Report on Gencot Development

Gunnar Teege

November 30, 2023

Chapter 1

Introduction

This report is a living document which is constantly modified to reflect the development of Gencot. Not all parts described in the design chapter need already be implemented.

Gencot (GENerating COgent Toolset) is a set of tools for generating Cogent code from C code.

Gencot is used for parsing the C sources and generating 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. To distinguish these names from the names in the original C code Gencot uses name mapping schemas which support mapping all kinds of names to a different 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).

The remaining names in C are type names, function names, enum constant names, and names for global and local variables and function parameters. Additionally, there may be C constant names defined by preprocessor macro directives.

The scope of function parameters and local variables is restricted to the surrounding C function body. There their scope is determined by the block structure where parameters belong to the outermost block and names in a block shadow names in surrounding blocks. This block structure is translated to a similar block structure in Cogent, where both kinds of names are translated to Cogent variable names bound in the correspnding block in the Cogent function body expression. 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 other names on the global level are always mapped by Gencot, irrespective whether they have the correct case or not.

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`. A name with internal linkage can also be defined in an included file, but this is rarely done, because it introduces a separate object in every compilation unit which includes the file. Gencot uses the same prefix in both cases, because if two files `x.c` and `x.h` exist, the file `x.c` usually includes `x.h` and no different objects with the same name can be defined in both files.

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 (which have no linkage) 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 without linkage 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 and union.

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 C types are pointer types, function types, and array types. They always depend on a base type which in the C program must be defined before the base type is used. In Cogent a similar dependency can only be expressed

by a generic type: the Cogent compiler takes care that in the generated C code each instance of a generic type is introduced after all its type arguments have been defined. Therefore Gencot translates every C derived type to a generic Cogent type with its base type as its (single) type argument.

For C pointer types a fixed set of generic types is used by Gencot (see Section 2.6.7). Function types are translated to Cogent function type expressions. Only for array types generic type names are introduced which are specific for the translated C program.

To be able to process every source file independently from all other source files, Gencot uses a schema which generates a unique generic type name for every C array type. Derived types may syntactically occur at many places in a C program, so it is not feasible to generate the name from a position in the source file.

For C array types its size (the number of elements) may be part of the derived type specification. Gencot uses a seperate generic type name for every size for which an array type is used in the C program. If the size is specified by a literal the name has the form

```
CArr<size>
```

where `<size>` is the literal size specification. If the size is specified by a single identifier the name has the form

```
CArrX<size>X
```

where `X` is a letter not occurring in the identifier. In all other cases (also if no size is specified) the predefined name

```
CArrXX
```

is used which may lead to name conflicts in Cogent and must be handled manually.

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 the `<kind>` strings or the string `"CArr"` used for mapping C array types, or any other predefined type used by Gencot.

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.

In particular, if the C program contains a definition of the function `main`, it is treated in the same way: it is translated to a Cogent function with a mapped name (usually `cogent_main`) and an entry wrapper named `main` is generated with the same interface as the original `main` function. This entry wrapper thus replaces the original main function and the resulting Cogent compilation unit can be translated to an executable binary and used as a standalone program.

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 C (.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 C 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. If the global variable is actually a constant and never modified by the C program, it can be translated as an abstract parameterless function which is implemented in antiquoted C. If the global variable is modified, this is not possible, even if it is never modified by the Cogent code. Semantically a parameterless function must always return the same value and thus properties which take into account that the variable value may change could not be proven for the Cogent representation.

Global variables with non-constant values must be translated by explicitly passing them as arguments to all functions which access them. However, for efficiency, they should be passed by reference, i.e. using a pointer to the variable, which corresponds to a parameter of linear type in Cogent. If the access is read-only the parameter may have a read-only type in Cogent, otherwise it must not be discarded and must be returned as part of the function result. Gencot supports mechanisms to systematically introduce such parameters for translated functions.

An alternative approach would be to aggregate all global variables to a single "system state" and pass that state as argument to all functions which access a global variable. Gencot does not use this approach for the following reasons. If the variables are actually rearranged in a common struct which comprizes the system state this would not be binary compatible to the C data layout. Instead, the system state must be a collection of pointers to all global variables. However, that would be inflexible: it would not be possible to pass some of the pointers as read-only types and it would be inefficient because always all pointers have to be passed even if only some are used. Moreover, it should be easier to prove functional properties when only those arguments are present which are actually required.

The definition of a global variable cannot be translated to Cogent. It must either remain in the C code outside of the Cogent compilation unit or it must be implemented in antiquoted C as part of the Cogent compilation unit. Gencot decides this according to the definition's location: if it is located in a C source which is translated to Cogent it is implemented in antiquoted C, otherwise the original definition is used unmodified and the variable is declared as extern in antiquoted C.

The actual access to the global variable is implemented by the entry wrappers: whenever a Cogent function takes a pointer to a global variable as argument, the entry wrapper constructs the pointer with the help of the `&` operator and adds it to the function arguments. Pointers returned by Cogent functions are simply discarded. Since the wrappers are implemented in antiquoted C they can directly access the global variable in both cases, either defined externally or in antiquoted C.

When a global variable is defined or declared in antiquoted C as part of the Cogent compilation unit, its type must be specified. Here the Cogent antiquotation feature is used which allows to use the translated Cogent type as specification. Thus no additional type definitions are required from the C program, if Gencot translates all type information used in the translated parts of the C program.

The names of global variables are not mapped to Cogent names, since they

never occur in the Cogent program. In particular, they are not used for naming the additional function parameters used for passing the pointers to the global variables.

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. An external reference of a name with internal linkage occurs if the definition is contained in an include file which does not belong to the <packge>.

If the name 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 linkage must always be provided manually, since they are not defined in the <package>. The information for external references with no linkage (type and tag names) is not required since Gencot fully translates the corresponding definitions to Cogent as usual. References to variables with external linkage are never used in Cogent code, they are only used in antiquoted C code in the entry wrapper functions, there the information is provided automatically by Gencot. 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 no information is provided in Cogent, it must always be accessed through additional function arguments where they are originally passed by the entry wrappers.

- 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 C files where the Cogent `include` statement is not available. This is only possible if the included content is also an antiquoted C file. The fourth kind is the `#include` directive of the C preprocessor which can be used in antiquoted C 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 C 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 C 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 declarations in antiquoted C. Since they are only accessed in the entry wrapper functions they are put into the files `x-entry.ac` at the beginning before the entry wrapper definitions. For all external variables which are used by a function defined in `x.c` and must be passed as argument to the translated function a declaration is generated in `x-entry.ac`.

Derived Types

Gencot generates definitions of unary generic type definitions for derived array types used in the C source. These definitions are put in the file `<package>-dvdtypes.cogent`.

Corresponding type implementations in C are generated by Gencot in the file `<package>-dvdtypes.h`.

Global Variables

In C a compilation unit may define global variables. Gencot does not generate a direct 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 antiquoted C definitions for all global variables (toplevel object) defined in `x.c`. If the variable has external linkage the definition uses the original name, thus it can be accessed from outside the Cogent compilation unit. If the variable has internal linkage the definition uses the mapped name so that it is unique in the Cogent compilation unit.

In the Cogent compilation unit the defined global variables are only accessed in the entry wrapper functions. Therefore Gencot puts them into the files `x-entry.ac` at the places where their definition occurred in the original source. Additionally they are declared at the beginning of the file, so that they can be accessed in entry wrappers defined before that position.

Predefined Gencot Types and Functions

Gencot provides several Cogent types and functions which are used to translate C types and operators for them (see Sections 2.6 and 2.7). Cogent definitions for all these types and functions are provided in files which are part of the Gencot distribution and are contained in a directory `include/gencot`. When Cogent is used to process a source generated by Gencot the directory `include` must be made known to the Cogent preprocessor using the compiler option `-cogent-pp-args`. Then the files can be included in the Cogent source in the form

```
#include "gencot/xxx.cogent"
```

Some of the types and functions are abstract, the corresponding definitions in antiquoted C are provided in single files `gencot.ah` and `gencot.ac` in the Gencot distribution. These files must be processed by the Cogent compiler using the options `-infer-c-types` and `-infer-c-funcs`, respectively, to generate the C source as part of the Cogent compilation unit. For generic abstract types the Cogent compiler generates a C source file for every instance used in the Cogent program, these files are generated in the subdirectory `abstract` of the directory where the Cogent compilation unit is compiled. The file `gencot.ac` is compiled to a single file `gencot_pp_inferred.c`. Since nearly all predefined Gencot functions are polymorphic the file `gencot.ac` is used as a template and only the C source code for those function instances are generated in `gencot_pp_inferred.c` which are actually used in the Cogent program.

Abstract Data Types

There may also be cases of C types where no corresponding Cogent type can be defined or is predefined be Gencot. In this case it must be manually mapped to an abstract data type T in Cogent, consisting of an abstract type together with abstract functions. Both are put in a file `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 C code in the file `T.ac`. If T is generic, the additional file `T.ah` is required for implementing the types, otherwise they are implemented in `T.h`. In this case `T.h` must be `$esc`-included in `T.ac` so that it is included in the final C source of the Cogent compilation unit.

