains a sequence of complete lines, the information about the origin lines in the input is fully specified by the `#ORIGIN` markers, the `#ENDORIG` markers are not relevant for placing the conditional directives.

For every `#ORIGIN` marker in the input the following steps are performed:

- if the line belongs to a group of an active section which is after the active group, all directives of the active section are output until the group containing the line is reached, this group is the new active group of the section.

- if for an active section the line does not belong to the active group or any of its following groups, the `#endif` directive for the section is output and the section becomes inactive (is removed from the stack of active sections).
- if the line belongs to a group of a section which is (after the previous step) not active, the section is set active (put on the stack) and all its directives are output until the group containing the line is reached.
- finally the `#ORIGIN` directive and all following lines which are not an `#ORIGIN` marker are output without any changes.

Note that due to the semantics of the `#elif` and `#else` directives, for every group in a section all directives of preceding groups are relevant and must be output when the active group changes, even if this produces empty groups in between.

Due to the nesting structure, when newly active sections are pushed on the stack in the order of the position of their first directive, the stack reflects the section nesting and the sections will be inactivated in the reverse order and can be removed from the stack accordingly.

If an `#ORIGIN` marker indicates, that the next line belongs to a group *before* the active group, the steps described above imply that the current section is ended and restarted from the beginning.

If the line before an inserted conditional directive contains an `#ENDORIG` marker the newline after it belongs to the marker. If the marker is not used to insert a comment it will be removed completely and the conditional directive will be appended to the previous code line. In this case an additional newline is inserted before the conditional directive.

Filter `gencot-mrgpp`

The filter for merging other directives into the target code is implemented as an awk script. It can be used to merge arbitrary code with origin markers into target code with origin markers, if every code part should be merged in exactly once.

The filter only interpretes `#ORIGIN` markers. Whenever an `#ORIGIN` marker is reached in the target code, all content is merged in up to the first `#ORIGIN` marker with a line number not before that in the target code.

If the line before an inserted code contains an `#ENDORIG` marker an additional newline is inserted before the code for the same reason as for conditional directives.

3.4.5 Special Processing for Configuration Files

The filter `gencot-preconfig` simply removes all `//` comment starts before a preprocessor directive. It must be applied before `gencot-remcomments` so that the commented directives are still present. Gencot assumes that commented directives have no continuation lines.

The filter `gencot-postconfig <file>` reads the original file as its argument and determines from it the line numbers of all commented directives. As its input it expects a target code file with origin markers. According to the origin

markers it inserts a -- comment start at the beginning of every line which originated from a line with a commented directive.

Although the Cogent preprocessor does not remove comments (neither C style nor Cogent style), the comment start marker prevents it to process a following directive. The commented target code is later discarded by the Cogent parser.

3.5 Parsing and Processing C Code

Parsing and processing C code in Gencot is always implemented in Haskell, to be able to use an existing C parser. There are at least two choices for a C parser in Haskell:

- the package "language-c" by Benedikt Huber and others,
- the package "language-c-quote" by Geoffrey Mainland and others.

The Cogent compiler uses the package language-c-quote for outputting the generated C code and for parsing the antiquoted C source files. The reason is its support for quasiquotation (embedding C code in Haskell code) and antiquotation (embedding Haskell code in the embedded C code). The antiquotation support is used for parsing the antiquoted C sources.

Gencot performs three tasks related to C code:

- read the original C code to be translated,
- generate antiquoted C code for the function wrapper implementations,
- output normal C code for the C function bodies as placeholder in the generated Cogent function definitions.

The first task is supported by both packages: a C parser reads the source text and creates an internal abstract syntax tree (AST). Every package uses its own data structures for representing the AST. However, the language-c package provides an additional "analysis" module which processes the rather complicated syntax of C declarations and returns a "symbol map" mapping every globally declared identifier to its declaration or definition. Since Gencot generates a single Cogent definition for every single globally declared identifier, this is the ideal starting point for Gencot. For this reason Gencot uses the language-c parser for the first task.

The second task is only supported by the package language-c-quote, therefore it is used by Gencot.

The third task is supported by both packages, since both have a prettyprint function for outputting their AST. Since the function bodies have been read from the input and are output with only minor modifications, it is easiest to use the language-c prettyprinter, since language-c has been used for parsing and the body is already represented by its AST data structures. However, the language-c prettyprinter cannot be extended to generate the ORIGIN markers, therefore the AST is translated to the language-c-quote AST and the corresponding prettyprinter is used for the third task (see Section 3.5.9).

Note that in both packages the main module is named `Language.C`. If both packages are exposed to the ghc Haskell compiler, a package-qualified import

must be used in the Haskell program, which must be enabled by a language pragma:

```
{-# LANGUAGE PackageImports #-}
...
import "language-c" Language.C
```

3.5.1 Including Files

The filter `gencot-include <dirlist> [<filename>]` 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. The optional `<filename>`, if present, must be the name of a file with a list of names of files which should be omitted from being included.

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.

Generating Line Directives

Line directives are inserted into the output 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 `"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.

Multiple Includes

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 interprete ifdef directives, instead, it simply prevents multiple includes for all files independent from their contents, only based on their full file pathnames. To mimic the behavior of cpp, if a file is not include due to repeated include, the corresponding line directives are nevertheless generated in the form

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

Omitted Includes

A special case of multiple include is the recursive include of the main input file. However, since it is read from standard input, its name is not known to `gencot-include`. If it may happen that it is recursively included, the corresponding pathname, as it appears in an include directive, must be added to the list of includes to be omitted.

There are other reasons, why some include files should be omitted. One case is that an include file may or may not exists which is configured through a preprocessor flag. Since `gencot-include` ignores all conditional directives, this would not be detected and an error message would be caused if the file does not exist.

The list of files to be omitted from inclusion is specified in the optional file passed as second argument to `gencot-include`. Every file must be specified on a separate line by its pathname exactly in the form it occurs in a quoted include directive (without quotes). Since system includes and includes where the included file is specified as a macro call are not processed by `gencot-include` they need not be added to the list.

Includes for files listed to be omitted are simply ignored. No line directives are generated for them.

3.5.2 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, 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.

3.5.3 Reading the Input

Parsing

To apply the language-c 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" Language.C (parseC,initPos)
```

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

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

Analysis

Although it is not complete and only processes toplevel declarations (including typedefs), and object definitions, the language-c analysis module is very useful for implementing Gencot translation. Function definition bodies are not covered by analysis, but they are not covered by Gencot either.

The main result of the analysis module is the symbol table. Since at the end of traversing a correct C AST the toplevel scope is reached, the symbol table only contains all globally defined identifiers. From this symbol table a map is created containing all toplevel declarations and object definitions, mapping the identifiers to their 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 symbol table. The map also gives for every defined type name its 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. Additionally, Gencot uses the symbol table built by the analysis module during its own processing to access the types of globally defined identifiers and for managing local declarations when traversing function bodies, as described in Section 3.5.10.

Additionally, the analysis module provides a callback handler which is invoked for every declaration entered into the symbol table (with the exception of tag forward declarations and enumerator declarations). The callback handler can accumulate results in a user state which can be retrieved after analysis together with the semantics map. Since the callback handler is also invoked for all local declarations it is useful when all declarations shall be processed in some form.

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

```
(table,state) <- either (error . show) return $
  runTrav uinit (withExtDeclHandler (analyseAST ast >> getDefTable) uhandler)
```

which binds the resulting symbol table to variable `table` and the resulting state to `ustate`. `runTrav` returns a result of type `Either [CError] (DefTable, TravState s)`, where `DefTable` is the type of the symbol table and `s` is the type of the user state. The error list in the first alternative contains fatal errors which made the analysis fail. The state in the second alternative contains warnings about semantic inconsistencies, such as unknown identifiers, and it contains the user state. `uinit` is the initial user state and `uhandler` is the callback handler of type

```
DeclEvent -> Trav s ()
```

It returns a monadic action without result.

The semantics map is created from the symbol table by the function `globalDefs`, its type is `GlobalDecls`.

On this basis, Gencot implements the following functions in the module `Gencot.Input` as utility for parsing and analysis:

```
readFromInput :: s -> (DeclEvent -> Trav s ()) -> IO (DefTable, s)
readFromFile :: FilePath -> s -> (DeclEvent -> Trav s ()) -> IO (DefTable, s)
```

The first one takes as arguments an initial user state and a callback handler. It reads C code from standard input, parses and analyses it and returns the symbol table and the user state accumulated by the callback handler. The second function takes a file name as additional argument and does the same reading from the file.

All Gencot filters which read C code use one of these two functions.

Source Code Origin

The language-c parser adds information about the source code origin to the AST. For every syntactic construct represented in the AST it includes the start origin of the first input token and the start origin and length of the last input token. The start origin of a token is represented by the type `Position` and includes the original source file name and line number, affected by line directives if present in the input. It also includes the absolute character offset in the input stream. The latter can be used to determine the ordering of constructs which have been placed in the same line. The type `Position` is declared as instance of class `ORD` by comparing the character offset, hence it can easily be used for comparing and sorting.

The origin information about the first and last token is contained in the type `NodeInfo`. All types for representing a syntactic construct in the AST are parameterized with a type parameter. In the actual AST types this parameter is always substituted by the type `NodeInfo`.

The analysis module carries the origin information over to its results, by including a `NodeInfo` in most of its result structures. This information can be used to

- determine the origin file for a declared identifier,
- filter declarations according to the source file containing them,
- sort declarations according to the position of their first token in the source,
- translate identifiers to file specific names to avoid conflicts.

For the last case the true name of the processed file is required, however, the parsed input is read from a pipe where the name is always given as `<stdin>`. The true name is passed to the Haskell program as an additional argument, as described in Section 3.7.1. Since there is no easy way to replace the file name in all `NodeInfo` values in the semantic map, Gencot adds the name to the monadic state used for processing (see Section 3.5.10).

Preparing for Processing

The main task for Gencot is to translate all declarations or definitions which are contained in a single source file, where nested declarations are translated to a sequence of toplevel Cogent definitions. This is achieved by parsing and analysing the content of the file and all included files, filtering the resulting set of declarations according to the source file name `<stdin>`, removing all declarations which are not translated to Cogent, and sorting the remaining ones in a list. Translating every list entry to Cogent yields the resulting Cogent definitions in the correct ordering.

For syntactically nested constructs in C the analysis phase creates separate declarations. This corresponds to the Cogent form where every declaration becomes a seperate toplevel construct. However, for generating the origin information a seperate processing of these declarations would yield repeated origin ranges which may result in repeated comment units in the target code. Therefore Gencot processes nested constructs as part of the containing constructs. Sorting the declarations according to their positions always puts the nested

declarations after their containing declarations. Processing them as part of the containing declaration will always be done before the nested declaration occurs in the main list of declarations. By maintaining a list of declarations already processed as nested, Gencot skips these declarations when it finds them in the main list.

The only C constructs which can be nested are tag definitions for struct, union, and enum types (all other cases of nesting occurs only in C function bodies where the nested parts are not contained as seperate entries in the main list). Tag definitions can occur as part of every type specification, the most important case is the occurrence in a typedef as in

```
typedef struct s { ... } t
```

Other relevant cases are the occurrence in the declaration of a global variable, in the result or parameter types of a function declaration and in the declaration of a struct or union member (which may result in arbitrarily deep nesting).

For these cases, if the type references a tag definition, Gencot inspects the position information of the tag definition. If it is syntactically embedded, it processes it and marks it as processed so that it is skipped when it finds it in the main list.

The type `GlobalDecls` consists of three separate maps, one for tag definitions, one for type definitions, and one for all other declarations and definitions. Every map uses its own type for its range values, however, there is the wrapper type `DeclEvent` which has a variant for each of them.

The language-c analysis module provides a filtering function for its resulting map of type `GlobalDecls`. The filter predicate is defined for values of type `DeclEvent`. If the map has been bound to the variable `gmap` it can be filtered by

```
filterGlobalDecls globalsFilter gmap
```

where `globalsFilter` is the filter predicate.

Gencot uses a filter which reduces the declarations to those contained directly in the input file, removing all content from included files. Since the input file is always associated with the name `<stdin>` in the `NodeInfo` values, a corresponding filter function is

```
(maybe False ((==) "<stdin>") . fileOfNode)
```

Additionally, for a specific Gencot component, the declarations are reduced to those which are processed by the component.

Every map range value, and hence every `DeclEvent` value contains the identifier which is mapped to it, hence the full information required for translating the definitions is contained in the range values. Gencot wraps every range value as a `DeclEvent`, and puts them in a common list for all three maps. This is done by the function

```
listGlobals :: GlobalDecls -> [DeclEvent]
```

Finally, the declarations in the list are sorted according to the offset position of their first tokens, using the compare function

```
compEvent :: DeclEvent -> DeclEvent -> Ordering
compEvent ci1 ci2 = compare (posOf ci1) (posOf ci2)
```

Together, the list for processing the code is prepared from the symbol table `table` by

```
sortBy compEvent $ listGlobals $ filterGlobalDecls globalsFilter $ globalDefs table
```

All this preprocessing is implemented in module `Gencot.Input`. It provides the function

```
getDeclEvents :: GlobalDecls -> (DeclEvent -> Bool) -> [DeclEvent]
```

It performs the preprocessing and returns the list of `DeclEvent`s to be processed. As its second argument it expects a predicate for filtering the content of `<stdin>` to the `DeclEvent`s to be processed by the specific Gencot component.

3.5.4 Reading Packages

In some cases several source files of the <package> must be processed together. The typical case is when the main files for the Cogent compilation unit are generated (see Section 2.1.3). For this it is necessary to determine and process the external name references in a set of C source files. This set is the subset of C sources in the <package> which is translated to Cogent and together yields the Cogent compilation unit.

General Approach

There are different possible approaches how to read and process this set of source files.

The first approach is to use a single file which includes all files in the set. This file is processed as usual by `gencot-include`, `gencot-remcomments`, and `gencot-rempp` which yields the union of all definitions and declarations in all files in the set as input to the language-c parser. However, this input may contain conflicting definitions. For an identifier with internal linkage different definitions may be present in different source files. Also for identifiers with no linkage different definitions may be present, if, e.g., different `.c` files define a type with the same name. The language-c parser ignores duplicate definitions for identifiers with internal linkage, however, it treats duplicate definitions for identifiers without linkage as a fatal error. Hence Gencot does not use this approach.

The second approach ist to process every file in the set separately and merge the generated target code. However, for identifiers with external linkage (function definitions) the external references cannot be determined from the content of a single file. A non-local reference is only external if it is not defined in any of the files in the set. It would be possible to determine these external references in a separate processing step and using the result as additional input for the main processing step. Since this means to additionally implement reading and writing a list of external references, Gencot does not use this approach.

The third approach is to parse and analyse the content of every file separately, then merge the resulting semantic maps discarding any duplicate definitions. This approach assumes that the external name references, which are relevant for processing, are uniquely defined in all source files. If this is not the case, because conflicting definitions are used inside the <package>, which are external to the processed file subset, this must be handled manually. This approach is used by Gencot.

Specifying Input Files

Due to the approach used, the Gencot filters for generating the files common to the Cogent compilation unit expect as input a list of names of the files which comprise the Cogent compilation unit. The file names must be pathnames which are either absolute or relative to the current directory. Every file name must occur on a single line.

Like all other input to the language-c parser their content must have been processed by `gencot-include`, `gencot-remcomments`, and `gencot-rempp`. This implies that all included content is already present and need not be specified separately, usually only `.c` files need to be specified as input, after they have been processed as usual.

In contrast to the single-file filters, the original file name is not required for the files input to a Gencot processor. A processor only processes items which are external to all input files, whereas the original file name is needed for items which are defined in the input files. Therefore the list of input file names is sufficient and the input files may have arbitrary names which need not be related to the names of the original `.c` files.

The utility function

```
readPackageFromInput :: IO [DefTable]
```

in module `Gencot.Package` reads the file name list from input and parses and analyses all files using `readFromFile`. It returns the list of resulting symbol tables.

Combining Parser Results

When the parser results are combined it is relevant, how they are structured, in particular, if the same file is included by several of the `.c` files. Most of the information only depends on the parsed text. However, the language-c parser also uses unique identifiers, which are counted integers starting at 1 for every parsed file. This implies, that these identifiers are not unique anymore, if several files are parsed separately and then the results are combined.

The unique identifiers are associated with most of the AST nodes and are part of the `NodeInfo` values. In the analysis phase they are used to cache the relation between defining and referencing occurrences of C identifiers, and the relation between C expressions and their types. The corresponding caches are part of the symbol table structure. However, the first relation cache seems to be built but not used, the second relation cache is only used during type analysis for expressions. In both cases, after the analysis phase the caches are still present, but are not relevant for the further processing by Gencot.

As a consequence, it is not possible to combine the raw AST structures and then perform the language-c analysis on the combined AST, since then the non-unique identifiers may cause problems. Instead, Gencot parses and analyses every file separately and the combines the resulting symbol tables.

However, the unique identifiers are additionally used for identifying tagless struct/union/enum types in the symbol table. Language-c uses the alternative type `SUERef` to identify struct/union/enum types, with the alternatives `NamedRef` and `AnonymousRef`, where the latter specifies the unique integer identifier of the AST node of the type definition. The symbol table maps `SUERef`

values to their type definitions. This implies, that tagless types may be entered in different symbol tables under different keys. This must be detected and handled when combining the symbol tables.

As described in Section 3.5.3, Gencot uses for its processing the list of `DeclEvent`s which is derived from the `GlobalDecls` map. This means, the combination could be implemented on the `GlobalDecls` maps or even on the `DeclEvent` lists. However, as described in Section 3.5.10, Gencot also uses the symbol table during processing, for looking up identifiers. For this reason the combination is implemented on the symbol tables.

A language-c symbol table is implemented by the type