Gencot does not provide any support for using abstract data types, they must be managed manually according to the following proposed schema. An abstract data type T is implemented 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`. 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`. Declarations of all external global variables used in the entry wrappers, and definitions of all global variables defined in `x.c`

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

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

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

`<package>-dvdtypes.cogent` Generic type definitions for all used derived array types.

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

`<package>-dvdtypes.h` Implementations of abstract types defined in `<package>-dvdtypes.cogent`.

Main Files

To put everything together we use the files `<package>.cogent` and `<package>.c`. The former includes all existing `x.cogent` files and the files `<package>-exttypes.cogent`, `<package>-externs.cogent`, `<package>-dvdtypes.cogent`, and all required files `gencot/Xxx.cogent` for predefined types and functions. It is the file processed by the Cogent compiler which translates it to files `<package>-gen.c` and `<package>-gen.h` where `<package>-gen.c` includes `<package>-gen.h` and `<package>-gen.h` includes all files in subdirectory `abstract` which have been generated from `gencot.ah` and any other file `T.ah` for manually implemented abstract data types.

Cogent also compiles all files `x-entry.ac`, `T.ac`, `<package>-externs.ac`, and `gencot.ac` to corresponding files `x-entry_pp_inferred.c`, `T_pp_inferred.c`, `<package>-externs_pp_inferred.c`, and `<package>-gencot_pp_inferred.c`, respectively. Gencot postprocesses these files to yield corresponding files `x-entry.c`, `T.c`, `<package>-externs.c`, and `<package>-gencot.c`, respectively.

The file `<package>.c` includes all existing files `x-entry.c`, `T.c`, `<package>-externs.c`, `<package>-gencot.c`, together with `<package>-dvdtypes.h` and `<package>-gen.c` and constitutes the C code of the Cogent compilation unit. It is the file to be compiled by the C compiler to produce the executable program and to be read by the Isabelle C parser when checking the refinement proof.

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, using Cogent value definitions instead of having unrelated literals spread across the code has the advantage, that the Cogent compiler transfers the value definition to the generated Isabelle code which becomes more readable this way.

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". In every line it must contain the exact file specification, as it appears in the include directive, for example

```
"x.h"
MACROCALL
```

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 Including Files for C Code Processing

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.

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.

2.6 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 Item Properties

The Cogent type system reflects more and other data properties than the C type system, e.g., linear types and readonly types. Instead of simply determining these properties automatically, Gencot supports declaring them by the developer. Before translating a C program to Cogent, the developer may specify properties for several "items" in the C program. Gencot reads these declarations in addition to the C source code and uses them when determining the Cogent type for the translated item.

Items

An item for which properties can be declared for Gencot may be every entity in the C program which has a declared type. An item may also be "virtual" in the sense that it is not an entity in the C program, but Gencot introduces a corresponding translation in the Cogent program. An example are additional function parameters introduced wih the help of virtual items.

Items may be global, such as global variables and functions, or they may be sub-items of other items, such as a struct member or a function parameter. Sub-items are determined by the type of the main item. If that type is specified by a common identifier (typedef name or tag name) for several main items together, properties for a sub-item may not be declared individually anymore, instead, they must be declared for the corresponding sub-item of all main items together. Such sub-items are called "collective items" here.

Items may also be local variables defined in a function body. Gencot identifies local items by their C identifier. The C identifier need not be unique, not even in the same function, due to the block structure of C function bodies with the possibility to define variables in every block. Therefore local variables are additionally identified by their number of occurrence in the function body.

There are two kinds of global items: C variables (also called "objects" in the C standard) and C functions, which are defined on toplevel (called an "external definition" in the C standard), i.e., not locally in a function. Note that these two kinds only differ in their types: functions have a derived function type, objects may have any other type.

Global items are always named by a C identifier. Depending on the linkage, the identifier may be unique (external linkage) or it may only be unique for the compilation unit (internal linkage). Gencot uses the C identifier to identify global items. In the case of internal linkage the file name is used for disambiguation.

Gencot supports property declarations for the following kinds of sub-items:

- the members of a struct or union
- the elements of an array
- the data referenced by a pointer
- the parameters and the result of a function

Members are identified by their name. Parameters are identified by their name or by their position. The other sub-items are identified using specific mechanism.

Depending on their type, sub-items may again have sub-items. Gencot supports property declarations for arbitrary deep sub-item nesting.

Global items are always individual. A sub-item is individual only if

- the containing item is individual
- the containing item's type is not specified by a defined type identifier.

This means that the containing item's type must be an expression for a derived type.

If a struct or union declares a tag, all members can only be used as collective items, since the tag makes it possible to assign the type to different items so that a member declaration may be shared by several sub-items. Gencot does not check whether a tagged struct or union is only used for a single item, it always treats its members as collective.

A tagless struct or union can only be used at the place where it syntactically occurs, this may suggest that it may only be the type of one item and thus its sub-items cannot be shared. However, a tagless struct or union can be used as base type for several different declarators which may declare different items having this base type. It may even be used in a typedef to define one or more typedef names for it which may then be used to define any number of items with these types. Therefore Gencot also treats sub-items of tagless struct and unions always as collective.

In a similar way sub-items are collective, if the containing item's type is specified by a defined type identifier (a "typedef name"). As an example, if two function pointers are defined by

```
int (*fp1)(int i, char* p);
int (*fp2)(int i, char* p);
```

the parameter `p` occurs as individual sub-items of the individual items `fp1` and `fp2` and may have different properties for both. If the function pointers are instead defined by

```
typedef int (*fptyp)(int i, char* p);
fptyp fp1, fp2;
```

using a typedef name for the common function pointer type, the parameter `p` occurs only once, as collective sub-item of the collective item `fptyp` and its properties may not be different for `fp1` and `fp2`. The reason is that also in the Cogent code a common type is defined and used for the definition of both pointers. This implies, that no different properties can be applied to their function parameters, since there is only one common translation of the function type.

If, however, these definitions are extern to the Cogent compilation unit and the name `fptyp` is not used elsewhere in the Cogent compilation unit, it is resolved as described in Section 2.1.2 and never used in the Cogent code. There every function pointer is defined using a separate copy of the translated function type. In this case Gencot again uses two individual sub-items `p` for the individual items `fp1` and `fp2`, which implies, that different properties could be defined for them. This is ok, since the function type can be translated differently for both items.

Basically, collective items are identified by their type. If a property is declared for a type, it applies to all items which are declared to have this type. Additionally, for a collective item its sub-items can be determined in the same way as for an individual item.

Gencot does not support all kinds of C types to identify collective items. Only the following kinds are supported:

- typedef names,
- tag names and generated names for tagless struct/union types,
- derived pointer types with a supported base type.

Note that collective items have the form of a generalization hierarchy, where a collective item may include one or more other collective items which are more specific. This hierarchy is built by using typedef names.

For example, if the C program contains the type name definition

```
typedef struct str { int m1,m2; } str_t;
```

the collective item identified by `struct str` corresponds to all items declared to have the type `struct str`, whereas the collective subitem `m1` of `str` corresponds to all members `m1` of such items. The collective item identified by the type `struct str*` corresponds to all items declared as pointers to `str` structs. The collective item identified by the type `str_t` identifies all items declared to have this type. In general this is semantically a subset of all items declared to have the type `struct str`, therefore `str_t` corresponds to a more specific collective

item than `struct str`. Properties declared for `struct str` also apply to items with declared type `str_t` but not vice versa.

Gencot does not support primitive types for item identification, because (currently) there are no properties supported for them and they have no sub-items. Moreover, it seems not useful to declare a property for *all* items of a primitive type. Gencot also does not support derived array types for item identification because (currently) the only property supported for them is Read-only and it does not seem useful to declare this for all items of a certain array type. It could be useful to declare properties for the elements of all such items, but Gencot does not realize that. For derived function types or their parameters and results it could be useful to declare properties, but this is not implemented by Gencot either. The main reason for not supporting derived array and function types is the difficulty for defining a unique syntactical representation which is needed for the current implementation of the item property mechanism as described in Section 3.2.

Together, the rules for identifying items to declare properties for them are

- Basically, each item has a unique identifier. For a global item its identifier is derived from its name in the C program.
- For a sub-item its identifier is a path relative to the most general type item identifier of its nearest super-item for which the type is a typedef name or a struct or union type. If there is no such super-item, or it has no most general type item identifier, it is a path relative to the toplevel global item.
- If a typedef name `t1` is defined to denote another typedef name `t2`, then the type item identifier for `t2` is more general than that for `t1`. If a typedef name `t` is defined to denote a struct or union type `s`, then the type item identifier for `s` is more general than that for `t`.
- If the type of an item is an external typedef name which is always resolved (see Section 2.1.2) and the same holds for all more general type item identifiers, the item has no most general type item identifier.
- If the type of an item is a typedef name, a struct or union type, or an arbitrary deeply nested pointer to such a type the type item identifier and all more general type item identifiers can also be used to declare properties for the item. Properties declared for an item by different item identifiers are merged.

Properties

The properties which can be declared for an item are usually specific for the item's type. If a property is declared for an item for which its type does not support the property, the property is silently ignored by Gencot.

It is also possible to declare a negative property for an item. Negative properties have higher priority than normal properties and can be used to suppress properties declared elsewhere, for example using a more general type item identifier.

For some items Gencot derives properties automatically from the C program. Explicit negative properties can also be used to suppress such properties.

Gencot supports declaring the following item properties.

Read-Only The Read-Only property is supported for items of arbitrary type. If declared, the type of the translated item is made readonly by applying the Cogent bang operator `!` to it.

Note, that for several types the Cogent bang operator has no effect, these cases are equivalent to ignoring the property.

Note also that semantically the Read-Only property does not apply to the item itself, but to the item's possible values. The item's value can still be replaced although the item has been declared as Read-Only.

Const-Val The Const-Val property is only supported for global variables, i.e. for toplevel items which are neither types nor functions. Semantically it means that the variable value never changes, the variable always has its initial value. In this case Gencot translates the variable to a parameterless access function which returns the variable value.

The Const-Val property implies the Read-Only property but is stronger. If a global variable has a pointer type, the Read-Only property means that the data referenced by the pointer cannot be modified, whereas the Const-Val property means that additionally the variable value itself cannot be modified.

Not-Null The Not-Null property is only supported for items of pointer type. If declared, the type of the translated item is not wrapped by the generic `MayNull` type (see Section 2.6.7).

Although Gencot translates array types in the same way as pointer types, it never wraps them by `MayNull`. This has the same effect as specifying the Not-Null property for all items of array type, therefore it is ignored for items of array type.

No-String The No-String property is only supported for items of type `const char *`. If declared, Gencot uses the type `(CPtr U8)!` instead of `String` for the translated item (see Section 2.6.7).

If the type of an item is a typedef name, the No-String property can only be declared using its most general type item identifier.

Heap-Use The Heap-Use property is only supported for items of function type. If declared, the parameters and result of the translated function are extended by a component of type `Heap` (see Section 2.6.6). In the body of the function definition the name `heap` is used for this additional component. If there is already another parameter named `heap` it is named `globheap<n>` where `<n>` is the lowest number starting with 1 so that there is no other parameter with this name.

If the type of an item is a typedef name, the Heap-Use property can only be declared using its most general type item identifier.

Input-Output The Input-Output property is only supported for items of function type. If declared, the parameters and result of the translated function are extended by a component of type `SysState` (defined in the Cogent standard library). In the body of the function definition the name `io` is

used for this additional component. If there is already another parameter named `io` it is named `globio<n>` where `<n>` is the lowest number starting with 1 so that there is no other parameter with this name.

If the type of an item is a typedef name, the Input-Output property can only be declared using its most general type item identifier.

Add-Result The Add-Result property is only supported for parameter items. If declared, a component of the same type is added to the result, intended for returning a modified value of the parameter (see Section 2.6.6). The Add-Result property is ignored, if it is combined with the Read-Only property.

If at least one parameter has the Add-Result property the function result type is changed to a tuple type with the original result as first component followed by components for all parameters with the Add-Result property in the parameter order.

Global-State The Global-State property has an additional argument which must be an integer value greater or equal to zero. Every argument represents a distinct global variable in the C program. The property is supported for global variables and for virtual parameter items.

A Global-State property declared for a global variable introduces an association between the variable and the numerical argument of the property. If more than one Global-State property is declared for a global variable, only the first one is used, all others are ignored.

If a Global-State property is declared for a parameter item, it must be a virtual item, i.e., the parameter must not exist in the C program. A corresponding parameter is introduced by Gencot in the Cogent program for passing a pointer to the global variable to the function. As type for the parameter the pointer type is derived from the type of the global variable associated with the Global-State property according to its numerical argument. This pointer type is translated to Cogent as usual.

If two virtual parameters of the same function have a Global-State property declared, the properties must use different numerical arguments. Two parameters with the same argument would correspond to two parameters for pointers to the same global variable. This would introduce sharing for parameters of linear type and is forbidden in Cogent.

The Global-State property implicitly includes the Add-Result property. A parameter introduced by a Global-State property always has a linear type in Cogent. Since the global variable cannot be deallocated the parameter must never be discarded and always be returned as part of the function result. Thus Gencot automatically treats the virtual parameter in the same way as a parameter with the Add-Result property. Any explicit specification (positive or negative) of an Add-Result property is ignored when combined with a Global-State property.

The Global-State property also implicitly includes the Not-Null property. The pointer passed to the function is always a valid pointer to the global variable, it will never be `NULL`. Any explicit specification (positive or negative) of a Not-Null property is ignored when combined with a Global-State property.

The Global-State property for virtual parameters can be combined with the Read-Only property. In this case the Read-Only property affects the parameter type which is the pointer to the variable, whereas a Read-Only property specified for the corresponding global variable affects the type of the variable. If only the variable has the Read-Only property but not the parameter, a variable value may not be modified, but it can be replaced through the parameter by assigning a new value to the dereferenced parameter. If the Read-Only property is specified for a parameter with Global-State property the implicit effect of the Add-Result property is omitted.

All other item properties are ignored when specified for a parameter with a Global-State property.

Modification-Function (Deprecated) The Modification-Function property is only supported for parameter items of pointer type. If declared, the parameters and result of the translated function are rearranged to the form of a modification function (see Section 2.6.6) where the modified parameter is the first one for which the Modification-Function property has been declared. Note that, although the property is declared for a parameter, the function is affected by it as a whole.

The parameter type of the function is constructed as a pair consisting of the modified parameter and a tuple of all other parameters in the same order as in C. If the modified parameter also has the Add-Result property the result type of the function is constructed as a pair consisting of the modified parameter and a tuple of the original function result and all other parameters with the Add-Result property. If the modified parameter does not have the Add-Result property it is assumed that the original result returns the modified parameter value and the function result type is constructed as the pair consisting of the original result and the tuple of all parameters with the Add-Result property (which may be the unit type if there are no such parameters).

The Read-Only property is ignored, if it is combined with the Modification-Function property.

The Modification-Function property has been deprecated. Functions with this property are mostly useful for passing as parameter to other modification functions. This is not used by code generated by Gencot, it always uses lambda expressions for this purpose. On the other hand, a normal invocation of such a function is difficult to process by Gencot, because the function result components are permuted. Functions with Modification-Function property would only be useful for manually translated code, then they can also be created manually.

2.6.2 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 `long long int` (on 64 bit systems). Gencot assumes a 64 bit system and maps it like `long long int`.

2.6.3 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.

2.6.4 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. In a former version it reordered the fields lexicographically, but in the current version the fields are mapped in the same order as specified in Cogent. Thus, 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.

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.

2.6.5 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*`.

As of June 2020 there is an experimental Cogent builtin array type written `T[n]`. It is binary compatible with the C array type `t[n]` and is thus a good candidate for mapping C array types. However, it is still under development. Its compilation to C is not yet stable and the refinement proof has not yet been covered. Anyways, we expect it to become stable and use it for mapping C array types.

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 necessarily in a single `struct` directly before the array pointer.

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.

The Cogent builtin array type has the form `El[n]` for the boxed (linear) case and `El#[n]` for the unboxed case. The unboxed case can alternatively be written as `#(El[n])`, i.e., by applying the unbox type operator to the linear case. The array size `n` must be a constant expression of type `U32` which can be evaluated at compile time. As usual for `U32` modulo arithmetics is used so the result is never negative. The array size must not be `0`.

Let `el` be the translation of the element type `El` to C. Cogent directly translates every occurrence of type `El[n]` to the C type `el*` and it translates every occurrence of type `El#[n]` to the C struct type `struct {el data[n];}` with

the array as single member. This way values of the type can be assigned and passed as parameter by copying the whole array.

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 in expressions C arrays are always represented by a pointer to the first element. The builtin array types support both cases in Cogent: The unboxed form can be used for allocating space in memory, the boxed form can be used when an array is referenced by a pointer. Thus, C array types could normally be mapped to boxed array types, if they occur in the unboxed form the unbox operator can be applied.

As described in Section 2.7.3 Gencot provides for every mapped linear type a separate type for "empty" values. For a record type `R` the type `R take (..)` is used. For an array type the empty-value type must be constructed in a different way, instead of the operator `take (..)` an operator of the form `@take (0,1,2,n-1)` must be used (an operator of the form `@take (..)` seems to be intended but not yet implemented). Therefore it is not possible to define a common `EVT` macro for array and non-array types.

Moreover, in the case of boxed array types `El[n]` the information about the array size `n` is not preserved by the translation to C. As described in Section 3.9 the Genops mechanism for arrays depends on the possibility to determine the array size from the C type.