```
data DefTable = DefTable {
  identDecls   :: NameSpaceMap Ident IdentEntry,
  tagDecls     :: NameSpaceMap SUERef TagEntry,
  labelDefs    :: NameSpaceMap Ident Ident,
  memberDecls  :: NameSpaceMap Ident MemberDecl,
  refTable     :: IntMap Name,
  typeTable    :: IntMap Type
}
```

where a `NameSpaceMap k v` is a mapping from `k` to `v` with nested scopes. The last two components are the relation caches as described above, they are ignored and combined to be empty. After the analysis phase, on the toplevel, the maps `labelDefs` and `memberDecls` are empty, since in C there are no global labels and `memberDecls` contains the struct/union members only while processing the corresponding declaration. So only the first two maps must be combined.

The map `identDecls` contains all identifiers with file scope. For them, the linkage is relevant. If an identifier has internal linkage, it is only valid in its symbol table and my denote a different object in another symbol table. These identifiers are not relevant for Gencot when processing several source files together, thus they are omitted when the symbol tables are combined. Only identifiers with external or no linkage are retained in the combined symbol table. Note that this implies that identifiers with internal linkage cannot be looked up in the combined table anymore.

If an identifier has external linkage, it is assumed to denote the same object in every symbol table. However, it may be declared in one symbol table and defined in another one. In this case, always the definition is used for the combined table, the declaration is ignored. Only if it is declared in both symbol tables one of the declarations is put into the combined table. Note that if the C program is correct, the identifier may be defined in at most one C compilation unit and thus a dfinition for it occurs in at most one of the symbol tables.

The typical cases for toplevel identifiers with no linkage are typedef names and struct/union/enum tags. Such an identifier may occur in two symbol tables, if it is defined in a file included by both corresponding `.c` files. In this case it names the same type and one of both entries is put in the combined table. However, it may also be the case that the identifier is defined in both `.c` files and used for different types. In this case the combination approach does not work. Gencot assumes that this case does not occur and tests whether the identifier is mapped to the same semantics (determined from the position information of the corresponding definition). If not, an error is signaled, this must be handled manually by the developer.

The map `tagDecls` contains all tags of struct/union/enum types, mapped to the type definition. For tagless types the unique identifier is used as key, as described above. If a tag is present, it is treated in the same way as other identifiers with no linkage.

A tagless struct/union/enum type in C can be referenced only from a single place, since it must be syntactically embedded there. This means, when the same tagless type occurs in two symbol tables using a different key, every symbol table contains at most one reference to the key. When the symbol tables are combined, at most one of these references are transferred to the combined table. Thus it is possible to use the same key for the transferred definition to yield a consistent combined table. The simples way to do so would be to always use the entry of the same symbol table for the combination when an object occurs in both. However, this is not possible, since for identifiers with external linkage the definition must be preferred over a declaration, independent where it occurs. Therefore Gencot first transfers both definitions to the combined table and afterwards removes all definitions which are not referenced there.

Finally, it may happen that the same unique identifier is used in different symbol tables to reference different types, as described for type names above. This is not a problem in the C sources, it is an internal collision of the parser-generated unique identifiers. The easiest way to solve this is to introduce a tag for at least one of both tagless types.

The combination is implemented by the function

```
combineTables :: DefTable -> DefTable -> DefTable
```

in module `Gencot.Package`. It should be applied to the tables built by `readFromFile` for two different `.c` files of the same package. It can be iterated to combine the result of `readPackageFromInput`. The result can then be processed mainly in the same way as described in Section 3.5.3 for a table built from a single input file.

3.5.5 Dummy Declarations for Preprocessor Macros

As described in Section 2.4.3 macro calls in C code must either be syntactically correct C code or they must be converted to syntactically correct C code. Due to the language-c analysis step this is not sufficient. The analysis step checks for additional properties. In particular, it requires that every identifier is either declared or defined.

Thus for every identifier which is part of a converted macro call a corresponding declaration must be added to the C code. They are called "dummy declarations" since they are only used for making the analysis step happy.

For all preprocessor defined constants Gencot automatically generates the required dummy declarations. The corresponding macro calls always have the form of a single identifier occurring at positions where a C expression is expected. The type of the identifier is irrelevant, hence Gencot always uses type `int` for the dummy declarations. For every preprocessor constant definition of the form

```
#define NNN XXX
```

a dummy declaration of the form

```
int NNN;
```

is generated. This is implemented by the additional filter `gencot-gendummydecls`. It is applied to the result of `gencot-selppconst`. The resulting dummy declarations are prepended to the input of the language-c preprocessor since this prevents the lines from being counted for the `<stdin>` part.

Flag macro calls do not occur in C code, hence no dummy declarations are required for them.

For all other macros the required dummy declarations must be created manually and added to the Gencot macro call conversion. Even if no macro call conversion is needed because the macro calls are already in C syntax, it may be necessary to add dummy declarations to satisfy the requirements of the language-c analysis step.

3.5.6 Generating Cogent Code

When Gencot generates its Cogent target code it uses the data structures defined by the Cogent compiler for representing its AST after parsing Cogent code. The motivation to do so is twofold. First, the AST omits details such as using code layout and parentheses for correct code structure and the Cogent compiler provides a prettyprint function for its AST which cares about these details. Hence, it is much easier to generate the AST and use the prettyprinter for output, instead of generating the final Cogent program text. Second, by using the Cogent AST the generated Cogent code is guaranteed to be syntactically correct and current for the Cogent language version of the used compiler version. Whenever the Cogent language syntax is changed in a newer version, this will be detected when Gencot is linked to the newer compiler version.

Cogent Surface Syntax Tree

The data structures for the Cogent surface syntax AST are defined in the module Cogent.Surface. It defines parameterized types for the main Cogent syntax constructs (`TopLevel`, `Alt`, `Type`, `Polytype`, `Pattern`, `IrrefutablePattern`, `Expr`, and `Binding`), where the type parameters determine the types of the sub-structures. Hence the AST types can easily be extended by wrapping the existing types in own extensions which are then also used as actual type parameters.

Cogent itself defines two such wrapper type families: The basic unextended types `RawXXX` and the types `LocXXX` where every construct is extended by a representation of its source location.

All parameterized types for syntax constructs and the `RawXXX` and `LocXXX` types are defined as instances of class `Pretty` from module `Text.PrettyPrint.ANSI.Leijen`. This prettyprinter functionality is used by the Cogent compiler for outputting the parsed Cogent source code after some processing steps, if requested by the user.

As source location representation in the `LocXXX` types Cogent uses the type `SourcePos` from Module `Text.Parsec.Pos` in package `parsec`. It contains a file name and a row and column number. This information is ignored by the prettyprinter.

Extending the Cogent Surface Syntax

Gencot needs to extend the Cogent surface syntax for its generated code in two ways:

- origin markers must be supported, as described in Section 3.1,
- C function bodies must be supported in Cogent function definitions, as described in Section 2.9.1.

Origin Markers The origin markers are used to optionally surround the generated target code parts, which may be arbitrary syntactic constructs or groups of them. Hence it would be necessary to massively extend the Cogent surface syntax, if they are added as explicit syntactic constructs. Instead, Gencot optionally adds the information about the range of source lines to the syntactic constructs in the AST and generates the actual origin markers when the AST is output.

Although the `LocXXX` types already support a source position in every syntactic construct, it cannot be used by Gencot, since it represents only a single position instead of a line range. Gencot uses the `NodeInfo` values, since they represent a line range and they are already present in the C source code AST, as described in Section 3.5.3. Hence, they can simply be transferred from the source code part to the corresponding target code part. For the case that there is no source code part in the input file (such as for code generated for external name references), or there is no position information available for the source code part, the `NodeInfo` is optional.

It may be the case that a target AST node is generated from a source code part which is not a single source AST node. Then there is no single `NodeInfo` to represent the origin markers for the target AST node. Instead, Gencot uses the `NodeInfo` values of the first and last AST nodes in the source code part.

It may also be the case that a structured source code part is translated to a sequence of sub-part translations without target code for the main part. In this case the `#ORIGIN` marker for the main part must be added before the `#ORIGIN` marker of the first target code part and the `#ENDORIG` marker for the main part must be added after the `#ENDORIG` marker of the last target code part.

To represent all these cases, the origin information for a construct in the target AST consists of two lists of `NodeInfo` values. The first list represents the sequence of `#ORIGIN` markers to be inserted before the construct, here only the start line numbers in the `NodeInfo` values are used. The second list represents the sequence of `#ENDORIG` markers to be inserted after the construct, here only the end line numbers in the `NodeInfo` values are used. If no marker of one of the kinds shall be present, the corresponding list is empty.

Additional information must be added to represent the marker extensions for placing the comments (the trailing "+" signs). Therefore, a boolean value is added to all list elements.

Together, Gencot defines the type `Origin` for representing the origin information, with the value `noOrigin` for the case that no markers will be generated:

```
data Origin = Origin {
  sOfOrig :: [(NodeInfo,Bool)],
  eOfOrig :: [(NodeInfo,Bool)] }
noOrigin = Origin [] []
```

Gencot adds an `Origin` value to every Cogent AST element. The type `Origin` is defined in the module `Gencot.Origin`

Embedded C Code Cogent function definitions are represented by the `FunDef` alternative of the type for toplevel syntactic constructs:

```
data TopLevel t p e =
  ... | FunDef VarName (Polytype t) [Alt p e] | ...
```

The type parameter `e` for representing syntactic expressions is only used in this alternative and in the alternative for constant definitions. Cogent constant definitions are generated by Gencot only from C enum constants (preprocessor constants are processed by `gencot-prcconst` which is not implemented in Haskell). The defined value for a C enum constant is represented in the C AST by the type for expressions. Together, instead of Cogent expressions, Gencot always uses either a C expression or a Cogent expression together with a C function body (which syntactically is a statement) in the Cogent AST.

To modify the Cogent syntax in this way, Gencot defines an own expression type with two alternatives for a C expression and for a Cogent expression together with a C statement:

```
data GenExpr = ConstExpr Exp
             | FunBody RawExpr Stm
```

where `Exp` and `Stm` are the types for C expressions and statements as defined by the language-c-quote AST (see Section 3.5.9). Note that no `Origin` components are added, since the types `Exp` and `Stm` already contain `Origin` information. The type `RawExpr` is used for the dummy result expression. It has no origin in the C source, therefore the raw type without origin information is sufficient.

Other than for the dummy result expression, the Cogent AST expression type is not used by Gencot. Since bindings only occur in expressions, the AST type for Cogent bindings is not used either.

For the type parameters `t` and `p` for representing types and patterns, respectively, the normal types for the Cogent constructs are used, since Gencot generates both in Cogent syntax. The pattern generated for a function definition is always a tuple pattern, which is irrefutable. Gencot never generates other patterns, hence the AST type for irrefutable patterns is sufficient.

Together, Gencot uses the following types to represent its extended Cogent surface AST:

```
data GenToplv =
  GenToplv Origin (TopLevel GenType GenIrrefPatn GenExpr)
data GenAlt =
  GenAlt Origin (Alt GenIrrefPatn GenExpr)
data GenIrrefPatn =
  GenIrrefPatn Origin (IrrefutablePattern VarName GenIrrefPatn)
data GenType =
  GenType Origin (Type GenExpr GenType)
data GenPolytype =
  GenPolytype Origin (Polytype GenType)
```

The first parameter of `Type` for expressions is only used for Cogent array types, which are currently not generated by Gencot.

All five wrapper types are defined as instances of class `Pretty`, basically by applying the Cogent prettyprint functionality to the wrapped Cogent AST type.

3.5.7 Mapping Names

Names used in the target code are either mapped from a C identifier or introduced, as described in Section 2.1.1. Different schemas are used depending on the kind of name to be generated. The schemas require different information as input.

General Name Mapping

The general mapping scheme is applied whenever a Cogent name is generated from an existing C identifier. Its purpose is to adjust the case, if necessary and to avoid conflicts between the Cogent name and the C identifier.

As input this scheme only needs the C identifier and the required case for the Cogent name. It is implemented by the function

```
mapName :: Bool -> Ident -> String
```

where the first argument specifies whether the name must be uppercase.

Cogent Type Names

A Cogent type name (including the names of primitive types) may be generated as translation of a C primitive type, a C typedef name, a C struct/union/enum type reference, or a C derived type.

A C primitive type is translated according to the description in Section 2.6. Only the type specifiers for the C type are required for that.

A C typedef name is translated by simply mapping it with the help of `mapName` to an uppercase name. Only the C typedef name is required for that.

A C struct/union/enum type reference may be tagged or tagless. If it is tagged, the Cogent type name is constructed from the tag as described in Section 2.1.1: the tag is mapped with the help of `mapName` to an uppercase name, then a prefix `Struct_`, `Union_` or `Enum_` is prepended. For this mapping the tag and the kind (struct/union/enum) are required. Both are contained in the language-c type `TypeName` which is used to represent a reference to a struct/union/enum.

If the reference is untagged, Gencot nevertheless generates a type name, as motivated and described in Section 2.1.1. As input it needs the kind and the position of the struct/union/enum definition. The latter is not contained in the `TypeName`, it contains the position of the reference itself. To access the position of the definition, the definition must be retrieved from the symbol table in the monadic state. Hence, the mapping function is defined as a monadic action.

Together the function for translating struct/union/enum type references is

```
transTagName :: TypeName -> FTrav String
```

Since an untagged struct/union/enum can be contained in any type specification and type specifications may occur in all other C constructs, the `GlobalDecls`

map must be passed as argument to all translation functions from C constructs to Cogent constructs.

If the definition itself is translated, it is already available and need not be retrieved from the map. However, as described in Section 3.5.6, the map may be needed to map the generic name `<stdin>` to the true source file name. Therefore Gencot uses function `transTagName` also when translating the definition.

A C derived type is translated to a Cogent type name by translating the name of the basic type as described above, and then prepending the encoded sequence of derivation steps, as described in Section 2.1.1. The information about the derivation steps is contained in the type construct, no information in addition to that required for translating the basic type name is needed.

Cogent Function Names

Cogent function names are generated from C function names. A C function may have external or internal linkage, according to the linkage the Cogent name is constructed either as a global name or as a name specific to the file where the function is defined. For deciding which variant to use for a function name reference, its linkage must be determined. It is available in the definition or in a declaration for the function name, either of which must be present in the symbol table. The language-c analysis module replaces all declarations in the tyble by the definition, if that is present in the parsed input, otherwise it retains a declaration.

A global function name is generated by mapping the C function name with the help of `mapName` to a lowercase Cogent name. No additional information is required for that.

For generating a file specific function name, the file name of the definition is required. Note that this is only done for a function with internal linkage, where the definitions must be present in the input whenever the function is referenced. The definition contains the position information which includes the file name. Hence, the symbol table is sufficient for translating the name, to make it available the translation function is defined as a monadic action.

In C bodies function names cannot be syntactically distinguished from variable names. Therefore, Gencot uses a common function for translating function and variable names. For a description how variable names are translated see Section 3.5.9.

```
transObjName :: Ident -> FTrav String
```

Similar as for tags, the function is also used when translating a function definition, although the definition is already available.

Cogent Constant Names

Cogent constant names are only generated from C enum constant names. They are simply translated with the help of `mapName` to a lowercase Cogent name. No additional information is required.

Cogent Field Names

C member names and parameter names are translated to Cogent field names. Only if the C name is uppercase, the name is mapped to a lowercase Cogent

name with the help of `mapName`, otherwise it is used without change. Only the C name is required for that, in both cases it is available as a value of type `Ident`. The translation is implemented by the function

```
mapIfUpper :: Ident -> String
```

3.5.8 Generating Origin Markers

For outputting origin markers in the target code, the AST prettyprint functionality must be extended.

The class `Pretty` used by the Cogent prettyprinter defines the methods

```
pretty :: a -> Doc
prettyList :: [a] -> Doc
```

but the method `prettyList` is not used by Cogent. Hence, only the method `pretty` needs to be defined for instances. The type `Doc` is that from module `Text.PrettyPrint.ANSI.Leijen`.

The basic approach is to wrap every syntactic construct in a sequence of `#ORIGIN` markers and a sequence of `#ENDORIG` markers according to the origin information for the construct in the extended AST. This is done by an instance definition of the form

```
instance Pretty GenToplv where
  pretty (GenToplv org t) = addOrig org $ pretty t
```

for `GenToplv` and analogous for the other types. The function `addOrig` has the type