For both reasons Gencot wraps builtin array types in Cogent records. The resulting form is

```
type CArr<size> el = {arr<size>: el#[<size>]}
```

This approach has the following consequences:

- Only one Cogent type name is required for every array size.
- Type `#(CArr<size> el)` is considered to be linear if the element type `el` is linear, because in this case the Cogent type checker recognizes the builtin array type `el#[<size>]` as linear.
- Since the unboxed array is translated by Cogent to a struct type with a singe array member, together this approach results in a double wrapping of the array.
- The unboxed array can be accessed through take and put operations. Cogent translates take and put operations for unboxed arrays in a record as loops for assigning the single elements instead of the array as a whole (which would be possible, since it is wrapped in a struct). Note, that an assignment of the wrapped array may be more efficient, but this should be a decision for the implementation of the builtin array types.
- Since only unboxed builtin array types are used, the information about the array size is always preserved by the translation to C and can be used by Genops.

The <size> specification is made a part of the component name name<size> to force the records on the right hand side to be different. This is not necessary, however, as of December 2020, Cogent otherwise considers the record types to be equivalent for all sizes. When it generates the shallow embedding it selects one of them and uses the corresponding identifier CArr<size> for all array sizes. Since the size is ignored in the shallow embedding this is correct, but it is more readable if distinct identifiers are used.

Multidimensional C array types could now be translated using a multidimensional Cogent array type, such as el#[<s1>]#[<s2>] (note that all dimensions must be unboxed). Cogent translates such types back to an array of wrapped arrays, which is binary compatible to the multidimensional C array type. However, this form does not correspond to the form CArr<s1> #(CArr<s2> el) where both dimensions are wrapped twice. For consistency reasons the latter form is used by Gencot.

If the <size> is specified by a preprocessor identifier the identifier is resolved by Cogent when it translates the builtin array type and the actual size value is specified in the C array type where it can directly be retrieved by Genops.

Specifying Array Types

For constructing an array type from a given size and element type Gencot provides the preprocessor macro

```
CARR(<size>,<ek>,el)
```

defined in include/gencot/CArray.cogent. The parameter <ek> can be used to specify an unbox operator for the element type, it may be U or empty. The form CARR(<size>,U,el) is equivalent to CARR(<size>,,#el). The main reason for this parameter is compatibility to the macro CPTR defined in Section 2.6.7, so that it is possible to construct an array type and the pointer-to-element type from the same input. Both macros are mainly intended to be used for defining other macros.

The unboxed type #(Carr<size> El) is binary compatible with the C array type. It is a usual Cogent unboxed record and can be used freely. If another Cogent record has a field of an unboxed array type, the take and put operations can be applied in the usual way, assigning the array content as a whole.

If the element type El is again an array type, the meaning in C is a multidimensional array. This case is mapped to Cogent in the straightforward way, using an unboxed inner array type as element type for the outer array type. For example, the C type int[2][7] is mapped to the Cogent type

```
CArr2 #(CArr7 U32)
```

which in C is a doubly wrapped array of doubly wrapped elements which should nevertheless be binary compatible to int[2][7]. Using the macro CARR it can be specified as CARR(2,U,CARR(7,,U32)).

As described in Section 2.1.1 the form CArrX<size>X is used if the size is specified by an identifier and the single predefined generic abstract type CArrXX is used for all other size specifications and for array types without size specification. In the latter case C type definitions for instances must be provided manually. The type CArrXX is defined in include/gencot/CArray.cogent.

To specify the `<size>` of an array type by an identifier, the identifier must be a preprocessor constant defined in the Cogent source code. The expansion of the identifier must be a valid C expression. In simple cases, where the expansion is a single integer literal or an expression built from literals and the operators `+ - * /` the expansion is both valid C and Cogent code, then the identifier can also be used at other places in the Cogent source code. An example usage is

```
#ifdef LARGE
#define SECT 20
#else
#define SECT 5
#endif
#define SIZE 3*SECT
type CArrXSIZEX el = {arrXSIZEX: el#[SIZE]; }
```

It defines the generic type `CArrXSIZEX` to have either 60 or 15 elements, depending on the preprocessor flag `LARGE`. For additional information on possibilities how to specify the identifier see Section 3.12.3.

Whether a C array type is mapped to the boxed or unboxed form depends on its usage context. 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 to function parameters for passing a pointer to the variable and to an antiquoted C definition of the variable itself. The function parameters use the boxed type `CArr<size> El`. The definition uses the unboxed type `#(CArr<size> El)`.

- as type of a function parameter. In C this is "adjusted" to the pointer-to-element type. Since this is binary compatible with type `CArr<size> El`, this type is used as parameter type in Cogent. Alternatively the adjusted type could be mapped, resulting in type `CPtr El` or `El` (in case of elements of unboxed record or array type). The type `CArr<size> El` is preferred here by Gencot since it preserves more information. Although technically `NULL` may be passed as parameter value, the type is not wrapped by `MayNull` (see 2.6.7), thus always requiring a non-null pointer as argument.

- as type of a member in a struct type. Here the size of the array is relevant, therefore the unboxed type `#(CArr<size> El)` is used as type for the corresponding Cogent record field.

- as element type of another array, resulting in a multidimensional array. Again the size of the array is relevant here, the unboxed type `#(CArr<size> El)` is used as element type.

- as base type of a derived pointer type. In C such a type includes the information about the array size, it is a pointer to the whole array instead a pointer to a single element. Therefore the unboxed type `#(CArr<size> El)` is used as element type. Since the derived pointer type includes the `NULL` value it is wrapped by `MayNull` (see 2.6.7).

- 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 2.6.8, Gencot defines the mapped typedef name always as an alias for the boxed

type `CArr<size> El`. Then applying the unbox operator to the typedef name is equivalent to applying it to `CArr<size> El`.

Together we have the following mapping rules for C arrays with element type `el` and size specification `<size>`. Depending on the context of the C array type, additionally the unbox operator must be applied.

```
el[<literal size>] -> CArr<literal size> El
el[<size id>] -> CArrX<size id>X El
el[<complex size>] -> CArrXX El
el[] -> CArrXX El
el[*] -> CArrXX El
```

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

In Cogent linear types (pointers) are not supported if they point to data on the stack. Otherwise the Isabelle C parser will not accept the C code generated by Cogent. Therefore, whenever the boxed form is used for an array (i.e., whenever it is passed as parameter to a function or defined as a global variable) it must not be allocated on the stack. If it is stack allocated in the C program this must be adapted manually by allocating and deallocating it on the heap in the Cogent program or by any other solution.

Mapping Arrays to Abstract Types

Before the builtin array types have been considered for Gencot, C array types were mapped to abstract Cogent types. The corresponding concept is documented here to allow for comparing it to the use of builtin array types.

As a C derived type, every array type depends on its base type, which is the element type. The element type must be defined in the C source before the array type can be used. The Cogent compiler can only respect this ordering requirement if it knows that the abstract type to which the array type is mapped depends on the type to which the element type is mapped. The only way to make this known to the Cogent compiler is to use a generic type for the array with a single type parameter for the element type.

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. So a different type name is required for every array size occurring in the C program.

An abstract type is always boxed. The unboxed form can be constructed by applying the unbox operator. However, if the array type is mapped to an abstract type which corresponds to C type `t*` then the unboxed form corresponds to C type `t` and not to the array type, as required.

If the unboxed case is implemented as a struct with the array as its only member, the boxed 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 with the pointer to its only member

which is binary compatible with the pointer to the first array element. 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.

Together, this way the C array type can be mapped for literal size specifications using a single generic abstract Cogent type

```
type UArr<size> el
```

whith an antiquoted C type definition

```
typedef struct $id:(UArr<size> el) {
  $ty:el arr[<size>];
} $id:(UArr<size> el);
```

A drawback of this approach is that the type `#(UArr<size> el)` is not considered to be linear if the element type `el` is linear. For an abstract parameterized type Cogent considers the unboxed type always as unboxed, independent of the actual type parameters. Thus an unboxed array of linear values could be shared or discarded without being detected by the Cogent type checker.

For the empty-value type we again seek a solution which is the same as for record types. However, the `take` operator can normally not be applied to an abstract type. There are three possible approaches if array types are mapped to abstract types.

The first approach uses the Cogent compiler flag `-flax-take-put` which allows the `take` operator to be applied to abstract types (including abstract generic types). However, as of October 2019 the Cogent type checker is instable for such types and cannot process all applications. Therefore this approach is not used by Gencot.

The second approach uses a wrapper struct in the same way as for the builtin array solution:

```
type CArr<size> el = {arr<size>: #(UArr<size> el)}
type UArr<size> el
```

This makes it possible to use the Cogent take type operator for modelling empty-value array types. The drawback is that two different Cogent type names are required and that arrays are wrapped twice as struct. Also, the type `#(CArr<size> el)` is still considered nonlinear for a linear element type, as above.

The double wrapping could be avoided if type `UArr<size>` is directly implemented by the array type in C as in

```
typedef $ty:el $id:(UArr<size> el)[<size>];
```

However, there are two problems with this approach. First, the Cogent compiler does not support antiquoted type definitions for generating this form (the type parameter `$ty:el` is only known in a struct type with the generic type as tag name). Second, Cogent translates take and put operations for a record field to an assignment in C. Therefore, if field `arr` in a value of type `CArr<size> El` is put or taken, the C code generated by Cogent will be wrong, since arrays cannot be assigned in C.

The third approach uses a second generic abstract type

```
type UEArr<size> el
```

for every `<size>` to represent the empty-value array type. It is defined in C in the same way as type `UArr<size>`. This also avoids the double wrapping. It does not use the `take` operator to construct the empty-value type for array types, so there is no relation between both which is known to Cogent. However, the only situation where this relation would be useful is for type variables, but the `take` operator cannot be applied to type variables since it is not possible in Cogent to restrict a type variable to record types.

The drawback of the third approach is that the empty-value type is constructed in different ways for mapped arrays and for mapped structs. This makes it impossible to define a single common macro `EVT` for this type construction in Section 2.7.3. This further implies that no single macro is possible if it uses `EVT`, and, in particular, since the type for pointers to an unboxed array is the boxed array type (see Section 2.6.7) this case must be distinguished in Macros which apply `EVT` to a constructed pointer type.

Therefore the decision in early versions of Gencot was to use the second approach. For every array `<size>` used in the C program the two generic Cogent types `CArr<size>` and `UArr<size>` were used where the latter is abstract and is defined in antiquoted C as described above.

If the `<size>` is specified by a preprocessor identifier there is another subtle difference to the approach with builtin array types: the size information is only encoded in the abstract type name, there the identifier is used. To access the actual size value Genops needs the preprocessor definition of that identifier. When using builtin array types the actual size value is specified in the C array type where it can directly be retrieved by Genops.

2.6.6 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 (*)(...)`.

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 in the same order 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 has the form `#(CFun ((...,VariadicCogentParameters!) -> <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 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 3.11.5.

Together the rules for mapping function types are

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

Item Properties Affecting Function Type Mapping

Like every other type, the type of a function parameter may be readonly because the Read-Only property has been declared for the parameter, as described in Section 2.6.1.

Add-Result 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. This can be achieved by specifying the Add-Result property for the parameter (see Section 2.6.1). 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 with the Add-Result property.

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 has linear type. This can be specified by the developer by omitting the Add-Result property for the parameter.

Gencot uses the parameter types and declared properties to make parameter types readonly or add parameters to the function result, according to the following rules:

- If a parameter neither has the Read-Only property, nor the Add-Result property, it is translated as described before.
- Otherwise, if the parameter has the Read-Only property 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 has the Add-Result property, the function result is changed to a tuple and the parameter is added as component to that tuple. For a single parameter of type `l` with the Add-Result property 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.

Heap-Use If the Heap-Use property has been declared for a function, its type is translated according to the rule

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

i.e., the heap is appended to the parameter and result tuple. If `n = 0` the parameter type is `Heap`. Parameters with the Add-Result property are inserted before the `Heap` component.

Input-Output If the Input-Output property has been declared for a function, its type is translated according to the rule

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

i.e., the global system state is appended to the parameter and result tuple. If `n = 0` the parameter type is `SysState`. Parameters with the Add-Result property and the heap parameter, if present, are inserted before the `SysState` component.

Global-State If the Global-State property has been declared for a (virtual) parameter of the function, the function type is translated according to the rule

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

where `TG` is the translated type of a pointer to the global variable associated with the Global-State property (see Section 2.6.1). It is added to the result in the same way is if the Add-Result property had been declared for the parameter. If several virtual parameters are present with a declared Global-State property they are ordered according to the numerical argument of the Global-State properties.

The function type which is extended this way may occur in a different C source file than the associated global variable. The additional parameter may even be used to only pass the global value through to other functions, then the global variable is actually never accessed in the function body and neither its definition nor a declaration needs to be visible for the function. This implies that Gencot may not be able to determine the type of the global variable when it translates the function type, and thus may not be able to determine type `TG`.

For this reason Gencot introduces for every global variable with a Global-State property a type synonym for the Cogent type corresponding to a pointer to the variable. These type synonyms have the form

```
GlobState<i>
```

where `<i>` is the numerical argument of the Global-State property associated with the global variable. For the numerical argument `0` it is omitted. Using the numerical argument, Gencot can always determine this type synonym from the Global-State property and use it as the type `TG`.

Gencot generates the type synonym definition in place of a translation of the global variable definition at the corresponding position in the Cogent file.

If the function has the Heap-Use property and/or the Input-Output property the additional parameter and result components are added before the `Heap` or `SysState` parameter.

2.6.7 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 unboxed mapped array type. Note, that this is equivalent to the mapping of the array type itself, as defined in Section 2.6.5, so it is used by Gencot.

As described for the mapping of array types to abstract types in Section 2.6.5, Cogent must know that a pointer type depends on its base type, which is the type of the referenced value. Therefore pointer types are mapped to generic types which have the base type as single type parameter. Since there is no additional information to be encoded for a pointer type other than its base type, a single generic type is sufficient for all pointer types. As described in Section 2.1.1 the type name `CPtr` is used for this purpose.

Mapping Pointer Types

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 further access to the union members 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 `T` to which `t` is mapped. For every such type the generic type `CPtr T` name is used. This makes the Cogent program slightly more readable.

The resulting generic type can be completely defined in Cogent as

```
type CPtr ref = { cont: ref }
```

it is predefined in file `include/gencot/CPointer.cogent`.

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 also mapped using `CPtr` to

```
CPtr #(CArr<size> El)
```

As usual, the unboxed array type is used here, since the pointer in C includes the information about the array size, it points to the region where the whole array is allocated. In the same way the unboxed array type is used when the array occurs as a struct member or as the element of another array.

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

```
CVoidPtr
```

defined in `include/gencot/CPointer.cogent`. It is intended as a placeholder. Here the type of the referenced data is unknown. In C it is typically used as a generic pointer type which is cast to specific types of referenced values. This cannot be transferred to Cogent, to be used, the type must be replaced manually. No C type definition is provided for it by Gencot.

To make the construction of a pointer type for a given Cogent type more generic (in particular for its use in other preprocessor macros), Gencot provides in `include/gencot/CPointer.cogent` the preprocessor macro

```
CPTR(<knd>,<type>)
```

where `<type>` is a Cogent type and `<knd>` is `U` or empty. `CPTR(U,T)` expands to `T` and must be used if the type to be pointed to is an unboxed type of the form `#Struct_s`, `#Union_s`, or `#(Carr<size> El)`. `CPTR(,T)` expands to `(CPtr T)` and must be used in all other cases.

For example, the C type `struct s*` is mapped to `CPTR(U,Struct_s)` and the C type `int*` is mapped to `CPTR(,U32)`.

`NULL` Pointers

A Cogent linear type has the additional property that its values are never `NULL`. For a boxed record type this guarantees that the record fields can always be accessed without caring whether the pointer may be `NULL`. A C pointer type, instead, may include the `NULL` value, so it cannot be mapped directly to the Cogent linear type.

This could be handled by replacing the `NULL` pointer by a valid pointer to dummy data which is never used. The main problem here 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`.

Since in Cogent the values of boxed record types must not be `NULL`, C pointer types must be mapped to other types, if they are used in a context where `NULL` values are allowed. Gencot uses the generic type `MayNull a` for this purpose where `a` is the Cogent linear type for the same values without `NULL`.

Semantically, `MayNull` is a marker type, its values are the same as for type `a` with the exception that `NULL` is also allowed as value. The C implementation of `MayNull a` must be equivalent to that for `a`.

The simplest way would be to define `MayNull` as abstract, then its values can never be used to access the referenced data, they can only be processed in specific abstract functions. However, this does not fit with the special Cogent type system. If `T` is a linear type which is not escapeable (because it contains components of readonly type), the type `MayNull T` is also linear, but escapeable, if `MayNull` is an abstract generic type. This could be used to store a value of readonly type in a value of type `T` in a banged context, coerce the value of type `T` to type `MayNull T`, return it from the banged context and then access the readonly value in it, thus bypassing the Cogent type checker guarantees. If `MayNull T` instead is always defined as readonly it is not escapeable, but no more linear and could be shared and discarded, which also violates Cogent type checker guarantees.

Instead, Gencot uses the definition (in `include/gencot/MayNull.cogent`)

```
type MayNull a = { no_access: #a }
```

Every instance `MayNull T` is linear and it is escapeable iff `a` is escapeable. Moreover, values of type `MayNull T` can be consistently casted to type `T` in C implementations of abstract functions for processing these values. This is

important because otherwise the C code generated by Cogent would not be accepted by the Isabelle C parser. The only drawback of this approach is that the field `no_access` could be accessed in Cogent which could cause a null pointer derefence. However, this can be syntactically checked for a Cogent program so that it is easy to prove that no such null pointer dereferences can occur for it.

A bug in the current version of Cogent prevents the correct processing of a type `{no_access: #a}!` for some linear types `a`. Therefore the type `MayNull` is temporarily defined as abstract.

In Cogent for a linear type `T` the type `#(T!)` is equivalent to `#T`. Therefore, `MayNull (T!)` is equivalent to `{ no_access: #T }` which is equivalent to `MayNull T`. Therefore, if a type `T` is readonly, type `(MayNull T)!` must be used instead of `MayNull T`.