```
addOrig :: Origin -> Doc -> Doc
```

and wraps its second argument in the origin markers according to its first argument.

The Cogent prettyprinter uses indentation for subexpressions. Indentation is implemented by the `Doc` type, where it is called "nesting". The prettyprinter maintains a current nesting level and inserts that amount of spaces whenever a new line starts.

The origin markers must be positioned in a separate line, hence `addOrig` outputs a newline before and after each marker. This is done even at the beginning of a line, since due to indentation it cannot safely be determined whether the current position is at the beginning of a line. Cogent may change the nesting of the next line after `addOrig` has output a marker (typically after an `#ENDORIG` marker). The newline at the end of the previous marker still inserts spaces according to the old nesting level, which determines the current position at the begin of the following marker. This is not related to the new nesting level.

This way many additional newlines are generated, in particular an empty line is inserted between all consecutive origin markers. The additional newlines are later removed together with the markers, when the markers are processed. Note that, if a syntactic construct is nested, the indentation also applies to the origin markers and the line after it. To completely remove an origin marker from the target code it must be removed together with the newline before it and with

the newline after it and the following indentation. The following indentation can be determined since it is the same as that for the marker itself (a sequence of blanks of the same length).

Repeated Origin Markers

Normally, target code is positioned in the same order as the corresponding source code. This implies, that origin markers are monotonic. A repeated origin marker is a marker with the same line number as its previous marker. Repeated origin markers of the same kind must be avoided, since they would result in duplicated comments or misplaced directives. Repeated origin markers of the same kind occur, if a subpart of a structured source code part begins or ends in the same line as its main part. In this case only the outermost markers must be retained.

An `#ENDORIG` marker repeating an `#ORIGIN` marker means that the source code part occupies only one single line (or a part of it), this is a valid case. An `#ORIGIN` marker repeating an `#ENDORIG` marker means that the previous source code part ends in the same line where the following source code part begins. In this case the markers are irrelevant, since no comments or directives can be associated with them. However, if they are present they introduce unwanted line breaks, hence they also are avoided by removing both of them.

Together, the following rules result. In a sequence of repeated `#ORIGIN` markers, only the first one is generated. In a sequence of repeated `#ENDORIG` markers only the last one is generated. If an `#ORIGIN` marker repeats an `#ENDORIG` marker, both are omitted.

There are several possible approaches for omitting repeated origin markers:

- omit repeated markers when building the Cogent AST
- traverse the Cogent AST and remove markers to be omitted
- output repeated markers and remove them in a postprocessing step

Note, that it is not possible to remove repeated markers already in the language-c AST, since there a `NodeInfo` value always corresponds to two combined markers.

Handling repeated markers in the Cogent AST is difficult, because for an `#ORIGIN` marker the context before it is relevant whereas for an `#ENDORIG` marker the context after it is relevant. An additional AST traversal would be required to determine the context information. The first approach is even more complex since the context information must be determined from the source code AST where the origin markers are not yet present.

For this reason Gencot uses the third approach and processes repeated markers in the generated target code text, independent from the syntactical structure.

Filter for Repeated Origin Marker Elimination

The filter `gencot-reporigs` is used for removing repeated origin markers. It is implemented as an awk script.

It uses five string buffers: two for the previous two origin markers read, and three for the code before, between, and after both markers. Whenever all buffers are filled (the buffer after both markers with a single text line; this line exists, since consecutive markers are always separated by an empty line), the markers

are processed as follows, if they have the same line number: in the case of two `#ORIGIN` markers the second is deleted, in the case of two `#ENDORIG` markers the first is deleted, and in the case of an `#ORIGIN` marker after an `#ENDORIG` marker both are deleted. In the latter case the line number of the `#ORIGIN` marker is remembered and subsequent `#ORIGIN` markers with the same line number are also deleted.

When both markers have different line numbers or if an `#ENDORIG` marker follows an `#ORIGIN` marker the first marker and the code before it are output and the buffers are filled until the next marker has been read.

3.5.9 Generating Expressions

For outputting the Cogent AST the prettyprint functionality must be extended to output C function bodies and the C expressions used for constant definitions. Additionally, at least in function bodies, origin markers must be generated to be able to re-insert comments and preprocessor directives. Finally, all names occurring free in a function body or a constant expression must be mapped to Cogent names.

The language-c prettyprinter is defined in module `Language.C.Pretty`. It defines an own class `Pretty` with method `pretty` to convert the AST types to a `Doc`. However, other than the Cogent prettyprinter, it uses the type `Doc` from module `Text.PrettyPrint.HughesPJ` instead of module `Text.PrettyPrint.ANSI.Leijen`. This could be adapted by rendering the `Doc` as a string and then prettyprinting this string to a `Doc` from the latter module. This way, a prettyprinted function body could be inserted in the document created by the Cogent prettyprinter.

Origin Markers

For generating origin markers, a similar approach is not possible, since they must be inserted between single statements, hence, the function `pretty` must be extended. Although it does not use the `NodeInfo`, it is only defined for the AST type instances with a `NodeInfo` parameter and has no genericity which could be exploited for extending it. Therefore, Gencot has to fully reimplement it.

In the prettyprint reimplementation the target code parts must be wrapped by origin markers in the same way as for the Cogent AST. However, for the type `Doc` from module `Text.PrettyPrint.HughesPJ` this is not possible, since newlines are only available as separators between documents and cannot be inserted before or after a document. An alternative choice would be to use the type `Doc` from `Text.PrettyPrint.ANSI.Leijen`, as the Cogent prettyprinter does. However, the approach of both modules is quite different so that it would be necessary to write a new C prettyprint implementation nearly from scratch.

It has been decided to use another approach which is expected to be simpler. The alternative C parser language-c-quote also has a prettyprinter. It generates a type `Doc` defined by a third module `Text.PrettyPrint.Mainland`. It is similar to `Text.PrettyPrint.ANSI.Leijen` and also supports adding newlines before and after a document. The language-c-quote prettyprinter is defined in the module `Language.C.Pretty` of language-c-quote and consists of the method `ppr` of the class `Pretty` defined in module `Text.PrettyPrint.Mainland.Class.Pretty`. This method is not generic at all, hence it must be completely reimplemented

to extend it for generating origin markers. However, this reimplementation is straightforward and can be done by copying the original implementation and only adding the origin marker wrappings. The resulting Gencot module is `Gencot.C.Output`.

Whereas the type `Doc` from `Text.PrettyPrint.ANSI.Leijen` provides a `hardline` document which always causes a newline in the output, the type `Doc` from `Text.PrettyPrint.Mainland` does not. Normal line breaks are ignored in certain contexts, if there is enough room. Using normal line breaks around origin markers could result in origin markers with other code in the same line before or after the marker.

For the reimplemented language-c-quote prettyprinter Gencot defines its own `hardline` by using a newline which is hidden for type `Doc`. This could be implemented without nesting the marker and the subsequent line. However, if at the marker position a comment is inserted, the subsequent line should be correctly indented. To achieve this, the `hardline` implementation also adds the current nesting after the newline.

Hiding the newline from `Doc` implies that the "current column" maintained by `Doc` is not correct anymore, since it is not reset by the `hardline`. Every `hardline` will instead advance the current column by the width of the marker and twice the current nesting. This has two consequences.

First, in some places the language-c-quote prettyprinter uses "alignment" which means an indentation of subsequent lines to the current column. This indentation will be too large after inserted markers. Gencot handles this by replacing alignment everywhere in the prettyprint implementation by a nesting of two additional columns.

Second, the language-c-quote prettyprinter is parameterized by a "document width". It automatically breaks lines when the current column exceeds the document width. The incorrect column calculation causes many additional such line breaks, since the current column increases much faster than normal. Gencot handles this by setting the document width to a very large value (such as 2000 instead of 80) to compensate for the fast column increase.

Using the language-c-quote AST

Language-c-quote uses a different C AST implementation than language-c. To use its reimplemented prettyprinter, the language-c AST must be translated to a language-c-quote AST. This is not trivial, since the structures are somewhat different, but it seems to be simpler than implementing a new C prettyprinter. The translation is implemented in the module `Gencot.C.Translate`.

Additionally the language-c-quote AST must be extended by `Origin` values. The language-c-quote AST already contains `SrcLoc` values which are similar to the `NodeInfo` values in language-c. Like these they cannot be used as origin marker information since they cannot represent begin and end markers independently. Therefore Gencot also reimplements the language-c-quote AST by copying its data types and replacing the `SrcLoc` values by `Origin` values. This is implemented in module `Gencot.C.Ast`.

Together, this approach yields a similar structure as for the translation to Cogent: The Cogent AST is extended by the structures in `Gencot.C.Ast` to represent function bodies and constant expressions. The function for translating from language-c AST to the Cogent AST is extended by the functions

in `Gencot.C.Translate` to translate function bodies and constant expressions from the language-c AST to the reimplemented language-c-quote AST, and the Cogent prettyprinter is extended by the prettyprinter in `Gencot.C.Output` to print function bodies and constant expressions with origin markers.

In addition to translating the C AST structures from language-c to those of language-c-quote, the translation function in `Gencot.C.Translate` implements the following functionality:

- generate `Origin` values from `NodeInfo` values,
- map C names to Cogent names.

Name Mapping

Name mapping depends on the kind of name and may additionally depend on its type. Both information is available in the symbol table (see Section 3.5.10). However, the scope cannot be queried from the symbol table. Hence it is not possible to map names depending on whether they are locally defined or globally.

The following kinds of names may occur in a function body: primitive types, typedefs, tags, members, functions, global variables, enum constants, preprocessor constants, parameters and local variables.

Primitive type names and typedef names can only occur as name of a base type in a declaration. Primitive type names are mapped to Cogent primitive type names as described in Section 2.6.1.

A typedef name may also occur in a declarator of a local typedef which defines the name. In both cases, as described in Section 2.9.1, Gencot only maps the plain typedef names, not the derived types. The typedef names are mapped according to Section 2.6.7: If they ultimately resolve to a struct, union, or array type they are mapped with an unbox operator applied, otherwise they are mapped without.

A tag name can only occur as base type in a declaration. It is always mapped to a name with a prefix of `Struct_`, `Union_`, or `Enum_`. Tagless structs/unions/enums are not mapped at all. Tag names are mapped according to Sections 2.6.2 and 2.6.3: struct and union tags are mapped with an unbox operator applied, enum tags are mapped without.

Gencot also maps defining tag occurrences. Thus an occurrence of the form

```
struct s { ... }
```

is translated to

```
struct #Struct_s { ... }
```

Every occurrence of a field name can be syntactically distinguished. It is mapped according to Section 2.1.1 to a lowercase Cogent name if it is uppercase, otherwise it is unchanged. Field names are also mapped in member declarations in locally defined structures and unions.

All other names syntactically occur as a primary expression. They are mapped depending on their semantic information retrieved from the symbol table. In a first step it distinguishes objects, functions, and enumerators.

An object identifier may be a global variable, parameter, or local variable. It may also be a preprocessor constant since for them dummy declarations have

been introduced which makes them appear as a global variable for the C analysis. For the mapping the linkage is relevant, this is also available from the symbol table.

Identifiers for global variables may have external or internal linkage and are mapped depending on the linkage. Identifiers for parameters always have no linkage and are always mapped like field names. Identifiers for local variables either have no linkage or external linkage. In the first case they are mapped like field names. In the second case they cannot be distinguished from global variables with external linkage, and are mapped to lowercase. The dummy declarations introduced for preprocessor constants always have external linkage, the identifiers are mapped to lowercase. Together, object identifiers with internal linkage are mapped as described in Section 2.1.1, object identifiers with external linkage are mapped to lowercase, and object identifiers with no linkage are mapped to lowercase if they are uppercase and remain unchanged otherwise.

An identifier for a function has either internal or external linkage and is mapped depending on its linkage. An identifier for an enumerator is always mapped to lowercase, like preprocessor constants.

Identifiers for local variables may also occur in a declarator of a local object definition which defines the name. They are also mapped depending on their linkage, as described above.

3.5.10 Traversing the C AST

The package language-c uses a monad for traversing and analysing the C AST. The monad is defined in module `Language.C.Analysis.TravMonad` and mainly provides the symbol table and user state during the traversal. The traversal itself is implemented by a recursive descent according to the C AST using a separate function for analysing every syntactic construct.

When processing the semantic map resulting from the language-c analysis Gencot implements similar recursive descents using a processing function for every syntactic construct. For this it uses the same monad for two reasons.

- the definitions and declarations of the global identifiers are needed for accessing their types and for mapping the identifiers to Cogent names,
- additionally, the definitions and declarations of locally defined identifiers are needed in C function bodies for the same purpose.

The global definitions and declarations in the symbol table correspond to the semantics map which is the result of the language-c analysis step. It is created from the symbol table after the initial traversal of the C AST. Although Gencot processes the content of the semantics map, it is not available as a whole in the processing functions. Instead of passing the semantics map as an explicit parameter to all processing functions, Gencot uses monadic traversals through the relevant parts of the semantics map, which implicitly make the symbol table available to all processing functions. This is achieved by reusing the symbol table after the analysis phase for the traversals of the semantics map.

Additionally, when processing the C function bodies, the symbol table is used for managing the local declarations. This is possible because although the analysis phase translates global declarations and definitions to a semantic representation, it does not modify function bodies and returns them as the

original C AST. Since the information about local declarations is discarded at the end of its scope, the information is not present anymore in the symbol table after the analysis phase. Gencot uses the symbol table functionality to rebuild this information during its own traversals.

The user state is used by Gencot to provide additional information, depending on the purpose of the traversal. A common case is to make the actual name of the processed file available during processing. In the `NodeInfo` values in the AST it is always specified as `<stdin>` since the input is read from a pipe. All C processing filters take the name of the original C source file as an additional argument. It is added to the user state of traversal monads so that it can be used during traversal.

This is supported by defining in module `Gencot.Name` the class `FileNameTrav` as

```
class (Monad m) => FileNameTrav m where
  getFileName :: m String
```

so that the method `getFileName` can be used to retrieve the source file name from all traversal monads of this class. Based on this the monadic action `srcFileName` is defined to return the file name for a `NodeInfo` value and replace it by the original source file name if it is equal to `<stdin>`.

Another information needed during C AST traversal is the parameter modification description (see Section 3.2). It is used for translating the types of all functions, as described in Section 2.6.5.

Finally, Gencot maintains a list of tag definitions which are processed in advance as nested, as described in Section 3.5.3. This list is implemented as a list of elements of type `SUERef` which is used by language-c to identify both tagged and untagged struct/union/enum types.

The utilities for the monadic traversal of the semantics map are defined in module `Gencot.Traversal`. The main monadic type is defined as

```
type FTrav = Trav (String,ParmodMap,[SUERef])
```

where `String` is the type used for storing the original C source file name in the user state, `ParmodMap` is the type for storing the parameter modification description, and the third component is the list for maintaining tag definitions processed as nested. `FTrav` is an instance of `FileNameTrav`. As execution function for the monadic actions the functions

```
runFTrav :: DefTable -> (String,ParmodMap) -> FTrav a -> IO a
runWithTable :: DefTable -> FTrav a -> IO a
```

are defined. The first one takes the symbol table, the original C source file name and the parameter modification description as arguments to initialize the state. The tag definition list is always initialized as empty. The second function leaves also the other two components empty. The functions are themselves `IO` actions and print error messages generated during traversal to the standard output.

In the monadic actions the symbol table can be accessed by actions defined in the modules `Language.C.Analysis.TravMonad` and `Language.C.Analysis.DefTable`. An identifier can be resolved using the actions

```
lookupTypeDef :: Ident -> FTrav Type
lookupObject :: Ident -> FTrav (Maybe IdentDecl)
```

For resolving tag definitions the symbol table must be retrieved by

```
getDefTable :: FTrav DefTable
```

then the struct/union/enum reference can be resolved by

```
lookupTag :: SUERef -> DefTable -> Maybe TagEntry
```

To maintain the list of tag definitions processed as nested two monadic actions are defined:

```
markTagAsNested :: SUERef -> FTrav ()
isMarkedAsNested :: SUERef -> FTrav Bool
```

Additionally, there are actions to enter and leave a scope and actions for inserting definitions.

An error can be recorded in the monad using the action

```
recordError :: Language.C.Data.Error.Error e => e -> m ()
```

The parameter modification description can be accessed by the monadic action

```
getParmods :: String -> FTrav [String]
```

where the `String` argument is a function identifier and the result is the list of parameter description values for the parameters of the identified function.

3.5.11 Creating and Using the C Call Graph

In some Gencot components we use the C call graph. This is the mapping from functions to the functions invoked in their body. Here we describe the module `Gencot.Util.CallGraph` which provides utility functions for creating and using the call graph.

The set of invoked functions is determined by traversing the bodies of all function definitions after the analysis phase. The callback handler is not used since it is only invoked for declarations and definitions and does not help for processing function invocations.

Invocations can be identified purely syntactically as C function call expressions. The invoked function is usually specified by an identifier, however, it can be specified as an arbitrary C expression. We only support the cases where the invoked function is specified as an identifier for a function or function pointer, by a chain of member access operations starting at an identifier for an object of struct or union type, or by an array index expression where the array is specified as an identifier or member access chain and the element type is a function pointer type. All other invocation where the invoked function is specified in a different way are ignored and not added to the call graph.

The starting identifier can be locally declared, such as a parameter of the function where the invocation occurs. The declaration information of these identifiers would not be available after the traversal which builds the call graph. To make the full information about the invoked functions available, Gencot inserts the declarations into the call graph instead of the identifiers. In the case of a member access chain it uses the struct or union type which has the invoked function pointer or the indexed function pointer array as its direct member.

The information about such an invocation in a function body is represented by the following type:

```
data CGInvoke =
    IdentInvoke IdentDecl Int
  | MemberTypeInvoke CompType MemberDecl Int
```

The additional integer value specifies the number of actual arguments in this invocation. Note that in a function definition the parameters are represented in the symbol table by `IdentDecl`s, not by `ParamDecl`s. In the case of an array element invocation the actual index is ignored, all array elements are treated in a common way.

The call graph has the form of a set of globally described invocations. These are triples consisting of the definition of the invoking function, the invocation, and a boolean value telling whether the identifier in the case of an `IdentInvoke` is locally defined in the invoking function:

```
type CallGraph = Set CGFunInvoke
type CGFunInvoke = (FunDef, CGInvoke, Bool)
```

The equality relation for values of type `CGFunInvoke` is based on the location of the contained declarations in the source file. This is correct since after the initial traversal every identifier has a unique declaration associated.

To access the declarations of locally declared identifiers, the symbol table with local declarations must be available while building the call graph. Therefore we traverse the function bodies with the help of the `FTrav` monad and `runWithTable` as described in Section 3.5.10.

The call graph is constructed by the monadic action

```
getCallGraph :: [DeclEvent] -> FTrav CallGraph
```

It processes all function definitions in its argument list and ignores all other `DeclEvent`s.

The function

```
getIdentInvokes :: CallGraph -> Set LCA.IdentDecl
```

returns the set of all invoked functions which are specified directly as an identifier. In particular, they include all invoked functions which are no function pointers.

The declaration of an invoked function also tells whether the function or object is defined or only declared. Note that the traversal for collecting invocations is a "second pass" through the C source after the analysis phase of language-c. During analysis language-c replaces declarations in the symbol table whenever it finds the corresponding definition.

To use the call graph the `CallGraph` module defines a traversal monad `CTrav` with the call graph in the user state (in addition to the own source file name). The corresponding execution function is