Whether a value may be `NULL` in C is not determined by its type, it depends on the way the value is used. It may be known to the C developer that for a parameter of pointer type a function is never (intended to be) invoked with `NULL` as actual parameter, but Gencot cannot determine that from the C program. Therefore Gencot maps pointer types to Cogent heuristically applying `MayNull` or not. Basically, pointer types are always mapped with `MayNull` applied, if they are not function pointers (for function pointers see the next section). Additionally, the item property Not-Null (see Section 2.6.1) is inspected, if it is declared for an item `MayNull` is omitted in the item's type.

A C pointer type may also result from "adjusting" an array type used as type for a function parameter. In this case Gencot assumes that the actual parameter is never `NULL` and omits `MayNull` in the mapped type of the parameter.

So together we have the basic mapping rules for pointer types:

```
void * -> MayNull CVoidPtr
struct s * -> MayNull Struct_s = MayNull { ... }
union s * -> MayNull Union_s
(*el)[...] -> MayNull (CArr... El)
otherwise: t * -> MayNull (CPtr T)
```

where `Struct_s` and `Union_s` are the names introduced for the struct or union types and `El` and `T` are the Cogent types to which `el` and `t` are mapped, respectively. When property Not-Null has been declared for the type's item the generic type `MayNull` is omitted.

The `String` type

Cogent supports the specific primitive type `String` which is translated by the Cogent compiler to `char*`. It is the type of the Cogent string literals. In contrast to all other Cogent types which correspond to a C pointer type it is neither linear nor readonly, it is a regular type for which no restrictions apply. This is motivated by the property of C string literals that they cannot be modified and are neither allocated nor disposed.

String literals are not supported by the Isabelle C parser, therefore a Cogent program using string literals cannot be processed and verified by Isabelle. However, string literals are useful during development time for debug output. Their use should be surrounded by conditional preprocessor directives so that they can be easily removed from the program for verification and for the production version.

Gencot supports using type `String` in Cogent by mapping string literals to values of this type. Since it is not possible to distinguish C string literals by their C type, Gencot uses a heuristics and the item properties No-String and Read-Only for this mapping.

Gencot basically maps the type `const char*` to type `String`. This type is often used in C as type for values which may be string literals, since it reflects the property of string literals to be immutable. The `String` type in Cogent has the additional property, that it is not possible to access the characters which are elements of the string. Therefore the item property No-String (see Section 2.6.1) is used to modify the mapping. If it is declared for a C item of type `const char*` the item's type is mapped as a normal pointer type, i.e. to the Cogent type `(CPtr U8)!`. The bang operator is applied because the C type is intrinsically readonly according to Section 2.6.9.

Gencot also maps the type `char*` to type `String` for all items for which the property Read-Only has been declared. To prevent this, the property No-String can be added to the item.

Mapping Function Pointers

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, as described in Section 2.6.6.

In other places, however, Cogent does not translate its function types to C function pointers. Instead, it uses a C enumeration 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.

These abstract types treat C function pointers in Cogent as fully opaque values. The only operation which can be applied to them is to translate them to the enumeration value used in Cogent. This is done by comparing the actual pointer values for equality. Since this also works for the `NULL` pointer, no specific measures are required for supporting `NULL` pointers for functions. In particular, the type `MayNull` is never used for function pointers.

Since comparing pointer values can be done in C independent from the pointer's type, a single common C pointer type such as `void *` is sufficient. Accordingly, a single abstract type in Cogent is sufficient to represent all C

function pointers. In particular, no dependencies on the parameter or result types need to be known by Cogent to position their definitions before the first use of the type in C.

However, for the developer the information about the parameter and result types are useful. Moreover, if different Cogent types are used for different C function pointer types, the Cogent type checker can be used to find mistakes. Since the full information about the parameter and result types is contained in the Cogent function types the easiest way is to use a generic abstract type

```
CFunPtr (P -> R)
```

with a function type `P->R` as single type argument. It is used for complete function pointer types where the parameter types are specified or it is specified that the function has no parameters using the keyword `void`. This can be thought of as an "annotated" or "wrapped" function type, although the constraint that the argument must be a function type cannot be specified or checked in Cogent.

For a complete C function pointer type its base type is mapped to a Cogent function type as described in Section 2.6.6. This includes the cases for variadic function types and the cases where parameter types are converted to readonly types or cause additional components in the result type for returning modified parameters.

For incomplete C function pointer types, where only the result type is specified but no parameter types, a second generic abstract type is used:

```
CFunInc R
```

where `R` is the result type. It may be an arbitrary mapped C type, only restricted by the C rules that it may not be a function or array type. Since however C array types are always mapped by wrapping them in a record, the corresponding unboxed types of the form `#(CArr<size> El)` *can* be used as function result types, although Gencot will never do so when translating a valid C program.

Both generic abstract types are defined in `include/gencot/CPointer.cogent`.

Although the C function pointer is a pointer, the pointer target value (the machine code implementing the function) normally cannot be modified through the pointer. 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 values of readonly types cannot escape from banged contexts. Gencot automatically applies the unbox operator in the form `#(CFunPtr (P->R))` whenever it uses one of the two generic function pointer types.

Avoiding Cyclic Type Dependencies

If a function pointer type of these forms occurs as type of a struct member, any direct or indirect reference of a parameter or result type to the struct type will be detected by Cogent as a cyclic type dependency. This is correct because this way the function may be recursive, although Cogent does not support recursive function definitions. If the function takes as parameter a struct which contains a pointer to itself, every invocation of the struct member will be a recursive call.

A common pattern in C is a struct with function pointers which take the struct as parameter, to provide processing functions for a data type together with the data type. In this case the functions will never invoke the function

pointers in the struct, they only process the struct data. So for this pattern recursive calls cannot occur.

To be able to support this pattern, Gencot hides the parameter and result types from Cogent by encoding the function type as an abstract type name (see next Section below). As a result, for every function type which is used as base type for a function pointer type there are now two forms: the Cogent type expression of the form `P -> R` and the abstract type name. Semantically, both are equivalent, however, this equivalence is deliberately hidden from the Cogent type checker. For incomplete function pointer types similar abstract type names are required for arbitrary non-function types.

An alternative approach would be to use Cogent recursive record types. They were not available in Cogent when Gencot has been designed, but they have been introduced later. We expect that this will prevent the Cogent type checker to signal an error, although it is not yet clear whether the refinement proof generated by Cogent will also cover the indirect recursive function calls which are made possible by it.

When generating the abstract type name 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. Therefore the name is generated by encoding the structure of the C function or result type. Gencot implements the encoding for all mapped C types. As described below, the encoding also supports the type operators for unboxed and readonly types and for `MayNull` types. Modified function parameters are not added to the result type, they are marked at the parameter type instead.

Now both kinds of function pointer types are mapped to the forms

```
CFunPtr Encfuntype
CFunInc Encrestype
```

with the abstract encodings as type arguments. Again, this constraint cannot be specified or checked in Cogent.

If a mapped function type and the corresponding mapped function pointer type are used in the same context it is useful to be able to syntactically generate both from a common specification with the help of a preprocessor macro. This can be supported by using the abstract type name also to define a synonym for the function type using another parameterized type of the form

```
type CFun Encfuntype = P->R
```

Note that this cannot be expressed in Cogent syntax or with the help of antiquoted C, since the type `P->R` must correspond to the encoding used as type argument. The Gencot mechanism for defining instances of polymorphic types and functions must be used here, as described in Section 3.9.

Another possibility is to use monomorphic abstract types instead. They have the form

```
CFunPtr_Encfuntype
CFunInc_Encrestype
type CFun_Encfuntype = P->R
```

It has the advantage that the function type synonyms can be defined using the normal Cogent syntax.

Due to an error in the Cogent compiler which does not treat types of the form `#(A B)` as unboxed, if `A` and `B` are abstract, the current Gencot version uses this approach.

Together the rules for mapping a function pointer type are

```
t (*)( ... ) -> #CFunPtr_encfuntype
t (*)() -> #CFunInc_encrestype
```

where `encfuntype` is the encoding of the function type `t( ... )` and `encrestype` is the encoding of the result type `t`.

As additional convenience, for every complete function pointer type a Cogent type name of the form `CFun_encfuntype` is introduced for the base function type. This makes it easy to generate both types from `encfuntype` in a preprocessor macro.

For example, the C function pointer type `int (*)(int, int[10])` is mapped to the abstract type

```
#CFunPtr_FXU32XA10_U32X_U32
```

with generated type definitions

```
type CFunPtr_FXU32XA10_U32X_U32
type CFun_FXU32XA10_U32X_U32 =
  (U32,CArr10 U32) -> U32
```

Note that the macro call `CPTR(,T)` expands to `(CPtr T)` also if `T` is a Cogent function type. This corresponds to a pointer to the function enumeration value used by Cogent and is not related to the C function pointer type. Therefore `CPTR` should not be used for function pointer types.

Encoding C Function Types

Gencot represents C function pointer types with the help of abstract Cogent types which encode C function types. These function type names are constructed from encodings of all parameter types and the result type. Hence Gencot implements a schema for encoding all C types as Cogent abstract type names.