```
runCTrav :: CallGraph -> DefTable -> String -> CTrav a -> IO a
```

The monadic action to access the call graph is

```
lookupCallGraph :: Ident -> CTrav CallGraph
```

It takes the identifier of an invoking function as argument and returns the part of the call graph for this function, consisting of all invocations in its body.

The monad `CTrav` is an instance of class `FileNameTrav`, so the own source file name can be accessed by `getFileName` (see Section 3.5.10).

3.6 Implementing Basic Operations

Gencot supports basic operations either by providing a Cogent implementation, or by providing a C implementation for functions defined as abstract in Cogent.

3.6.1 Implementing Polymorphic Functions

are used in a program this is not detected as error by the Cogent compiler. However, since Gencot does not provide implementations for such instances the resulting C program will not compile.

3.6.2 Implementing Pointer Types

3.6.3 Implementing Function Pointer Types

3.6.4 Implementing MayNull

3.6.5 Implementing Record Types

3.6.6 Implementing Array Types

Creating and Disposing Arrays

The `create` instance is implemented by simply allocating the required space on the heap, internally using the translation of type `#A<size>_El` to specify the amount of space.

3.7 C Processing Components

As described in Section 2.1.2 there are several different Gencot components which process C code and generate target code.

3.7.1 Filters for C Code Processing

All filters which parse and process C code are implemented in Haskell and read the C code as described in Section 3.5.3.

The following filters always process the content of a single C source file and produce the content for a single target file.

`gencot-translate` translates a single file `x.c` or `x.h` to the Cogent code to be put in file `x.cogent` or `x-incl.cogent`. It processes typedefs, struct/union/enum definitions, and function definitions.

`gencot-globals` translates a single file `x.c` to the Cogent code to be put in file `x-globals.cogent`. It processes all global variable definitions.

`gencot-entries` translates a single file `x.c` to the antiquoted C entry wrappers to be put in file `x-entry.ac`. It processes all function definitions with external linkage.

`gencot-abstypes` translates a single file `x.c` or `x.h` to the C typedefs to be put in file `x-abstypes.c` or `x-abstypes.h`. It processes typedefs and struct/union/enum definitions.

`gencot-remfundef` processes a single file `x.c` by removing all function definitions. The output is intended to be put in file `x-globals.c`

`gencot-deccomments` processes a single file `x.c` or `x.h` to generate the list of all declaration positions.

`parmod-gen` processes a single file `x.c` and generates the function parameter modification descriptions (see Section 3.2).

All these filters take the name of the original source file as additional first argument, since they need it to generate Cogent names for C names with internal linkage and for tagless C struct/union types.

There are other target files which are generated for the whole Cogent compilation unit. The filters for generating these target files take as input the list of file names to be processed (see Section 3.5.4). Filters of this kind are called "processors" in the following.

Usually only `.c` files need to be specified as input to processors. In contrast to the single-file filters, the original file name is not required for the files input to a Gencot processor. A processor only processes items which are external to all input files, whereas the original file name is needed for items which are defined in the input files. Therefore the list of input file names is sufficient and the input files may have arbitrary names which need not be related to the names of the original `.c` files.

Gencot uses the following processors of this kind:

`gencot-exttypes` generates the content to be put in the file `<package>-exttypes.cogent`. It processes externally referenced typedefs, tag definitions and enum constant definitions.

`gencot-dvdtypes` generates the content to be put in the file `<package>-dvdtypes.cogent`. It processes all used derived types.

`gencot-dvdtypesh` generates the content to be put in the file `<package>-dvdtypes.h`. It processes all used derived types.

`gencot-dvdtypesac` generates the content to be put in the file `<package>-dvdtypes.ac`. It processes all used derived types.

`gencot-externs` generates the abstract function definitions of the exit wrappers to be put in the file `<package>-externs.cogent`. It processes the declarations of externally referenced functions and variables.

`gencot-externsac` generates the exit wrappers to be put in the file `<package>-externs.ac`. It processes the declarations of externally referenced functions.

`parmod-externs` generates the list of function identifiers of all externally referenced functions and function pointer (array)s.

3.7.2 Main Translation to Cogent

The main translation from C to Cogent is implemented by the filter `gencot-translate`. It translates `DeclEvent`s of the following kinds:

- struct/union definitions

- enum definitions with a tag
- enum constant definitions
- function definitions
- type definitions

The remaining global items are removed by the predicate passed to `Gencot.Input.getDeclEvents`: all declarations, all object definitions, and all tagless enum definitions. No Cogent type name is generated for a tagless enum definition, references to it are always directly replaced by type `U32`.

The translation does not use the callback handler to collect information during analysis. It only uses the semantics map created by the analysis and processes the `DeclEvent` sequence created by preprocessing as described in Section 3.5.3.

The translation of the `DeclEvent` sequence is implemented by the function

```
transGlobals :: [DeclEvent] -> FTrav [GenToplv]
```

It performs the monadic traversal as described in Section 3.5.10 and returns the list of toplevel Cogent definitions of type `GenToplv`.

A struct/union/enum definition corresponds to a full specifier, as described in Section 2.8.1. The language-c analyser already implements moving all full specifiers to separate global definitions and the sorting step done by `Gencot.Input.getDeclEvents` creates the desired ordering. Therefore, only the translation of the single struct/union/enum definitions has to be implemented by `gencot-translate`.

A struct/union definition is translated to a Cogent type definition where the type name is constructed as described in Section 2.1.1. A struct is translated to a corresponding record type, a union is translated to an abstract type, as described in Section 2.6. In both cases the type name names the boxed type, i.e., it corresponds to the C type of a pointer to the struct/union.

An enum definition with a tag is translated to a Cogent type definition where the type name is constructed as described in Section 2.1.1. The name is always defined for type `U32`, as described in Section 2.6.

An enum constant definition is translated to a Cogent constant definition where the name is constructed as described in Section 2.1.1 and the type is always `U32`, as described in Section 2.6.

A function definition is translated to a Cogent function definition, as described in Section 2.9.

A type definition is translated to a Cogent type definition as described in Section 2.8.3.

All these single translations are implemented by the function

```
transGlobal :: DeclEvent -> FTrav GenToplv
```

A type reference is translated by the function

```
transType :: Type -> FTrav GenType
```

A type reference may be a direct type, a derived type, or a typedef name. For every typedef name a Cogent type name is defined, as described in Sections 2.1.1 and 2.1.2. A direct type is either the type `void`, a primitive C type, which is

mapped to the name of a primitive Cogent type, or it is a struct/union/enum type reference for which Gencot also introduces a Cogent type name or maps it to the primitive Cogent type `U32` (tagless enums). Hence, both direct types and typedef names can always be mapped to Cogent type names, with the exception of type `void`, which is mapped to the Cogent unit type `()`.

——— todo: rest of subsection must be revised ——— If a typedef name references (directly or indirectly) a struct or union type, the corresponding Cogent type name references the boxed type. Therefore, it must be modified by applying the unbox operator. If a typedef name references a primitive type, this is not necessary, since the corresponding Cogent type is always regular. However, the abstract Cogent surface syntax always associates a "sigil" with a type name. Unnecessary unbox operators are automatically suppressed by the Cogent prettyprint function. Therefore Gencot always associates an unbox sigil with the Cogent type name if it corresponds to a direct type or a typedef name referencing a direct type.

A derived type is either a pointer type, an array type, or a function type. It is derived from a base type which in case of a function type is the type of the function result. The base type may again be a derived type, ultimately it is a direct type or a typedef name.

For a pointer type the translation depends on the base type. If it is a struct or union type or a typedef name referencing a struct or union type, the pointer type is translated to the Cogent type name corresponding to the base type. If it is a function type or a typedef name referencing a function type, the pointer type is translated to the translation of the base type. In all other cases, as described in Section 2.6, the pointer type is translated to the name of an abstract type, which is introduced as described in Section 2.1.1, using a name for the base type.

For an array type, the translation also depends on the base type. If it is type `char`, the array type is translated to the primitive Cogent type `String`. In all other cases it is translated to the name of an abstract type, which is introduced as described in Section 2.1.1, using a name for the base type. This is even done if the base type is a typedef name which references type char, since Gencot assumes that in this case it is intended as a "real" array type and not as a string type.

If an array type occurs as type of a function parameter, it is "adjusted" by C to a pointer type with the same base type. This is also done by Gencot.

A function type is always translated to the corresponding Cogent function type, where a tuple type is used as parameter type if there is more than one parameter, and the unit type is used if there is no parameter.

If a pointer type or array type is translated to the name of an abstract type, a name is required for the base type. If the base type is a direct type or a typedef name, a Cogent type name always exists and is used. The only exception is type `void`, here the name `Void` is used. If the base type is a derived type, a name is constructed for it as described in Section 2.1.1, even if the base type would normally be mapped to a type expression (which is the case for a function type) or to its own base type (which is the case for pointer types to struct/union and function types). If the base type is array of char, the name `String` is used for it.

Note that for most cases where a typedef name occurs as reference or base type, it must be resolved to the ultimate direct type or derived type referenced

by it. This is implemented by the function

```
resolveTypeDef :: TypeDefRef -> Type
```

The `GlobalDecls` map is not required here, since the language-c analyser puts the (directly) referenced type in the `TypeDefRef`.

For a qualified C type Gencot only respects the `const` qualifier. For a direct type the `const` qualifier is ignored, since in Cogent values of unboxed and regular types are always immutable. For a function type the qualifier is also ignored since function types are regular in Cogent. For an array of char it is ignored since it is translated to type `String` which is regular.

All other array types and all pointer types are translated to linear types which can be mutable in Cogent. Whenever the C type contains no mutable pointer types, it is translated to a readonly Cogent type by applying a bang operator to it.

An array type contains mutable pointers if its base type does so. A function type never contains mutable pointers. A pointer type contains mutable pointers if its base type is not `const` qualified or contains mutable pointers. A primitive type and an enum type does not contain mutable pointers. A struct or union type contains mutable pointers, if the type of a member contains mutable pointers.

3.7.3 External Functions and Variables

For the exit wrapper functions Gencot generates the Cogent abstract function definitions and the implementations in antiquoted C. Exit wrappers are generated for all functions which are invoked but not defined in the Cogent code. The information required for each of these functions is contained in the C function declaration.

Additionally, Gencot generates Cogent abstract function definitions for access functions to selected external variables without providing an implementation in antiquoted C.

Determining External Functions

The easiest approach would be to generate an exit wrapper for every function which is declared but not defined in the C program. However, often an included C source file contains many function declaration where only some of them are invoked. Therefore, Gencot determines the declarations of all actually invoked external functions with the help of the call graph as described in Section 3.5.11. Only for them declarations are translated.

Note that external functions which are not invoked directly, but converted to a function pointer and invoked through a function pointer object are not included by this approach, they must be handled manually.

The external invoked functions are further filtered as follows:

- If the type is a function pointer, invocations are translated to Cogent in a different way as described in Section 2.6.5. Therefore we omit all function pointer invocations.
- A function which is only declared may be incomplete, i.e., its parameters (number and types) are not known. We could translate such function

declarations to Cogent abstract functions with Unit as argument type, however, that would not be related to the invocations and thus be useless. Therefore we omit invocations of incompletely declared functions.

An identifier seen by Gencot in a function invocation may also be a macro call which shall be translated by Gencot. In this case a dummy function declaration must be manually provided for it as part of the Gencot macro call conversion (see Section 2.4.3), so that the language-c parser can parse the invocations. By using either an incomplete declaration or a complete declaration it can be controlled whether a Cogent abstract function definition is generated for it. If the macro is translated to a macro in Cogent, no abstract function definition shall be generated and an incomplete dummy declaration must be used.

There are cases where an external function is used without being seen by the call graph. This is the case when the function is assigned to a function pointer or when the function invocation is created by a macro call and is thus not visible for the call graph. For these cases Gencot supports a list where additional functions to be processed can be specified in an auxiliary input file by their names, each in a seperate line. Only functions with external linkage can be specified, for them the name is unique in the <package>.

Determining External Variables

As for functions, Gencot also restricts the processed external variables to a subset of the declared variables. It would be possible to determine the accessed variables in a similar way as the functions are determined with the help of the call graph. Gencot does this for all variables of function pointer (array) type, using the call graph. Variables accessed in other ways or of other types may be specified explicitly as a list. This provides more control about the processed external variables for the developer.

The external variables are specified in the same list as the additional external functions.

Processor `gencot-externs`

The processor `gencot-externs` generates the Cogent abstract function definitions. It is invoked as

```
gencot-externs [<parmod file> [<varlist file>]]
```

where the optional arguments are a file with parameter modification descriptions and with the names of external variables to be processed.

The processor parses and analyses all C source files specified in the input file name list, resulting in the list of corresponding symbol tables, as described in Section 3.5.4. Then it determines for every table the invoked functions and function pointer (array)s as described in Section 3.5.11. This must be done before combining the tables, since invocations may also occur in the bodies of functions with internal linkage. These functions are removed during table combination.

After determining the invoked functions, the tables are combined as described in Section 3.5.4 and the invocations are reduced to those for which a declarations is present in the combined table. These are the invocations of

functions and function pointer (array) variables external to all read C sources. The processor translates their declarations using the function `transGlobals` of module `Gencot.Cogent.Translate`. It translates C function and variable declarations to abstract function definitions in Cogent. It reads and uses parameter modification descriptions to determine readonly parameter types, the descriptions are generated by `parmod-externs` as described in Section 3.7.7. The result is output using `prettyTopLevels` from module `Gencot.Cogent.Output`.

The processor takes two files as optional arguments. The first file contains parameter modification descriptions and is used for translating the types of functions and function pointers. If it is not specified all linear parameters are assumed to be modifyable by the function. The second file contains a list of newline-separated names of external variables to be processed in addition to the invoked function pointer (array)s. This file can only be specified if the parameter modification description file is also specified. If no parameter modifications shall be used, a file containing the empty list `[]` can be used.

Processor `gencot-externsac`

The processor `gencot-externsac` generates the wrapper implementations in antiquoted C code.

3.7.4 External Cogent Types

The processor `gencot-exttypes` generates all Cogent type definitions which origin from a C system include file. It is invoked as

```
gencot-exttypes [<parmod file> [<varlist file>]]
```

where the optional arguments are a file with parameter modification descriptions and with the names of external variables to be processed.

Since often only a small selection of the types defined by a system include file is used by a program, Gencot performs an analysis to reduce the type definitions to be translated. It determines all types actually *used* by referencing them somewhere in the program.

C Types are used in a C source in the following cases:

- As type of a global or local variable.
- As result or parameter type of a derived function type.
- As base type of a derived pointer or array type.
- As type of a field in a composite type (struct or union).
- As defining type for a typedef name.

There are other uses of types in C, such as in `sizeof` expressions, these cases are ignored by Gencot and must be handled manually.

We call the items which may have a separate syntactically specified C type a "type carrier". These are variables, functions, fields, and typedef names. Since the language-c parser constructs for every defined or declared function the derived function type, variables and functions can be treated in the same way, they

only differ in their type: functions have a derived function type, whereas variables may have all other types, including types derived from function types such as function pointer types. Fields always belong to a composite type, therefore we use as type carrier the composite type instead of the single fields. Together we have three kinds of type carriers: variable/functions, composite types, and typedef names.

Every type carrier has one or (in the case of a composite type) several associated "syntactic types" which are the type expressions as specified in the C source. If a syntactic type is a derived type it contains "syntactic part types" (base types, parameter and result types).

We also include enum types as type carriers. They have no associated types, since they always correspond to a name for a primitve (numeric) type. However, `gencot-exttypes` must determine and translate enum types which are defined in system include files similar to a composite type.

Generally, Gencot translates type carriers to Cogent. It translates functions as described in Section 2.9 and it translates enum types, composite types and type definitions as described in Section 2.6. Gencot translates global variables to an abstract access function, using the translated type of the variable. Gencot does not translate local variables, this must be done manually. To support the manual translation, however, Gencot treats local variables as if it would translate them to Cogent code referencing the translated declared type.

The task of `gencot-exttypes` is to translate all type carriers referenced in the translated code for which no definition has been generated by translating the C sources of the Cogent translation unit, i.e., which have been defined in a system include file.

When Gencot translates a type carrier, it references at most the associated syntactic types in the resulting Cogent code. This is the case for struct types and for functions. Union types are translated without referencing the field types (see Section 2.6.3). To support a manual translation `gencot-exttypes` treats them in the same way as struct types.

Gencot always translates a derived type to a type name of a Cogent abstract type. Hence the syntactic part types are not referenced in the translated Cogent code directly. However, for complementing the abstract type by abstract functions for working with it, the syntactic part types may be needed. For derived function pointer types Gencot automatically provides the abstract from/to/invk conversion functions (see Section 2.6.5), for derived array types it provides functions as described in Section 2.6.4, for other derived types functions may be defined manually. Therefore `gencot-exttypes` also handles all syntactic part types associated with a type carrier.

For functions, the associated syntactic type is either a typedef name (resolving to a derived function type) or a syntactic derived function type. In the latter case no Cogent type definition is needed for it, since Gencot directly translates it to a Cogent type expression (Section 2.9).

Collecting Used Types

A type can be used by code in a translated C soure file or by a declaration of an external function or variable. As described in Section 3.7.3, Gencot only translates external functions and variables if they are actually invoked or explicitly listed by the developer. Therefore, `gencot-exttypes` determines the

external invoked functions and variables in the same way. Note that the external invoked functions include those defined in the <package> outside the Cogent compilation unit, as well as those declared in a system include file.

Enum types, composite types and typedef names are reduced by an analysis of type usage by reference. Every type carrier can be associated with the enum types, composite types and typedef names referenced in all associated syntactic part types. For composite types and typedef names this usage relation is transitive. To determine the relevant enum types, composite types and typedef names defined in system include files, `gencot-exttypes` starts with all initial type carriers belonging to the Cogent compilation unit, together with all external invoked functions and the processed external variables. It then determines all transitively used enum types, composite types and typedef names (which all must be defined in system include files).

To further reduce type carriers, the types associated with external type carriers are "fully resolved", i.e. all typedef names occurring in them are transitively resolved and replaced by their definition. The exception are typedef names which are referenced by the initial type carriers, they are preserved since they are needed anyways.

For the resulting set of type carriers the required type definitions are generated.

The initial type carriers are collected with the help of the callback handler during analysis, as described in Section 3.5.3. All type carriers can be wrapped as a `DeclEvent`. The callback handler is automatically invoked for all type carriers in the parsed C source files, including local variables (which are not present anymore in the symbol table after the analysis phase).

Gencot uses the callback handler