A C type is either a primitive type, a derived type, a typedef name, or a struct/union/enum type. Gencot maps all struct/union/enum types to a Cogent type name. Primitive types and typedef names are specified by a single identifier in C and are also mapped to a Cogent type name by Gencot. These names are directly used as encoding. Thus only for the derived types (pointer, array, and function types) a nontrivial encoding must be defined.

Every derived type in C has a single base type from which it is derived. The base type of a derived type is always either another derived type, or it is mapped by Gencot 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

```
AXX
```

which may lead to name conflicts in Cogent and must be handled manually. As described in Section 2.6.5, a C array type may be mapped to an unboxed or boxed form, depending on its usage context. The unboxed form is only relevant here, if the array type is the element type of another array. Then it is encoded by an additional pseudo derivation step encoded as `"U"`.

A function derivation step is encoded in the form

```
FX<P1>X<P2>X...X<Pn>X
```

where the `<Pi>` are the encodings of the parameter types and `X` is a letter not occurring in any of the parameter type encodings. A parameterless function type is encoded as `FXX`, whereas an incomplete function type where no parameter types are specified is encoded as `F`. If the function type is variadic, an additional pseudo parameter type `VariadicCogentParameters` is added as last parameter type.

In some cases Gencot maps a C pointer type to a boxed Cogent type and its base type to the corresponding unboxed type by applying the unbox operator `#` to the Cogent type. In these cases the pointer derivation step is omitted in the encoding, and the base type is encoded by applying a pseudo derivation step of the form `"U"`.

C pointer types can be mapped as a linear type `P`, if the values are never `NULL`, otherwise they are mapped as the specific type `MayNull P`. In a type encoding the application of `MayNull` is represented by a pseudo derivation step of the form `"N"`.

Linear types may be mapped by Gencot as readonly or not (see Section 2.6.9). This is encoded by pseudo derivation steps of the form `"R"` for readonly and `"M"` ("modifyable") otherwise. Nonlinear types are not marked by either of these steps. Parameters of linear type for which the item property Add-Result has not been declared are also not marked by either of these steps.

For all derivation steps which are applied to a base type, their encodings are concatenated, beginning with the last derivation step, with an underscore `_` as separator. For a derived pointer or function type the base type can be the pseudo type `void`. In these cases the identifier `Void` is used as base type encoding.

Hence, for example for the C type

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

the encoding is

```
A5_P_FXU32XR_N_P_U16X_U32
```

These encodings are used for representing C function pointer types by prepending `CFunPtr_` or `CFunInc_` to construct abstract Cogent type names. For example, the C function pointer type

```
int (*)(int[16], struct str)
```

is mapped to the abstract Cogent type defined as

```
type CFunPtr_FXA16_U32XU_Struct_Cogent_strX_U32
```

2.6.8 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. Also, expanding all typedef names may cause an exponential growth of the source code.

Moreover, when a C type is derived from a typedef name, Gencot also maps the derived type using the mapped typedef name as type argument in the corresponding parameterized type or type encoding. Since the mapped typedef name is defined to be a synonym for the mapped type definition, the resulting parameterized Cogent type is equivalent to the mapping of the derived type with the resolved typedef name as base type.

An exception from this rule are typedef names for struct, union, and array types. Mapping pointer types derived from such typedef names with the help of `CPtr` would result in the following situation:

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

where `Struct_s` is the name of the Cogent record type corresponding to `struct s`. The problem here is that for the mapped typedef name the pointer corresponds to a pointer to pointer to struct which is not binary compatible with the mapped struct pointer.

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 (for arrays depending on the usage context), the pointer type derived from it is mapped without unbox operator applied.

A similar approach is used for typedef names resolving to a C pointer type which is mapped to a Cogent type of the form `MayNull P` (see Section 2.6.7). Since it may depend on the Not-Null property of the typed item whether its

values may be `NULL` or not, it may be necessary to use the type `P` for some items. To be able to do this also when a typedef name is defined for it, Gencot defines the typedef name as a synonym for the type `P` and translates its use by applying `MayNull` to the type synonym, if necessary.

Special treatment is also required for the abstract type names used for representing function pointer types, as described in Section 2.6.7. For them substitutability of contained type names (i.e. substituting a Cogent type name by its definition results in an equivalent type) is no more automatic, if mapped typedef names are used in names constructed by the type encoding rules. To provide substitutability for such types Gencot explicitly defines every occurring type encoding which contains mapped type names as a synonym for the encoding of the fully resolved type. Through transitivity of the type equivalence relation this results in full substitutability for all type encodings and makes it feasible to use mapped typedef names also in type encodings.

The form of the actual type synonym definitions depends on the approach used for constructing function pointer type mappings using type encodings (see Section 2.6.7). If generic types are used which take the encoding as type argument, only one synonym definition of the form

```
type Enctype = Encresolvtype
```

is required for every occuring type encoding `Enctype` with embedded mapped typedef names.

In particular, if the base type of a function pointer type is a typedef name for a function type, this makes the mapped typedef name `Enctype` a synonym for the fully resolved function type *encoding* instead of the function type itself. However, this is not a problem: in C a typedef name for a function type cannot be used for defining a function of that type, since a definition must always include the parameter names. A C typedef name for a function type can only be used for declaring functions and for constructing the corresponding C function pointer type, either explicitly or implicitly by adjustment. So for every use of the second kind it must be mapped to a (synonym for an) encoding anyways, and a synonym for the Cogent function type is never required. Function declarations are only translated to Cogent for external functions, resulting in abstract function definitions for exit wrappers. Here the function type name is resolved to the function type.

If monomorphic types are used, a synonym definition is required for every type constructed from such a type encoding:

```
type CFunPtr_Encfuntype = CFunPtr_Encresolvfuntype
type CFunInc_Encrestype = CFunInc_Encresolvrestype
```

Here, if `Encfuntype` is a mapped function type name, no synonym is defined for it in isolation. Therefore it can be used as synonym for the corresponding Cogent function type.

Names for function types are translated without unbox operator applied. It could be added to make apparent that it is a nonlinear type, however, the current Cogent version detects a function type with unbox operator as not equivalent to the function type without unbox operator.

The resulting mapping rules are for function type names:

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

for names of function pointer types:

```
tn -> #TN
```

for names of a struct or union type:

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

for names of an array type:

```
tn -> TN, #TN
tn* -> Maynull TN
```

depending on its usage context for `tn`, for names of a pointer type:

```
tn -> MayNull TN
```

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, union, or array 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 pointer type Gencot omits the application of `MayNull` to the defining type.

2.6.9 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 const qualified 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.

Pointer Types

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 transitively 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 Read-Only property is set for the corresponding item (see Section 2.6.1). If the property is not removed by the developer the item's type is translated to a Cogent readonly type by applying the bang operator `!` to the type after translating it as described in Section 2.6.7.

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

If a C pointer type is translated to a type of the form `MayNull t` (see Section 2.6.7) the bang operator is applied after `MayNull` resulting in `(MayNull t)!`. If a typedef name is defined for such a type the `MayNull` is removed as described in Section 2.6.8 and applied together with the bang operator to every occurrence of the translated typedef name. Since the defined type can only be banged if it or the typedef name has the Read-Only property the bang operator is still preserved for the defined type, other than the `MayNull`. The difference is that the bang operator is caused by the presence of the Read-Only property whereas the `MayNull` is caused by the absence of the Not-Null property.

Global Variables

If a function accesses a global variable, a pointer to the variable is passed to the function, if specified using a Global-State property (see Section 2.6.1). In this case a `const` qualifier for the variable's type becomes relevant, because it is now the base type of the function parameter. Moreover it means that the variable value is never modified and always has its initial value.

Gencot exploits this as follows. If a global variable has a `const` qualified type and all contained pointers have a `const` qualified base type, the Const-Val property is set for the global variable item (see Section 2.6.1). Note that in case of an array type a `const` qualified base type has the same meaning as if

the array type itself is `const` qualified. If the property is not removed by the developer, Gencot translates the variable to a parameterless access function. The function's name is the mapped name of the variable.

If the variable's type is a primitive type, an enum type, or a pointer type (including function pointers) the access function returns a value of the mapped type (which is not banged). If the variable has a derived array type the function returns a value of the readonly boxed mapped type, if it has a struct or union type the function returns a value of the readonly corresponding boxed type. In both cases the result corresponds to a pointer to the variable. The readonly type prevents modifying the variable through this pointer.

Additionally, a Global-State property can be declared for a variable with Const-Val property. Then function parameters can be introduced for passing pointers to the variable to functions. All these parameters must explicitly be declared Read-Only to prevent modifying the variable value through the pointer.

The String Type

As described in Section 2.6.7, the type `char*` may be translated to the Cogent type `String`. This is only done, if the Read-Only property is present for the corresponding item. Since `String` is not linear Gencot always omits the bang operator for type `String`.

If a typedef name is defined in C for a type which is translated to type `String` the bang operator is also omitted for the translated typedef name. This is consistent with the case where typedef names are defined for other C types which are translated to Cogent nonlinear types, here the translated typedef name is also never banged.