```
collectTypeCarriers :: DeclEvent -> Trav [DeclEvent] ()
```

in module `Gencot.Util.Types`. It adds all `DeclEvent`s with relevant type carriers to the list in the user state of the `Trav` monad. The handler is used for every `.c` file separately, the collected type carriers must be combined afterwards to get all type carriers in the Cogent compilation unit. This can be done by first combining the symbol tables as described in Section 3.5.4, and then using the union of all type carriers which are local, have internal linkage, or are still contained in the resulting combined table. This will avoid duplicate occurrences of type carriers.

All parameter declarations are ignored by the callback handler, since for every parameter declaration there must be a `DeclEvent` for the containing function, which is processed by the handler. Function and global variable declarations are also ignored, since they represent external functions and variables which are separately determined. Also, a function or variable declared in one source file may be defined in another source file of the Cogent compilation unit and thus be not external. Type carriers where all associated types are primitive (including enum types) are also ignored, since they may neither contain derived types nor reference other type carriers.

Enum types, composite types and typedef names declared in system include files are omitted, since they will be determined using the type usage relation. To determine whether a type carrier is defined in a system include file, the simple heuristics is used that the source file name is specified as an absolute pathname

in the `nodeinfo`. System includes are typically accessed using absolute pathnames of the form `/usr/include/...` (after being searched by cpp). If include paths are specified explicitly for Gencot, system include paths must be specified as absolute paths whereas include paths belonging to the <package> must be specified as relative paths to make the heuristics work.

After the language-c analysis phase the declarations of all invoked external functions are determined as described in Section 3.7.3. If the associated syntactic type is a typedef name, it is resolved to a derived function type. Afterwards, all invoked external functions where not all associated syntactic types are primitive are added to the combined type carriers determined by the callback handler. No duplicate type carriers will result here since the combined symbol table contains every external declaration only once.

The resulting set of type carriers is transitively completed by adding all type carriers used by referencing them, as described above. This will add all required enum types, composite types and typedef names defined in system include files. Here, type carriers where all associated types are primitive are not omitted, since `gencot-exttypes` must translate all added type carriers. The same holds for enum types. Duplicates could occur if a type carrier is referenced in several different types, they must be detected and avoided. Since all type carriers correspond to syntactic entities in a source file, they can be identified and compared by their position and source file.

Translating Type Definitions

For all external enum types, composite types and typedef names in the resulting set `gencot-exttypes` translates the definition to Cogent as described in Sections 2.6. All types used in these definitions are fully resolved, as described above.

To support the full resolving of types, the normal monadic action `transGlobals` used by the main translation to Cogent described in Section 3.7.2 cannot be used. The resolving needs access to all type names directly referenced from the Cogent compilation unit to stop the resolving there. The translation of external types is implemented by the monadic action

```
transExtGlobals :: [String] -> [DeclEvent] -> FTrav [GenToplv]
```

where the string list contains the referenced type names.

3.7.5 Derived Types

The processor `gencot-dvdtypes` generates all Cogent type and function definitions for the derived types used in the C program. It is invoked as

```
gencot-dvdtypes [<parmod file> [<varlist file>]]
```

where the optional arguments are a file with parameter modification descriptions and with the names of external variables to be processed.

A C derived type, is a pointer type, array type or function type. As described in Section 2.6, Gencot generates type names for all derived types. For these names type definitions must be provided in Cogent. Additionally, Gencot generates abstract function definitions for function pointer and for array types.

Derived types are used in C in the form of type expressions, i.e., base types, result and parameter types are syntactically included in the type specification. With some exceptions, derived types may occur wherever a type may be used. Therefore, to determine all used derived types, Gencot determines all used type carriers as described for `gencot-exttypes` in Section 3.7.4. These include all type carriers used in translated C sources, in declarations of external functions and variables and those defined in system include files.

For the same reason as for `gencot-exttypes` Gencot also processes the syntactic part types of derived types, if they are again derived types.

Derived function types which are directly used for defining or declaring a function are not processed, since for them Gencot does not generate a type name, it translates them to a Cogent function type expression. If, however, a typedef name is defined for a function type, it could also be used to construct the derived pointer type. Therefore Gencot processes the corresponding pointer type.

Together, `gencot-dvdtypes` processes *all* type carriers for all associated syntactic part types which are a derived type, with the exception of syntactic types associated with a function.

Generating Type Definitions

For all derived types associated as syntactic part type with a type carrier in the used set, an abstract type definition must be generated (with the exception of the syntactic types of functions). Only one definition may be generated, although a derived type may be associated with several different type carriers.

Derived function types are translated by Gencot with respect to a parameter modification description (see Section 3.2). This implies, that the same derived type may be translated to different Cogent type names, depending on which parameters of nonlinear type are readonly. To take this into account, Gencot first generates the translated Cogent type names for all derived types associated as syntactic part type with a type carrier, respecting the available parameter modification descriptions. Then it generates a single abstract definition for every resulting type name.

As described in Section 2.6.6, pointer types to composite types and array types are translated to the name of the pointed type. Therefore no separate abstract type definition is required for them, these derived types are omitted by `gencot-dvdtypes`.

Generating the type definitions for derived types is implemented by the monadic action

```
genTypeDefs :: [String] -> [DeclEvent] -> FTrav [GenToplv]
```

in module `Gencot.Cogent.Translate`, where the list of `DeclEvent`s corresponds to the type carriers to be processed. The string list contains the type names referenced in the Cogent compilation unit, it is used to fully resolve all types in external type carriers.

Together with every definition for an array type or a function pointer type Gencot generates the abstract function definitions as described in Sections 2.6.4 and 2.6.5.

Processor `gencot-dvdtypesh`

Processor `gencot-dvdtypesac`

3.7.6 Declarations

The filter `gencot-deccomments` is used to provide the information about the positions of all declarations with external linkage in a translated file, as described in Section 3.3.4. This information is used to move declaration comments to the corresponding definitions, as described in Section 2.2.4.

Only global declarations are processed by this filter. Therefore the filter processes the list of `DeclEvents` returned by `getDeclEvents` (see Section 3.5.3). The predicate for filtering the list selects all declarations with external linkage (i.e. removes all object/function/enumeration/type definitions and all declarations with internal or no linkage). The main application case are functions declared as external, but moving the comments may also be useful for objects.

If a function or object with external linkage is declared and defined in the same file, the language-c analysis step replaces the declaration by the corresponding definitions in the semantic map, hence it will not be found and processed by `gencot-deccomments`. However, it is assumed that in this case the documentation is assocatied with the definition and needs not be moved from the declaration to the definition.

For processing the declarations we do not need a monad since every declaration can be processed on its own. This is implemented by the function

```
transDecl :: DeclEvent -> String
```

in module `Gencot.Text.Decls`. It is mapped over the list of `DeclEvents` and the resulting string list is output one string per line.

3.7.7 Parameter Modification

`parmod-gen`

The filter `parmod-gen` is used to generate the Json parameter modification description from a C source file. It may be invoked in two forms:

```
parmod-gen <source file name>
parmod-gen <source file name> close
```

The second form runs the filter in "closing mode".

In normal mode the filter processes all `DeclEvent`s in the source file which are function definitions or definitions for objects with function pointer (array) type. Every such definition is translated to an entry in the parameter modification description. A function definition is translated to the sequence of its own description entry and the entries for all invoked local function pointer (array)s and all parameters for which the type is a function or function pointer (array) directly including the derived function type.

The filter processes every definition of a composite type in the source file and translates all its members with function pointer (array) type directly including the derived function type to the corresponding description. The filter processes every type name definition in the source file where the definig type is

a function (pointer (array)) type directly including the derived function type, and translates it to the corresponding description.

The descriptions of functions, their function parameters, composite type members, and typedef names are intended to be used by `gencot-translate` when it translates the function and type definitions. The descriptions of invoked local function pointer (array)s are intended for evaluating dependencies as described in Section 3.2.

In closing mode the filter processes all `DeclEvent`s in the source file and in all included files which are declarations for functions or function pointer (array)s. Every declaration is translated to an entry in the parameter modification description. These descriptions are only intended for "closing" the dependencies for the evaluation.

After analysing the C code as described in Section 3.5 the call graph is generated as described in Section 3.5.11. Then another traversal is performed using the `CTrav` monad and `runWithCallGraph` with action

```
transGlobals :: [DeclEvent] -> CTrav Parmods
```

defined in module `Gencot.Json.Translate`. The resulting list of JSON objects is output as described in Section 3.2.4.

Processor `parmod-externs`

The processor `parmod-externs` is used to generate the parameter modification descriptions intended to be used by `gencot-externs` (see Section 3.7.3) and `gencot-exttypes` (see Section 3.7.4). It is invoked as

```
parmod-externs [<varlist file>]
```

where the optional argument is a list of names of external variables to be processed.

Since all these parameter modification descriptions are already generated by `parmod-gen`, the processor only generates a list of function identifiers, as described in Section 3.2.2. Every identifier is output in a separate line. The identifiers can be used to specifically select and manually process only those descriptions which are actually required by `gencot-externs` and `gencot-exttypes`.

However, only identifiers for those descriptions are listed, which are not yet needed by `gencot-translate` to translate the source files of the Cogent compilation unit. The idea is to merge these with the descriptions selected by the listed identifiers to determine the descriptions needed by `gencot-externs` and `gencot-exttypes`.

The processor `gencot-externs` processes declarations of external invoked functions. It needs parameter modification descriptions for these functions and for their parameters of function (pointer (array)) type. The descriptions for the parameters are needed by `gencot-externs` to translate the parameter type as part of the generated abstract Cogent function definition.

Although `gencot-externs` only processes complete function declarations of non-variadic external functions, `parmod-externs` also lists identifiers for incompletely declared functions and for variadic functions, if they are invoked in the Cogent compilation unit. This is intended as an information for the developer who has to translate invocations of such functions manually. All these external

invoked functions are determined with the help of the call graph in the same way as described for `gencot-externs` in Section 3.7.3.

The processor `gencot-externs` also processes external variables, if they are invoked or listed explicitly by the developer. Parameter modification descriptions are only relevant for variables of function pointer (array) type. There may be such variables which are not invoked, but explicitly specified by the developer. For this reason `parmod-externs` also reads the list of external variables to be processed in addition to the invoked variables.

External invoked composite type members are not included, since they will be added through the used types below. To invoke a composite type member an object must exist which is a type carrier for the composite type. The parameter modification description identifier for the member will be listed when this type carrier or a typedef referenced by it is processed.

The processor `gencot-exttypes` processes derived types of all type carriers and translates enum types, composite types and typedef names defined in system include files. It needs parameter modification descriptions for both cases.

Since parameter modification descriptions must be associated with a function identifier, they cannot be used for arbitrary types. They are only used for composite type members with a function pointer (array) type and for typedef names where the defining type is a function (pointer (array)) type. In both cases the derived function type must be syntactically included. If it is specified using a typedef name, the parameter modification description is only used for translating the corresponding type definition.

Accordingly, `parmod-externs` lists for all processed type carriers the function ids of all composite type members and of all typedef names which have a type of this kind. These descriptions cover all cases needed by `gencot-exttypes`.

The type carriers to be processed are determined in the same way as by `gencot-exttypes`. Note that the external invoked functions and variables are already included there, however, those with only primitive types are omitted. Here we also include them, since `gencot-externs` processes them. For external function pointer (array)s descriptions for parameters are not listed, since for the parameter types of function pointers no parameter modification descriptions are used by Gencot, they are always translated assuming as default that all nonlinear parameters are modified.

Descriptions for composite types and typedef names need only be processed if they are defined in a system include file. In all other cases they are already needed by `gencot-translate` for translating the source files of the Cogent compilation unit and are omitted here.

For the determined type carriers `parmod-externs` lists the identifiers of the parameter modification descriptions for the following entities:

- declarations of external invoked functions, together with all their parameters, if the parameter type is a function (pointer (array)) type directly specifying the function type,
- declarations of external invoked or listed function pointer (array)s, if the declared type directly specifies the function type,
- members of all composite types defined in system include files, if their type resolves to a function pointer (array) type.

- typedef names defined in a system include file, if they resolve to a function (pointer (array)) type.

Note that typedef names and member types in system include files need not specify a function type directly for being listed, since types in external type carriers are fully resolved by `gencot-exttypes` and the resulting type is needed to translate the type definition.

3.8 Other Components

The following auxiliary Gencot components exist which do not process C source code:

`parmod-proc` processes parameter modification descriptions in JSON format (see Section 3.2).

3.8.1 Parameter Modification Descriptions

Processing parameter modification descriptions is implemented by the filter `parmod-proc`. It reads a parameter modification description from standard input as described in Section 3.2.4. The first command line argument acts as a command how to process the parameter modification description. The filter implements the following commands with the help of the functions from module `Gencot.Json.Process` (see Section 3.2.6):

`check` Verify the structure of the parameter modification description according to Section 3.2.1 and lists all errors found (not yet implemented).

`unconfirmed` List all unconfirmed parameter descriptions using function `showRemainingPars`.

`required` List all required invocations by their function identifiers, using function `showRequired`.

`funids` List the function identifiers of all functions described in the parameter modification description.

`sort` Takes an additional file name as command line argument. The file contains a list of function identifiers. The input is sorted using function `sortParmods`. This command is intended to be applied after the `merge` command to (re)establish a certain ordering.

`filter` Takes an additional file name as command line argument. The file contains a list of function identifiers. The input is filtered using function `filterParmods`.

`merge` Takes an additional file name as command line argument. The file contains a parameter modification description in JSON format. Both descriptions are merged using function `mergeParmods` where the first parameter is the description read from standard input. After merging, function `addParsFromInvokes` is applied. Thus, if the merged information contains an invocation with more arguments than before, the function description is automatically extended.

`eval` Using the functions `showRemainingPars` and `showRequired` it is verified that the parameter modification description contains no unconfirmed parameter descriptions and no required dependencies. Then it is evaluated using function `evalParmods`. The resulting parameter modification description contains no parameter dependencies and no invocation descriptions. It is intended to be read by the filters which translate C function types and function definitions.

All lists mentioned above are structured as a sequence of text lines.

If the result is a parameter modification description in JSON format it is written to the output as described in Section 3.2.4.

The first three commands are intended as a support for the developer when filling the description manually. The goal is that for all three the output is empty. If there are unconfirmed parameters they must be inspected and confirmed. This usually modifies the list of required invocations. They can be reduced by generating and merging corresponding descriptions from other source files.

3.9 Putting the Parts Together

The single filters and processors of Gencot are combined for the typical use cases in the shell scripts `gencot` and `parmod`.

3.9.1 The `gencot` Script

The intended use of filter `gencot-remcomments` is for removing all comments from input to the language-c parser. This input always consists of the actual source code file and the content of all included files. The simplest approach would be to use the language-c preprocessor for it, immediately before parsing.

However, it is easier for the filter `gencot-rempp` to remove the preprocessor directives when the comments are not present anymore. Therefore, Gencot applies the filter `gencot-remcomments` in a separate step before applying `gencot-rempp`, immediately after processing the quoted include directives by `gencot-include`.

The filters `gencot-selcomments` and `gencot-selpp` 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.

The `gencot` command supports the followin options:

`-I` used to specify directories where included files are searched, like for `cpp`. The option can be repeated to specify several directories, they are searched in the order in which the options are specified.

`-G`

`-C`

3.9.2 The `parmod` Script

Chapter 4

Application

The goal of applying Gencot to a C <package> is to translate a subset of the C sources to Cogent, resulting in a Cogent compilation unit which can be separately compiled and linked together with the rest of the <package> to yield a working system. The Cogent compilation unit is always a combination of one or more complete C compilation units, represented by the corresponding `.c` files.

We call this subset of `.c` files the "translation base". Additionally, all `.h` files included directly or indirectly by the translation base must be translated. Together, we call these source files the "translation set". Every file in the translation set will be translated to a separate Cogent source file, as described in Section 2.1.3. Additional Cogent sources and other files will be generated from the translation set to complete the Cogent Compilation Unit.

Since there is only one Cogent compilation unit in the package, we sometimes use the term "package" to refer to the Cogent compilation unit.

The Cogent compilation unit is defined by a "unit file". It contains the names of all files comprising the translation base, every name on a separate line.

4.1 Preparing to Read the Sources

4.2 Building Parmod Descriptions

The goal of this step is to create the parameter modification descriptions (see Section 2.10) for the C sources. Basically, the descriptions must be determined for all functions defined in the translation set. Usually function definitions only reside in `.c` files, but there are C <packages> which also put some function definitions in `.h` files.

Additionally, parmod descriptions are required for all functions invoked but not defined in the translation set ("external functions"). Since a parmod description is always derived from the function definition, this implies that a superset of the translation set must be processed in this step. The strategy described here tries to keep this superset minimal.

If for a function no definition is available (which is the case for function pointers and for functions defined outside the C <package>), Gencot only generates a description template which must be filled by the developer. In the

other cases Gencot creates a description in a best effort approach, which must be confirmed be the developer.

The automatic parts of this step are executed with the help of the script command `parmod` (see Section 3.9.2).

4.2.1 Describing Defined Functions

The descriptions for the functions defined in the translation set are determined iteratively in a first substep. The result is a single parmod description file in json format. It is extended in every iteration and finally evaluated.

To start, for every file in the translation base and for every other file in the translation set which contains function definitions, the command

```
parmod init
```

is used to create a parmod description file. These files are then merged using the command

```
parmod mergin
```

to yield the initial working file for the iterations.

In each iteration the descriptions must be manually confirmed (using a text editor) until the command

```
parmod show
```

does not signal any remaining unconfirmed descriptions. If it then does not signal any required invocations which are defined in the <package>, the iterations end. Otherwise, the developer must search for all files in the <package> where required invocations are defined (these files will not belong to the translation set) and generate parmod descriptions for them using `parmod init`. These descriptions are then added to the working file using the command

```
parmod addto
```

and the next iteration is performed.

When the iterations end, there may still be remaining required invocations. These are invocations of function pointers or functions defined outside the <package>. To generate the description templates for them the command

```
parmod close
```

is applied to all C sources to which the command `parmod init` has been applied previosly. The resulting parmod description template files are merged using `parmod mergin` and the resulting description template file is added to the working file using `parmod addto`. The command `parmod close` generates templates for all functions and function pointers for which a declaration is visible in the C source. Since for every function invoked in a C source a declaration must be visible, the working file will not have required invocations after this step. Since description templates introduce no additional required invocations, after a final confirmation step the working file is completed.

Instead of merging the parmod description template files generated by `parmod close`, they can also be added separately to the working file. Often, a C source

has much more visible declarations than it needs. Therefore it may cover required invocations of other files. This way it may be the case that not all single description template files must be generated to complete the working file.

Evaluating the completed working file using the command

```
parmod eval
```

will eliminate all transitive dependencies and yields the final parmod description file. It contains at least the descriptions for all functions defined in the translation set.

4.2.2 Describing External Functions

The descriptions for the invoked external functions are determined in a second substep. It results in a separate parmod description file.

The invoked external functions must be determined from the translation set, however, their descriptions must be generated from their definitions, which are outside the translation set. Both is done by the command

```
parmod extern
```

applied to the unit file and a parmod description file which must contain the descriptions for a superset of the invoked external functions. This parmod description file must be prepared in advance.

One possibility to prepare the file is to process all C source files which do *not* belong to the translation set with `parmod init` and merge the results. Since the functions defined outside the <package> are not contained, additionally the C source files which *do* belong to the translation set must be processed with `parmod close` and the resulting description templates merged to the previous results.

However, this implies that all files in the <package> must be prepared for parsing as described in Section 4.1. To avoid this, the file can be build by trying `parmod extern` with an empty parmod description (which is an empty JSON list code[]). It will signal all missing invoked functions not defined in the translation set. From this list the defining files (or declaring files in the case of a function defined outside the <package>) can be identified and used to build the description file passed to `parmod extern`.

Instead of starting with an empty description, all files created by `parmod init` in the first substep (see Section 4.2.1) can be merged and used as a starting point.

When the descriptions resulting from `parmod init` and the description templates resulting from `parmod close` are merged, care must be taken that a description is not replaced by a template for the same function. This may happen, since a function may be defined in one `.c` file but only declared in another, then both the description and the template will be present. The command `parmod mergin` always selects the description or template with less unconfirmed parameters. If both have the same number of unconfirmed parameters, it selects the description or template from its first argument file.

A fresh template generated by `parmod close` never has more confirmed parameter descriptions than a fresh description generated by `parmod close` for the same function. Thus, descriptions are selected over templates if first all

descriptions are merged, then the templates are merged into the result, with the file containing the templates being the second argument to `parmod mergin`.

When `parmod extern` has successfully been executed, the result must be iteratively completed in the same way as in Section 4.2.1. Initially, however, the result of the first substep should be added using `parmod addto`, since it may contain additional descriptions not needed in the first substep. All descriptions added this way are already fully confirmed and evaluated. Only if afterwards there are remaining required invocations, the file must be iteratively completed as in the first substep.

When the file has been completed, it must be evaluated using `parmod eval`.

4.3 Automatic Translation to Cogent

The goal of this step is to translate a single C source file which is part of the translation set to Cogent. This is done using Gencot to translate the file to Cogent with embedded partially translated C code. Afterwards the embedded C code must be translated manually, as described in Section 4.5.

The translated file is either a `.c` file which can be seperately compiled by the C compiler as a compilation unit, or it is a `.h` file which is a part of one or more compilation units by being included by other files. In both cases the file can include `.h` files, both in the package and standard C include files such as `stdio.h`.

To be translated by Gencot the C source file must first be prepared for being read by Gencot as described in Section 4.1. Then the parameter modification descriptions for all functions and function types defined in the source file must be created and evaluated as described in Section 4.2.1.

4.3.1 Translating Normal C Sources

A `.c` file is translated using the command

```
gencot [options] cfile file.c [<parmod>]
```

where `file.c` is the file to be translated and `<parmod>` is the file containing the parameter modification descriptions. If it is not provided, Gencot assumes for all functions and function types, that linear parameters may be modified. The result of the translation is stored as `file.cogent` in the current directory.

An `.h` file is translated using the command

```
gencot [options] hfile file.h [<parmod>]
```

where `file.h` is the file to be translated and `<parmod>` is as above. The result of the translation is stored as `file-incl.cogent` in the current directory.

4.3.2 Translating Configuration Files

Special support is provided for translating configuration files. A configuration file is a `.h` file mainly containing preprocessor directives for defining flags and macros, some of which are deactivated by being "commented out", i.e., they are preceded by `//`. If such a file is translated using the command

```
gencot [options] config file.h [<parmod>]
```

it is translated like a normal `.h` file, but before, all `//` comment starts at line beginnings are removed, and afterwards the corresponding Cogent comment starts `--` are re-inserted before the definitions.

4.4 Manual Adaptation of Data Types

In some cases the mapped data type definitions generated by Gencot according to Section 2.6 must be manually adapted by editing the corresponding `x-incl.cogent` files. Here we describe typical cases where an adaptation is necessary.

4.4.1 NULL Pointers

If a linear value is used in the program at a place where in the C program it is not known to be non-null, it's type must be changed from `T` to `(MayNull T)` (see Section 2.7.10).

A good hint for the property that a value may be null is if it is tested for being null in the C program. If not, however, this case may have been overlooked. This will show up if the value's type is not changed and during the manual translation of the function bodies an assignment of `NULL` must be translated for it.

A linear value may occur in the Cogent program as a function parameter, a function result, or a field in a record. In each case all occurrences of the value must be analyzed to find out whether the type must be changed or not.

If the value is a field in a record the null pointer may be represented by the field being taken. This is possible when it is statically known for each occurrence of the record in the Cogent program, whether the field value is null or not. Then the type of the record can be changed to the type with the field taken for all cases where the field value is null. A typical application case is when the field is initialized at a specific point in the program and never set to null afterwards.

If the value is the parameter or result of a function pointer or the element of an array, or it is referenced by a pointer, a type name must be introduced for the `MayNull` type instance and then used to consistently rename the type name generated by Gencot for the corresponding derived type. For example, if a C function pointer has type

```
int (*)(some_struct *p)
```

it is mapped by Gencot to have the abstract type

```
F_XStruct_Cogent_some_structX_U32
```

If parameter `p` may be null, its type should be changed to a new abstract type introduced by:

```
type F_XMayNull_Cogent_some_structX_U32
type MayNull_Cogent_some_struct =
  MayNull Struct_Cogent_some_struct
```

4.4.2 Grouping Fields in a Record

Sometimes in a C struct type several members work closely together by pointing to memory shared among them. Then in Cogent it would be possible to take the corresponding record fields separately, resulting in shared values of linear type. Instead, the fields should only be operated on by specific functions for which the correct handling of the fields can be proven. To avoid taking the fields separately, these functions should be defined on the record as a whole. Then it can be statically checked that the fields are never accessed or taken/put outside these functions.

To make this more explicit, the corresponding record fields can be grouped into an embedded unboxed record in Cogent. This is binary compatible if the members of the original struct are consecutive and in the same order and are no bitfields. For example in the record type `R` as defined in

```
type A
type R = {f1: A, f2: A, f3: A, f4: A, f5:A}
```

the fields `f2` and `f3` can be grouped by introducing the new record type `E`:

```
type A
type E = {f2: A, f3: A}
type R = {f1: A, embedded: #E, f4: A, f5:A}
```

This is translated by Cogent to an embedded struct.

Now operations working on some or all of the grouped fields can be defined on the type `E` instead of `R`, guaranteeing that the functions cannot access the other fields of `R`.

Note that it is still possible to take the fields separately by first taking `embedded` from `R` and then taking the fields from the taken value. This can now be prevented by checking that the field `embedded` is never accessed or taken/put directly in a value of type `R`.

Instead, using the conceptual operations `getref` and `modref` as defined in Section 2.7.6 it is possible to apply the functions defined on `E` in-place without copying the field values. This requires to manually define abstract functions

```
getrefEmbeddedInR: R! -> E!
modrefEmbeddedInR: all(arg,out). ModPartFun R E arg out
```

together with type `E`. They are implemented in C using the address operator `&` and should be the only ways how to access the field `embedded`.

To provide even more shielding, the type `E` can be defined as abstract, providing the definition as a struct in C:

```
typedef struct {A f2; A f3; } E;
```

Then it is guaranteed that the fields can never be accessed in the Cogent program. However, then also all function working on type `E` must be defined as abstract functions which are implemented in C.

4.4.3 Using Pointers for Array Access

A common pattern in C programs is to explicitly use a pointer type instead of an array type for referencing an array, in particular if the array is allocated on the heap. This is typically done if the number of elements in the array is not statically known at compile time. The C concept of variable length array types can sometimes be used for a similar purpose, but is restricted to function parameters and local variables and cannot be used for structure members.

In C the array subscription operator can be applied to terms of pointer type. The semantics is to access an element in memory at the specified offset after the element referenced by the pointer. This makes it possible to use a pointer as struct member which references an array as in

```
struct ip {... int *p, ...} s;
```

and access the array elements using the subscription operator as in `s.p[i]`.

Gencot translates the struct type to a record type of the form

```
type Struct_Cogent_ip = {... p: P_U32, ...}
```

and does not generate abstract functions which allow to treat `p` as an array, this must be done manually.

The main problem here is the unknown array size. In the C program, however, for working with the array it must be possible to determine the array size in some way at runtime. Here we distinguish two approaches how this is done.

Self-Descriptive Array

In the first case the array size can always be determined from the array content. Then the pointer to the first element is always sufficient for working with the array. The two typical patterns of this kind either use a stop element to mark the array end, such as the zero character ending C strings, or store the array size in a specific element or elements, such as in the header part of a network package represented as a byte array.

Working with the array is supported by manually defining and implementing an abstract data type for the array as follows. Let `tt` be a unique name for the specific kind of array (how its size is determined) and let `El` be the name of the array element type. We use `CArray_tt_El` as type name for the array, thus the type of `p` must be manually changed as follows:

```
type Struct_Cogent_ip = {... p: CArray_tt_El, ...}
```

The type `CArray_tt_El` is defined as

```
type CArray_tt_El = { ttArr: #El }
```

which has the same form as the Gencot pointer type for the element type.

For the type `CArray_tt_El` the polymorphic function `create` cannot be used since no instance has been generated by Gencot for it. Gencot cannot do this because the array size is unknown to it. When an array of this type is created, the size must be specified as an argument. Hence an abstract function of the form

```
create_CArray_tt_El : UNN -> EVTYPE(CArray_tt_El)
```

must be defined and implemented manually. It takes the actual array size (number of elements) as argument and is implemented in C using `calloc` with the array size and the size of the element type. The type `UNN` must be chosen by the developer so that it can represent all sizes which are used for the array. It is also used for the index values below. The type `EVTYPE(CArray_tt_El)` denotes, as usual, the type of the uninitialized array, realized by marking its single field `ttArr` as taken. Corresponding abstract functions for initializing and freeing the array elements must be provided manually.

For disposing values of the type `CArray_tt_El` the usual polymorphic function `dispose` can be used, Gencot is able to automatically provide a C correct implementation for all types for which it is used.

The element access functions for single elements should be defined as instances of the operations for accessing parts of structured values described in Section 2.7.6 in the specific form for arrays as described in Section 2.7.12:

```
getCArray_tt_El : (CArray_tt_El!,UNN) -> Option El!
setCArray_tt_El : (CArray_tt_El,UNN,El) -> (CArray_tt_El,())
exchngCArray_tt_El : (CArray_tt_El,UNN,El) -> (CArray_tt_El,El)
modifyCArray_tt_El : all(arg:<D,out).
  ModFun CArray_tt_El (UNN, ModFun El arg out, arg) out
getrefCArray_tt_El : (CArray_tt_El!,UNN) -> Option P_El!
modrefCArray_tt_El : all(arg:<D,out).
  ModFun CArray_tt_El (UNN, ModFun P_El arg out, arg) out
```

where `setCArray_tt_El` can only be defined if the element type `El` is discardable.

Externally Described Arrays

In the second case the array size is determined by additional information separate from the pointer to the array. Either the size is specified as an integer value as in

```
struct ip {... int *p, int psize; ...} s;
```

or it is specified by a second pointer, e.g., pointing to the last element as in

```
struct ip {... int *p, int *pend; ...} s;
```

In general there may be additional information, such as pointers into the array which are used to reference "current" positions in the array. We assume that all this information is provided by a sequence of members in the surrounding struct:

```
struct ip {...; t1 m1;...tn mn; ...} s;
```

These members can be grouped into an embedded struct as described in Section 4.4.2. If they are consecutive and are grouped in the same order the modified record type should be binary compatible. We propose to name the embedded record type `SArray_tt_El` ("structured array") in analogy to the array type name for self-descriptive arrays. It can either be defined as a record type in Cogent:

```
type SArray_tt_El = { m1: T1, ... Mn: Tn }
```

or, providing additional shielding as a wrapped abstract type

```
type SArray_tt_El = { ttSArr: #USArray_tt_El }
```

with a C definition:

```
typedef struct { t1 m1;...tn mn; } USArray_tt_El;
```

Note that in both cases `EVTYPE(SArray_tt_El)` yields a usable empty-value type.

Now the single fields in the original structure can be replaced by a field of the embedded structured array type:

```
type Struct_Cogent_ip = {... a: #SArray_tt_El, ...}
```

To access and modify the group in the struct the abstract functions

```
getrefAInStruct_Cogent_ip:
  Struct_Cogent_ip! -> SArray_tt_El!
modrefAInStruct_Cogent_ip: all(arg,out).
  ModPartFun Struct_Cogent_ip SArray_tt_El arg out
```

must be defined and used in the same way as described in Section 4.4.2.

Initialization and clearing functions for `SArray_tt_El` must be implemented manually, they always need the heap for allocating or disposing the actual array.

The element access functions for single elements can be defined as abstract instances of the operations for accessing parts in the same way as for self-descriptive arrays, although the implementations will differ because they have to take into account the fields of `SArray_tt_El`. Usually, additional functions will be required for working with values of type `SArray_tt_El`, such as for moving an internal pointer to a "current" element. If such a function modifies some of the fields of `SArray_tt_El` it must be defined as a modification function which can be applied with the help of `modrefAInStruct_Cogent_ip`.

4.5 Manual Translation of C Function Bodies

The manual part of the translation is mainly required for the bodies of all C functions defined in the C source file. A `.h` file normally does not contain function or object definitions, then it may be the case that no manual actions are necessary. But some `.h` files define functions, typically as `static inline`, then also their bodies must be translated manually.

The manual translation is done by modifying the `.cogent` file generated by the automatic translation in an editor. As described in Section 2.9.1, the result of automatically translating a function definition has the form

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

where the `<compound statement>` is plain C code with global names mapped to Cogent. The task of the manual translation is to replace the `<dummy result>` by a Cogent expression equivalent to the `<compound statement>`.

In the following sections we provide some rules and patterns how to translate typical C constructs occurring in a `<compound statement>`. These rules and patterns are not exhaustive but try to cover most of the common cases.

4.5.1 General

The basic building blocks of C function bodies are declarations and statements. Here we only cover declarations of local variables. Such a declaration specifies a name for the local variable and optionally an initial value.

A C statement causes modifications in its context. Cogent, as a functional language, does not support this concept, in particular, it does not support modifying the value of a variable. Cogent only supports expressions which functionally depend on their input, and it supports introducing immutable variables by binding them to a value. The main task of translating function bodies is to translate C statements to Cogent expressions.

The main idea of the translation is to translate a C variable which is modified to a sequence of bindings of Cogent variables to the values stored in the C variable over time. Everytime when a C statement modifies a variable, a new variable is introduced in Cogent and bound to the new value. A major difference of both approaches is that in Cogent the old variable and its value are still available after the modification. To prevent this, we use the same name for both variables. Then the new variable "shadows" the old one in its scope, making the old value unaccessible there. For example the C code

```
int i = 0; i = i+1; ...
```

is translated to

```
let i = 0 in let i = i+1 in ...
```

where the single C variable `i` is translated to two Cogent variables which are both named `i`.

Statements

A C statement in a function can modify several of the following:

- function parameters
- local variables
- global variables

All of them are specified by an identifier which is unique at the position of the statement. The modification may replace the value as a whole or only modify one or more parts in the case of a structured value.

As a first step modifications of global variables must be eliminated by passing all such variables as additional parameters to the function and returning them as additional component in the result. Then a modification of a global variable becomes a modification of a function parameter and only the first two cases remain.

After this step the effect of the modification caused by a C statement can be described by a set of identifiers of all modified parameters and local variables together with the new values for them. Syntactically this corresponds to a Cogent *binding* of the form

```
(id1,...,idn) = expr
```

where `expr` is a Cogent expression for the tuple of new values for the identifiers.

Pointers

Cogent treats C pointers in a special way as values of "linear type" and guarantees that no memory is shared among different values of these types. More general, all values which may contain pointers (such as a struct with some pointer members) have this property. All other values are of "nonlinear type" and never have common parts in C.

If a C program uses sharing between values of linear type, it cannot be translated directly to Cogent. We use the following approach for a translation of such cases.

If several parameters of a function may share common memory, they are grouped together to a Cogent record or abstract data type and treated as a single parameter. All functions which operate on one of the values are changed to operate on the group value. Then no sharing occurs between the remaining function parameters.

If several local variables share common memory they can be treated in the same way.

If a variable shares memory with a parameter this solution is not applicable. In this case the variable must be eliminated. This is easy if the variable is only used as a shortcut to a part of the parameter value, then it can be replaced by explicit access to the part of the parameter. For example, in the C function

```
void f (struct{int i; x *p;} p1) {
  x *v = p1.p; ...
}
```

the occurrences of variable `v` can be replaced by accesses to `p1.p`. In other cases individual solutions must be found. Note that parameters and variables of nonlinear type never cause such problems.

After these steps no sharing occurs among the parameters and local variables in the function. This implies that a C statement can only modify parameters and variables for which the identifiers occur literally in the statement source code text. Thus it is possible to determine the effect of the modification caused by a C statement syntactically from the statement.

Variable Declarations

An initializer `init` in a variable declaration `t v = init;` is either an expression or an initializer for a struct or an array. The C declaration can be rewritten as

```
t v;
v = init;
```

with a separate statement for initializing the variable. In the first case this is valid C code. In the other cases the statement is not, but we will translate it according to the intended semantics. For a struct type Cogent provides corresponding expressions for unboxed records. For other types Gencot provides its initialization functions described in Section 2.7.5 and `design-operations-array`, or the initialization can be done by several applications of operation `set` (see Section 2.7.6).

A declaration without an initializer should always be followed by an assignment to it before it is accessed. If not we insert an assignment of a default value before the first access.

Then the declarations need not be translated to Cogent, since in Cogent a new variable is introduced whenever the C variable is modified by a statement. Therefore, only the C statements need to be translated to Cogent.

4.5.2 Expressions

C statements usually have C expressions as syntactic parts. For translating C statements the contained C expressions must be translated. C expressions have a value but may also cause modifications as side effects. Especially, in C an assignment is syntactically an expression. C expressions are translated depending on whether they have side effects or not.

Expressions without Side Effects

C expressions without side effects are literals, variable references, member accesses of the form `s.m` or `s->m`, index expressions of the form `a[e]`, applications of binary operators of the form `e1 op e2`, applications of the unary operators `+,-,!,` of the form `op e`, and function call expressions of the form `f(e1,...,en)`, if all subexpressions `s,a,e,f,e1,...,en` have no side effects and function `f` does not modify its parameter values.

These expressions are translated to Cogent expressions in a straightforward way with the same or a similar syntax. A member access `s->m` is translated as `s.m`. An index expression `a[e]` is translated as function call `getArr(a,e)`. Note, that some C operators have a different form in Cogent:

```
C  Cogent
!=   /=
^    .^.
&    .&.
|    .|.
!    not
~    complement
```

Member accesses of the form `s->m` (or written `(*s).m`) and index expressions `a[e]` can only be translated in this way if the container value `s` or `a`, respectively, is translated to a readonly value in Cogent. If it is a parameter, variable, or record field, it may have been defined as readonly. Otherwise, it can be made readonly in the expression's context by applying the bang operator `!` to it at the end of the context:

```
... s.m ...  !s
```

The resulting expressions are also of readonly type. For expressions of nonlinear type this is irrelevant, for expressions of linear type it implies that they cannot be modified. If they are used in C by modifying them, they must be translated in a different way. For example in the C code fragment

```
*(s->p) = 5
```

the expression `s->p` denotes a pointer which is modified, therefore it cannot be translated to `s.m` where s is readonly.

For resultig expressions of linear types, using the bang operator `!` also means that the readonly result cannot escape from the banged context, it can only be used inside the context.

Translation of expressions using the address operator `&` are described in Section 4.5.6.

Dereferencing (application of the indirection operator `*` to) a pointer is translated depending on the type of value referenced by the pointer. If it points to a function, `*p` is translated as `fromFunPtr(p)` (See Section 2.7.9).

If it points to a primitive type, an enum type, or again a pointer type, `*p` is translated as `getPtr(p)` (See Section 2.7.8). In this case `p` must be readonly as above, and the result is readonly. If it should be modified it must be translated differently.

Otherwise it points to a type which is mapped to Cogent as a record or abstract type. Then `*p` is translated as `p`.

Expressions using the C operators `sizeof` or `_Alignof` cannot be translated to Cogent. Usually, an abstract function implemented in C is required here.

Expressions with Side Effects

We translate an expression with side effects to a Cogent binding of the form `pattern = expr`. Here, the `pattern` is a tuple of variables `(v,v1,...,vn)`. The variable `v` is a new variable which is not already bound in the context of the expression, it is used to bind the result value of the expression. The other variables are the identifiers of all parameters and local variables modified by the expression. Since we presume that the C expression has side effects, there is at least one such variable. The `expr` is a Cogent expression for a corresponding tuple consisting of the result value of the C expression and the new values of the identifiers modified by the C expression.

Expressions with side effects are applications of the increment and decrement operators `++,-`, assignments, and invocations of functions which modify one or more parameter values.

Applications of increment and decrement prefix operators must be rewritten in C using assignments. Assignments using assignment operators other than `=` must be rewritten in C using the `=` operator. After these steps the only remaining expressions with side effects are assignments using `=`, increment and decrement postfix operators, and invocations of functions which modify parameter values.

If for such an expression all subexpressions are without side effects, they are translated as follows.

The translation of an assignment expression of the form `lhs = e` depends on the form of `lhs`. If it is a single identifier `v1` (name of a parameter or local variable), it is translated as

```
(v,v1) = let v = expr in (v,v)
```

where `expr` is the translation of `e`.

Note that this code is illegal in Cogent, if `expr` has a linear type, since it uses the result value twice. However, this is a natural property of C code, where an assignment is an expression and the assigned value can be used in the context. For example, the C code fragment `f(p = q)` assigns the value of `q` to `p` and also passes it as argument to function `f`.

There are several ways how to cope with this situation. In the simplest case, the outer variable `v` is never used in its scope, then the double use of the value can be eliminated by simplifying the Cogent binding to the form

```
v1 = expr
```

This is typically the case if the assignment is used as a simple statement, where its result value is discarded. Most assignments in C are used in this way.

Otherwise it depends on how the variable `v` is used in its scope. In some cases it may be possible to replace its use by using `v1` instead, then it can be eliminated in the same way as above. If that is not possible, the C program uses true sharing of pointers, then the code cannot be translated to Cogent and must be translated using abstract functions.

If `lhs` is a logical chain of `n` member access, index, and dereferencing expressions starting with identifier `v1` (name of a parameter or local variable), the most general translation is

```
(v,v1) =
  let v = expr
  in (v,fst(modify1(v1,(modify2,(...(modifyn-1,(set,v))...)))))
```

where `expr` is the translation of `e` and the sequence of `modifyi` functions is determined by the chain of access expressions and `set` is an instance of the operation `set` (see Section 2.7.6). Again, the result of `expr` is used twice, the same considerations as above apply if it has a linear type.

For example the C assignment expression

```
s->a[i]->x = 5
```

is translated according to this rule to the Cogent binding

```
(v,s) = let v = 5
  in (v,fst(modrefFldA(s,(modifyArr,(i,exchngFldX,v)))))
```

where `modrefFldA` and `exchngFldX` are abstract modification functions for the fields `a` and `x`, respectively, as described in Section 2.7.11. `exchngFldX` is used instead of `setFldX` because `modifyArr` expects a modification function where the additional input and result have the same type.

No other cases for `lhs` are valid in a C assignment.

If `lhs` logically starts with a chain of member accesses `v1->m1->...->mn...` an alternative translation using the Cogent take and put operations is the binding

```
(v,v1) =
  let v1{m1=m1{m2=...mn-1{mn}...}}
  and (v,mn) = expr
  in (v,v1{m1=m1{m2=...mn-1{mn=mn}...}})
```

where `(v,mn) = expr` is the binding to which `mn... = e` is translated, if `mn` is assumed to be a local variable. For example, a corresponding translation of `*(s->m1->m2) = 5` is

```
(v,s) =
  let s{m1=m1{m2}}
  and (v,m2) = let v = 5 in (v,setPtr(m2,v))
  in (v,s{m1=m1{m2=m2}})
```

This approach avoids the need to manually define and implement the functions `modifyFldM...`.

An application of an increment/decrement postfix operator `ss` where `s` is `+` or `-` has the form `lhs ss`. If `lhs` is a single identifier `v1` this identifier must have a numerical type and the expression is translated to the binding

```
(v,v1) = (v1,v1 s 1)
```

Using `v1` twice is always possible here since it has nonlinear type. As an example, `i++` is translated to

```
(v,i) = (i,i+1)
```

If `lhs` is a logical chain of `n` member access, index, and dereferencing expressions starting with identifier `v1` this identifier must have linear type and the most general translation is the binding

```
(v,v1) =
  let v = tlhs !v1
  in (v, fst(modify1(v1,(modify2,(...(modifyn-1,(set,v s 1))...))))
```

where `tlhs` is the translation of `lhs` when `v1` is readonly and `modifyi` and `set` are as for the assignment. Here `v1` is needed twice, first for retrieving the old numerical value `v` and afterwards to set it to the incremented/decremented value. Since `v1` is linear the old value must be retrieved in a readonly context and the modification must be done seperately. The double use of `v` is always possible since it has numeraical type.

If `lhs` logically starts with a chain of member accesses `v1->m1->...->mn...` an alternative translation using the Cogent take and put operations is the binding

```
(v,v1) =
  let v = tlhs !v1
  and v1{m1=m1{m2=...mn-1{mn}...}}
  in (v, v1{m1=m1{m2=...mn-1{mn=expr}...}})
```

where `expr` is as above for the assignment using `(v s 1)` as the new value.

For example, a corresponding translation of `(*(s->m1->m2))++` is

```
(v,s) =
  let v = getPtr(s.m1.m2) !s
  and s{m1=m1{m2}}
  in (v,s{m1=m1{m2=setPtr(m2,v+1)}})
```

A C function which modifies parameter values is translated by Gencot to a Cogent function returning the tuple `(y,p1,...,pn)` of the original function result `y` and the modified parameter values `p1,...,pn` (which must be pointers). A C function call `f(...)` is translated depending on the form of the actual arguments passed to `f` for the modified parameters.

If all such arguments are identifiers (names of parameters and local variables) the function call is translated to the binding

```
(v,v1,...,vn) = f(...)
```

where `v1,...,vn` are the identifiers passed to the parameters modified by `f` in the order returned by `f`.

If some of the arguments are member access chains of the form `vi->mi1->...->miki` they must be translated using the Cogent take and put operations as above to a binding of the form

```
(v,v1,...,vn) =
let v1{m11=...{m1k1}...}
and ...
and vn{mn1=...{mnkn}...}
and (v,m1k1,...,mnkn)=f(...)
in (v,v1{m11=...{m1k1=m1k1}...},
     ...
      vn{mn1=...{mnkn=mnkn}...})
```

where in the arguments of `f` the chains are replaced by their last member name `miki`.

For example, the function call `f(5,s->m,t->n,z)` where `f` modifies its second and third parameter, is translated to the binding

```
(v,s,t) =
let s{m}
and t{n}
and (v,m,n)=f(5,m,n,z)
in (v,s{m=m},t{n=n})
```

If function `f` modifies only one parameter `p` of Cogent type `P` its standard type generated by Gencot can be changed to the form of a modification function

```
f: ModFun P (...) Res
```

where `(...)` is the tuple of types of the other parameters and `Res` is the result type. Then for an arbitrary chain of member access, index, and dereferencing expressions used as actual argument for `p` the function call can be translated to a binding of the form

```
(v,v1) = let (v1,v) =
  modify1(v1,(modify2,(...(modifyn,(f,(a1,...,an)))...)))
  in (v,v1)
```

where the sequence of `modifyi` functions is determined by the chain of access expressions. Note that the order of the variables must be exchanged since the original result of `f` is the second component in the inner tuple pattern due to the way `ModFun` is defined.

For example, the function call `f(5,s->a[i]->x,z)` can be translated by first modifying the translation of `f` so that it takes as parameters instead of the tuple `(a,b,c)` the pair `(b,(a,c))`. Then a translation for the function call is

```
(v,s) = let (s,v) =
  modrefFldA(s,(modifyArrDflt,(i,modifyFldX,(f,(5,z)))))
  in (v,s)
```

Here function `modifyArrDflt` must be used since the additional input and output of `f` have different types. If index `i` is invalid `f` is never applied and a default value is used instead of its result.

Again, if the chain logically starts with member accesses, that part of the chain can be translated using take and put operations.

In all other cases the function `f` must be modified so that it takes the starting identifiers of the chains as arguments instead of the chains, then it can be translated as in the first case where all actual arguments are identifiers. Note that different translations of the function to Cogent may be required for translating different function calls.

Nested Expressions with Side Effects

If an expression contains subexpressions with side effects, these must be translated separately.

Let `e1,...,en` be the expressions with side effects directly contained in the expression `e` and let `p1 = expr1, ..., pn = exprn` their translations to Cogent bindings. Since in C the order of evaluation of the `e1,...,en` is undefined, we can only translate `e` if the subexpressions modify pairwise different sets of identifiers, i.e., the `p1,...,pn` are pairwise disjunct tuples. Let `x1,...,xn` be the first variables of the patterns `p1,...,pn` and let `w1,...,wm` be the union of all other variables in `p1,...,pn` in some arbitrary order. Let `e'` be `e` with every `ei` substituted by `xi`. Then `e'` contains no nested expressions with side effects and can be translated to Cogent according to the previous sections.

If `e'` has no side effects, let `expr` be its translation to a Cogent expression. Then the translation of `e` is the binding

```
(v,w1,...,wm) =
  let p1 = expr1 and ...
  and pn = exprn
  in (expr,w1,...,wm)
```

For example the expression `a[i++]` is translated according to this rule to

```
(v,i) =
  let (v,i) = (i,i+1)
  in (getArr(a,v),i)
```

If `e'` has side effects, let `(v,v1,...,vk) = expr` be its translation to a Cogent binding. Then the translation of `e` is the binding

```
(v,v1,...,vk,w1,...,wm) =
  let p1 = expr1 and ...
  and pn = exprn
  and (v,v1,...,vk) = expr
  in (v,v1,...,vk,w1,...,wm)
```

For example the expression `a[i++] = 5` is translated according to this rule to

```
(v,a,i) =
  let (v,i) = (i,i+1)
  and (w,a) = let w = 5 in (w,setArr(a,v,w))
  in (w,a,i)
```

which can be simplified in Cogent to the binding

```
(v,a,i) = (5,setArr(a,i,5),i+1)
```

Readonly Access and Modification of the Same Value

If a C expression uses and modifies parts of a linear value at the same time, a special translation approach is required. An example is the expression `r->sum = r->n1 + r->n2`. According to the rules above, translating the right hand side requires to make `r` readonly by applying the bang operator `!` which then prevents translating the expression as a whole, modifying `r`. There are three cases how to deal with this situation.

If all used parts of the value are nonlinear, they can be retrieved in a separate step using `!`, bound to Cogent variables and then used in the modification step. The translation of an expression `e` then has the general form

```
(v,v1,...,vn) =
  let (w1,...,wm) = ... !r1 ... !rk
  in expr
```

where `w1,...,wm` are auxiliary Cogent variables for binding the used values, `r1,...,rk` are all linear values from which parts are used in `e` and `(v,v1,...,vn) = expr` is the normal translation of `e` with all used parts replaced by the corresponding `wi`.

We used this approach for translating applications of postfix increment/decrement operations to complex expression, such as `(*(s->m1->m2))++`.

If the used values are themselves linear, such as in `r->p = f(r->p)` where `p` is a pointer, this approach cannot be used since in Cogent the binding `w = r.p !r` is illegal, `r.p` has a readonly linear type and is not allowed to escape from the banged context so that it can be bound to `w`.

If the used linear value is replaced by the modification, as in the example, it is possible to use the take and put operations. In a first step the used values are taken from the linear containers `r1,...,rk`, in the modification step they are put back in. The translation of `e` then has the general form

```
(v,v1,...,vn) =
  let r1{... = w1 ...} ...
  and rk{... = wk ...}
  in expr
```

where `expr` contains the necessary put operations for all `r1,...,rk`.

It may also be the case that the modification cannot be implemented by a combination of take and put operations, either because data types like arrays and pointers are involved, which are not translated to Cogent records, or because the modification causes sharing or discarding linear values. In both cases the modification cannot be implemented in Cogent, it must be translated by introducing an abstract function `fexpr` which implements the expression `e` as a whole. Then the translation has the form

```
(v,v1,...,vn) =
  fexpr(v1,...,vn,x1,...,xm)
```

where `x1,...,xm` are additional nonlinear values used by the expression.

Comma Operator

In C the expression `e1,e2` first evaluates `e1`, discarding its result and then evaluates `e2` for which the result is the result of the expression as a whole. Thus, the comma operator only makes sense if `e1` has side effects. Let `(v,v1,...,vn) = expr1` be the translation of `e1` to a Cogent binding.

If `e2` has no side effects and is translated to the Coent expression `expr2`, the expression `e1,e2` is translated to the Cogent binding

```
(v,v1,...,vn) =
  let (_,v1,...,vn) = expr1
  in (expr2,v1,...,vn)
```

Note that the translation is only valid if the result value of `e1` is not linear, since it is discarded. If `expr1` is a tuple, this can be simplified by omitting the first component. This avoids the discarding and may even remove a double use of a linear value in `expr2`.

If `e2` has side effects let `(w,w1,...,wm) = expr2` be the translations of `e2`. Then the translation of `e1,e2` is

```
(v,u1,...,uk) =
  let (_,v1,...,vn) = expr1
  and (w,w1,...,wm) = expr2
  in (w,u1,...,uk)
```

where `u1,...,uk` is the union of `v1,...,vn` and `w1,...,wm`.

Conditional Expression

A conditional expression `e0 ? e1 : e2` is translated as follows. Let `(x,x1,...,xn) = expr0`, `(y,y1,...,ym) = expr1`, `(z,z1,...,zp) = expr2` be the translations of `e0`, `e1`, `e2`, respectively. Then the translation of the conditional expression is

```
(v,v1,...,vk) =
  let (x,x1,...,xn) = expr0
  and (v,u1,...,uq) =
    if x then
      let (y,y1,...,ym) = expr1
      in (y,u1,...,uq)
    else
      let (z,z1,...,zp) = expr2
      in (z,u1,...,uq)
  in (v,v1,...,vk)
```

where `u1,...,uq` is the union of `y1,...,ym` and `z1,...,zp` in some order, and `v1,...,vk` is the union of `y1,...,ym` and `x1,...,xn` in some order.

Usually, the condition `e0` has no side effects, the the translation can be simplified to

```
(v,u1,...,uq) =
  if expr0 then
    let (y,y1,...,ym) = expr1
```

```
    in (y,u1,...,uq)
  else
    let (z,z1,...,zp) = expr2
    in (z,u1,...,uq)
```

If also `e1` and `e2` have no side effects the translation can be further simplified to the Cogent expression

```
if expr0 then expr1 else expr2
```

As an example the C expression `i == 0?  a = 5 :  b++` is translated to

```
(v,a,b) =
  if i == 0 then
    let a = 5
    in (a,a,b)
  else
    let (v,b) = (b,b+1)
    in (v,a,b)
```

However, these translations have a drawback. When Cogent translates the Cogent code back to C it translates an expression of the form

```
let v = if a then b else c
in rest
```

to a C statement of the form

```
if a then { v = b; rest}
else { v = c; rest}
```

duplicating the translated code for `rest`. The same happens for all bindings and expressions after the binding of `v` in the context of the `let` expression. If several such bindings to a conditional expressions are used this leads to an exponential growth in size of the C code.

To prevent this the conditional expression can be wrapped in a lambda expression in the form

```
let v = (\(x1,...,xn) => if a then b else c)
  (x1,...,xn)
in rest
```

where `x1,...,xn` are all Cogent variables used in the expressions `a, b, c`. Cogent translates this code to C without duplicating `rest`.

Therefore the conditional Cogent expressions in the translation of a conditional C expression should be wrapped in a lambda term whenever they are not the last binding in the translated function body or `rest` is only very small.

4.5.3 Statements

A C statement has no result value, it is only used for its side effects. Therefore we basically translate every statement to a Cogent binding of the form `pattern = expr` where pattern is a tuple of all identifiers modified by the statement.

However, a C statement can also alter the control flow in its environment, which is the case for `return` statements, `goto` statements etc. We treat this for `return` statements by adding two components to the `pattern`, so that the translation becomes

```
(v,c,v1,...,vn) = expr
```

As for expressions, the variable `v` is a new variable which is not already bound in the context of the expression, it is used to bind the result value of the surrounding function. Similarly, `c` is a new variable for the condition when to return immediately after the statement. The other variables are the identifiers of all parameters and local variables modified by the statement. Since a statement need not modify variables, the pattern may also be a pair `(v,c)`.

Simple Statements

A simple C statement consists of a C expression, where the result is discarded. Therefore it makes only sense if the C expression has side effects.

The binding in the translation of a simple C statement `e;` is mainly the binding resulting from the translation of the expression `e`, as described in Section 4.5.2. Let its translation be `(v,v1,...,vn) = expr`. Then the binding for the statement `e;` is

```
(v,c,v1,...,vn) =
  let (v,v1,...,vn) = expr
  in (dummy,False,v1,...,vn)
```

where `dummy` is the dummy expression `gencotDummy[Res]()` for the function result type `Res`, as described in Section 2.7.1. If the C code is correct, it can always be eliminated by simplifying the generated Cogent code for the function body.

The result `v` of `e` is discarded, the same considerations apply here as described for expressions using the comma operator in Section 4.5.2. Since no function result is provided by the simple statement, the outer variable `v` is bound to `()` as a placeholder.

If `expr` is a tuple `(e0,e1,...,en)` the binding can further be simplified to

```
(v,c,v1,...,vn) = (dummy,False,e1,...,en)
```

As an example, the translation of the simple statement `i++;` is

```
(v,c,i) = let (v,i) = (i,i+1) in (dummy,False,i)
```

which can be simplified to

```
(v,c,i) = (dummy,False,i+1)
```

The empty C statement `;` is translated as

```
(v,c) = (dummy,False)
```

Return Statements

A C return statement has the form `return;` or `return e;`. It always ends the control flow in the surrounding function body. The first form can only be used in functions returning `void`, these are translated to Cogent as returning `()` or a tuple of only the modified parameters.

The translation of a return statement of the form `return;` is the binding

```
(v,c) = ((), True)
```

If `e` has no side effects, the translation of statement `return e;` is

```
(v,c) = (expr, True)
```

where `expr` is the translation of `e`.

Otherwise, let `(v,v1,...,vn) = expr` be the translation of `e`. Then the translation of `return e;` is

```
(v,c,v1,...,vn) =
  let (v,v1,...,vn) = expr
  in (v, True, v1,...,vn)
```

which can be simplified if `expr` is the tuple `(e0,e1,...,en)` to

```
(v,c,v1,...,vn) = (e0,True,e1,...,en)
```

Conditional Statements

A conditional C statement has the form `if (e) s1` or `if (e) s1 else s2` where `e` is an expression and `s1` and `s2` are statements. Let `(v,c,v1,...,vn) = expr1` be the translation of `s1` and `(w,d,w1,...,wm) = expr2` be the translation of `s2`. If `e` has no side effects and translates to the Cogent expression `expr` we translate the second form of the conditional statement to the binding

```
(v,c,u1,...,uk) =
  if expr then
    let (v,c,v1,...,vn) = expr1
    in (v,c,u1,...,uk)
  else
    let (w,d,w1,...,wm) = expr2
    in (w,d,u1,...,uk)
```

where `u1,...,uk` is the union of `v1,...,vn` and `w1,...,wm` in some order. Note that this closely corresponds to the translation of the conditional expression in Section 4.5.2.

The first form of the conditional statement is translated as

```
(v,c,v1,...vn) =
  if expr then expr1
  else ((),False,v1,...vn)
```

If expression `e` has side effects the translation is extended as for the conditional expression.

As described for the conditional expression, the conditional Cogent expression on the right hand side of the binding can cause exponential growth of the generated C code. This can be prevented by wrapping it in a lambda expression.

As an example the translation of the conditional statement `if (i==0) return a; else a++;` is the binding

```
(v,c,a) =
  if i==0 then
    let (v,c) = (a,True)
    in (v,c,a)
  else
    let (w,d,a) = (dummy,False,a+1)
    in (w,d,a)
```

which can be simplified to

```
(v,c,a) =
  if i==0 then (a,True,a)
  else (dummy,False,a+1)
```

Compound Statements

A compound statement is a block of the form `{ s1 ... sn }` where every `si` is a statement or a declaration. Declarations of local variables are treated as described in Section 4.5.1: if they contain an initializer they are translated as a statement, otherwise they are omitted. This reduces the translation of a compound statement to the translation of a sequence of statements.

We provide the translation for the simplest case `{ s1 s2 }` where `s1` and `s2` are statements. The general case can be translated by rewriting the compound statement as a sequence of nested blocks, associating from the right.

Let `(v,c,v1,...,vn) = expr1` be the translation of statement `s1` and `(w,d,w1,...,wm) = expr2` be the translation of statement `s2`. Then the translation of `{ s1 s2 }` is

```
(v,c,u1,...,uk) =
  let (v,c,v1,...,vn) = expr1
  in if c then (v,c,u1,...,uk)
  else let (w,d,w1,...,wm) = expr2
  in (w,d,u1,...,uk)
```

where `u1,...,uk` is the union of `v1,...,vn` and `w1,...,wm` without the local variables declared in the block in some order.

The values of local variables declared in the block are discarded. If such variables have linear type, they must be allocated on the heap and disposed at the end of the block, as described in Section 4.5.6.

As an example, the compound statement `{ if (i==0) return a; else a++; b = a; }` is translated to the binding

```
(v,c,a,b) =
  let (v,c,a) =
    if i==0 then (a,True,a)
    else (dummy,False,a+1)
```

```
  in if c then (v,c,a,b)
  else let (w,d,b) = (dummy,False,a)
  in (w,d,a,b)
```

If `s1` contains no return statement the inner variable `c` is bound to `False` and the translation can be simplified to the binding

```
(v,c,u1,...,uk) =
  let (v,c,v1,...,vn) = expr1
  and (w,d,w1,...,wm) = expr2
  in (w,d,u1,...,uk)
```

As an example, the compound statement `{ int i = 0; a[i++] = 5; b = i + 3; }` is translated to the simplified binding

```
(v,c,a,b) =
  let (v,c,i) = (dummy,False,0)
  and (w,c,i,a) = (dummy,False,i+1,setArr(a,i,5))
  and (x,c,b) = (dummy,False,i+3)
  in (v,c,a,b)
```

which can be further simplified to

```
(v,c,a,b) =
  let i = 0
  and (i,a) = (i+1,setArr(a,i,5))
  and b = i+3
  in (dummy,False,a,b)
```

eliminating all occurrencies of `dummy` in the inner bindings.

A compound statement used as the body of a C function is translated to a Cogent expression instead of a Cogent binding. If the C function has result type `void` and modifies its parameters `pm1,...,pmn`, its translation is a Cogent function returning the tuple `(pm1,...,pmk)`. If the translation of the compound statement used as function body is the binding `(v,c,v1,...,vn) = expr`, the body is translated to the expression

```
let (v,c,v1,...,vn) = expr
in (pm1,...,pmn)
```

If the function modifies no parameters it returns the unit value `()`. Note that if the C function is correct, `v` is bound to `()`.

Here all local variables and the unmodified parameters are discarded. If local variables have linear type, they must be allocated on the heap and disposed at the end of the function, as described in Section 4.5.6. If an unmodified parameter has linear type, it must be disposed at the end of the function.

If the C function returns a value its translation is a Cogent function returning the tuple `(v,pm1,...,pmk)` where `v` is the original result value of the C function. Then the body is translated to the expression

```
let (v,c,v1,...,vn) = expr
in (v,pm1,...,pmn)
```

For Loops

While Loops

4.5.4 Function Pointer Invocation

In C a function pointer may be `NULL`, therefore it is typically tested for being valid before the referenced function is invoked, such as in

```
if( fptr == NULL ) ...
else fptr( params );
```

We assume that fptr shall have a type which is binary compatible to the function pointer type in C. This prevents to use a Cogent type `Option a` as the natural equivalent of a value that may be undefined. `Option a` is a Cogent variant type and is not binary equivalent to the C function pointer type.

If `fptr` is a member of a C struct, it would be possible to use a Cogent record with a taken field, if the function pointer is undefined. Then it would be clear from the record type whether the function pointer is valid and can be invoked. However, this would require two implementations for all functions with such a records as parameter, which is usually infeasible.

The generic type `MayNull` (see Section 4.5.5 cannot be used for function pointers since it is linear and the corresponding abstract functions expect values of the boxed type, whereas function pointers are always represented by an unboxed type.

The simplest solution is to assume that function pointers are always defined. Instead of using the value `NULL`, they must then be set to point to a default function. Then no test is required, the function can simply be invoked in Cogent.

Note that to be binary compatible the Cogent type of `fptr` is an abstract type of the form `#F_...`. To invoke the referenced function the translation function `fromFunPtr` must be used (see Section 2.7.9).

For example, if the function pointer is declared in C as

```
int (*fptr)(int arg1, const char *arg2);
```

the corresponding invocation in Cogent would be

```
(fromFunPtr fptr) (arg1,arg2)
```

with result type `U32`.

4.5.5 The Null Pointer

Translating C code which uses the null pointer is supported by the abstract data type `MayNull` defined in `include/gencot/MayNull.cogent` (see Section 2.7.10).

A typical pattern in C is a guarded access to a member of a struct referenced by a pointer:

```
if (p != NULL) res = p->m;
```

In Cogent the value `p` has type `MayNull R` where `R` is the record type with field `m`. Then a translation to Cogent is

```
let res = roNotNull p
          | None -> dflt
          | Some s -> s.m
    !p
```

Note that a value `dflt` must be selected here to bind it to res if the pointer is `NULL`. Also note that the access is done in a banged context for p. Therefore it is only possible if the type of `m` is not linear, since otherwise the result cannot escape from the context.

Another typical pattern in C is a guarded modification of the referenced structure:

```
if (p != NULL) p->m = v;
```

A translation to Cogent is

```
let p' = notNull p
         | None -> null ()
         | Some s -> mayNull s{m = v}
```

where the reference to the modified structure is bound to `p'`. Note that in the None-case the result cannot be specified as `p` since this would be a second use of the linear value `p` which is prevented by Cogent.

Alternatively this can be translated using the function `modifyNullDflt`:

```
let p' = fst (modifyNullDflt (p, (setFldM, v)))
```

using function `setFldM` for modifying the structure.

4.5.6 The Address Operator &

The address operator `&` is used in C to determine a pointer to data which is not yet accessed through a pointer. The main use cases are

- determine a pointer to a local or global variable as in the example

  ```
  int i = 5;
  int *ptr = &i;
  ```

- determine a pointer to a member in a struct as in the example

  ```
  struct ii { int i1; int i2; } s = {17,4};
  int *ptr = &(s.i2);
  ```

- determine a pointer to an array elemnt as in the example

  ```
  int arr[20];
  int *ptr = &(arr[5]);
  ```

  determine a pointer to a function as in the example

  ```
  int f(int p) { return p+1; }
  ...
  int *ptr = &f;
  ```

In all these cases the pointer is typically used as reference to pass it to other functions or store it in a data structure.

The binary compatible Cogent equivalent of the pointer is a value of a linear type. However, there is no Cogent functionality to create such values. Hence it must be implemented by an abstract function.

Address of Variable

In the first use case there are several problems with this approach. First, there is no true equivalent for C variables in Cogent. Second, it is not possible to pass the variable to the implementation of the abstract function, without first determining its address using the address operator. Third, if the address operator is applied to the variable in the implementation of the abstract function, the Isabelle c-parser will not be able to process the abstract function, since it only supports the address operator when the result is a heap address.

All these problems can be solved by allocating the variable on the heap instead. Then the variable definition must be replaced by a call to the polymorphic function `create` (see Section 2.7.8) and at the end of its scope a call to the polymorphic function `dispose` must be added. Then the address operator need not be translated, since `create` already returns the linear value which can be used for the same purposes. The resulting Cogent code for a variable of type `int` initialized to `5` would be

```
create[P_U32] heap
| Success (ptr,heap) ->
  fst INIT(Full,P_U32) (ptr,#{cont=5})
  | Success ptr ->
    let  ... use ptr ...
    in dispose (fst CLEAR(Simp,P_U32) (ptr,()),heap)
  | Error eptr -> dispose (eptr,heap)
| Error heap -> heap
```

where `INIT(Full,P_U32)` is used to initialize the referenced value. Here, the result of the expression is only the heap, in realistic cases it would be a tuple with additional result values.

Address of Struct Member

The second use case cannot be translated in this way, since the referenced data is a part of a larger structure. If it is separated from the structure and allocated on the heap, the structure is not binary compatible any more.

If the overall structure is not allocated on the heap, the same three problems apply as in the first use case. This can be solved in the same way, by moving the overall structure to the heap. Then it can be represented by the linear Cogent type

```
type Struct_Cogent_ii = { i1: U32, i2: U32 }
```

If the overall structure in Cogent is readonly the abstract polymorphic function `getrefFldI2` can be used to access the Cogent field through a pointer. This is an instance of the general `getref` operation for records as described in Section 2.7.11.

If the overall structure is modifyable, determining a pointer to the field would introduce sharing for the field, since it can be modified through the pointer or by modifying the structure. A safe solution is to use the abstract polymorphic function `modrefFldI2` which is an instance of the general `modref` operation for records as described in Section 2.7.11.

Address of Array Element

The third use case is similar to the second. It often occurs if the array itself is represented by a pointer to its first element (see Section 4.4.3). This case could simply be replaced by using the element index instead of a pointer to the element. The array index is nonlinear and thus easier to work with. However, this solution is not binary compatible, if the element pointer is also accessed outside the Cogent compilation unit.

A binary compatible solution can be achieved in a similar way as for struct members. The first prerequisite for it is to allocate the array on the heap.

Then the polymorphic functions `getrefArr` and `modrefArr` (see Section 2.7.12) can be used for working with pointers to elements.

Address of Function

The last use case is translated in a specific way for function pointers, using the translation function `toFunPtr` (see Section 2.7.9), which is generated by Gencot for all function pointer types. The resulting Cogent code has the form

```
Cogent_f : U32 -> U32
Cogent_f p = p+1
...
let ptr = toFunPtr[_,#F_XU32X_U32] Cogent_f
in ...
```

In this case `ptr` has the nonlinear type `#F_XU32X_U32` in Cogent and may freely be copied and discarded in its scope.

4.6 Completing the Cogent Compilation Unit