Array Types

As described in Section 2.6.5, the mapping of C array types depends on the type's context and may be a linear type or not. The properties Const-Val and Read-Only for items with array type must be treated accordingly.

An array type is mapped to a linear type if it is used as type of a global variable or a function parameter. In case of a global variable with a Const-Val property, such as

```
const int gvar[5];
```

Gencot translates the variable to an access function returning a readonly pointer:

```
cogent_gvar: () -> (Carr5 U32)!
```

In case of a function parameter with Read-Only property, such as

```
void f(const int p[5]);
```

the parameter type is already "adjusted" to type `const int *p` in C and Gencot maps the parameter type accordingly. Here, the Read-Only property causes the parameter type to be readonly:

```
cogent_f: (Carr5 U32)! -> ()
```

If an array type is used as base type of a derived pointer type the pointer type is mapped to a readonly type, as described above, if its referenced subitem has a Read-Only property declared. The same applies if an array type is used as element type of another array type (multidimensional array) with a Read-Only property for the inner array item.

If an array type is used as the type of a member of a composite type it is mapped to a record field with an unboxed type, hence a Read-Only property for the member is irrelevant. Since the current Cogent version has problems when bang and unbox operators are combined, Gencot omits the bang operator in this case.

If a typedef name is defined for an array type in C, the mapped typedef name is defined for the linear type in Cogent. If the typedef name is used for a member of a composite type, the Cogent type must not be readonly, therefore it cannot be made readonly in the definition, even if the typedef name has a Read-Only property. Instead, the mapped typedef name must be banged whenever no unbox operator is applied.

Manual Readonly Types

Values of a C type where not all pointer base types are const qualified may still be used in an immutable way in the C program, if the C programmer forgot to specify the const qualifier or did not care about it. In such cases the item property Read-Only can be set manually to tell Gencot to apply the bang operator when it maps the type to Cogent.

For a `const` qualified array type used for a member of a compound type the Read-Only property is always ignored, even if it has been set manually.

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 translation when Gencot does not support the translation of a C expression or statement, so that the resulting Cogent code is syntactically valid and can be processed and checked by the Cogent compiler.

For constructing dummy expressions Gencot provides in `include/gencot/DummyExpr.cogent` the polymorphic abstract function

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

from a string 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.

Every dummy expression has the form of an application of this function to a string literal, which is used to specify the reason why Gencot could not translate the C code fragment.

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 all Gencot function pointer types (see Section 2.7.9) the default value is the `NULL` pointer (which cannot be denoted directly in Cogent).

For types of the form `MayNull P` (see Section 2.6.7) a useful default value would be the `NULL` pointer. However, since these types are the only linear types with a default value, and the `NULL` value can be denoted by the function `null` defined in Section 2.7.10, Gencot does not support a default value for such types.

For a regular tuple (which has no component of linear or readonly type) the default value 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 (in the C implementation) variant. For a regular unboxed Gencot array type it is the array of default value elements.

Gencot defines the polymorphic abstract function

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

in `include/gencot/Default.cogent`.

Gencot provides instances for all regular types which contain no Cogent functions.

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 existing data on the heap or for existing global variables, as explained in Section 4.7.4. 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

```
EVT(vvt)
```

which expands to the empty-value type `vvt take (..)` corresponding to the valid-value type `vvt`.

It is not possible to define a generic type for this purpose, since in Cogent it is not possible to apply the `take` operator to a type variable.

The macro 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 pointer type `CPtr U32` can be specified as

```
create[EVT(CPtr U32)]
```

Note that although it is syntactically possible to use a type `MayNull P` as type argument for `create` and `dispose`, it does not make sense, because a successful result of `create` is never `NULL` and `dispose` need not be applied to `NULL`.

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 or other modification functions to its first argument. An abstract Cogent function is a modification function if the C implementation always returns the same pointer, without deallocating it in between.

For generality we extend the types `ModTypeFun` and `ModFun` to non-pointer types `obj` and `res`. There they have the normal Cogent semantics and behave like the type `ChgFun` defined below. Note that the types `obj` and `res` may still be linear if they correspond to structures containing pointers.

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.

In case of an error a `ModTypeFun` may not be able to produce a result value of type `res`. For this case Gencot defines a function type for "transactional" modification functions which either succeed or "roll back" the modification and return the original input value:

```
type TModTypeFun obj res arg out =
   (obj,arg) -> Result (res,out) (obj,out)
```

using the variant type `Result` from the Cogent standard library. The result type is not defined in the form `(Result res obj,out)` because then it must always be bound explicitly before it can be used in a match expression. With

the definition used here the modification function call can be directly used in a match expression where the additional output is bound by the patterns in the alternatives.

Note that a `ModFun` can be transactional, if it returns the information that an error occurred as part of the additional result of type `out`.

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 (allocated) during initialization or they can be passed as arguments to the initialization function. 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 vvt arg out
type ClrFun vvt evt arg out = ModTypeFun vvt evt arg out
type TIniFun evt vvt arg out = TModTypeFun evt vvt 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 third type supports "transactional" initialization functions which may fail and are rolled back, they return a result of type `Result vvt evt`, as for type `TModTypeFun` (see Section 2.7.4). This is typically the case when initializing a part of the value includes allocating space on the heap. If this is not done, initialization only consists of storing values, which cannot fail. For clearing we always assume that no error can occur, then the rollback for transactional initialization 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). TIniFun evt vvt Heap Heap
clearHeap : all(vvt:<E,evt). ClrFun vvt evt Heap Heap
initSimp : all(evt,vvt:<E). IniFun evt vvt () ()
clearSimp : all(vvt:<E,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>[EVT(vvt),vvt]
CLEAR(<k>,vvt) -> clear<k>[vvt,EVT(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[EVT(vvt),vvt]`. The macros can be used for all types `vvt` which are implemented as a Cogent record type.

Gencot also defines the macros

```
INITTYPE(<k>,vvt)
CLEARTYPE(<k>,vvt)
```

which expand to the corresponding function types.

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` may not contain any readonly parts. One reason is that readonly values are a copy of a pointer, so they cannot be initialized internally. Another reason is that the current Gencot implementation cannot distinguish which parts are readonly and which parts are not, therefore it cannot treat them differently, which is necessary for clearing them: copies of a pointer must not be disposed. For this reason even parts of type `MayNull P` are not allowed, although they could be initialized by setting them to `NULL`. The restriction not to contain readonly parts can be expressed in Cogent by the type constraint `E`, this is specified for type `vvt` for both functions.

Function `initHeap` allocates all parts of linear type on the heap and `clearHeap` disposes them. Parts of type `MayNull a` are initialzed to `null` and are disposed only if they are not `null`. All parts of nonprimitive type are initialized or cleared using function `initHeap` or `clearHeap`, respectively. All parts of primitive or Gencot function pointer type are initialized to their default value `defaultVal ()` and are cleared by doing nothing. Whenever an allocation fails for `initHeap`, all other allocations are rolled back using `clearHeap`. The initialization function returns a variant value which signals success or error.

The third pair passes no additional information. Moreover, `vvt` must have no readonly or linear parts other than parts of a type `MayNull P`. Since `vvt` itself must be linear, the additional constraint cannot be expressed in Cogent, only readonly parts are excluded by specifying type constraint `E` for it. 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.

Since the type of `initHeap` is different from that of the other initialization functions they cannot be passed as argument to a common function parameter. An alternative would have been to define all initialization functions using the common type `TIniFun`, but then their use would always require unnecessary checks and implementing an error case which can never occur.

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. A better choice can be to use another part which safely exists as default value, since that has the expected type and no restrictions apply.

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. However, this approach makes chaining cumbersome, since the rest of the chain is passed as additional input and must therefore also be returned as result. A fourth possibility is to change a part which safely exists instead. Thus the part change function is executed and consumes its input and produces its output in the normal way. This is the most flexible approach, it allows arbitrary types as additional input and result to the part change function.

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 `CPtr 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 similar 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 2.7.4 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 `CPtr Ref` where `Ref` is an arbitrary non-function Cogent type (except the unit type). These types are generated by Gencot for several C pointer types (see Section 2.6.7).

Gencot pointer types always point to a value of primitive type, 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 `CVoidPtr`. Since this is the mapping of the C type `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 use additional Gencot pointer types by applying the generic type `CPtr` to other Cogent types. Gencot provides the operation support described in the following sections also for such types.

Creating and Disposing Pointers

Since every Gencot pointer type is implemented by a Cogent record type, the corresponding empty-value type can be constructed by taking the single field: `(CPtr Ref) take (..)` which is equivalent to the expansion of the macro call `EVT(CPtr Ref)` (see Section 2.7.3).

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

```
create[EVT(CPtr Ref)]
dispose[EVT(CPtr Ref)]
```

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 actually 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 `EVT(CPtr U32)` can be initialized to the value `5` with typechecks by the Cogent expression

```
INIT(Full,CPtr U32) (p,#{cont=5})
```

and a pointer `p` of type `CPtr (CPtr U32)` can be cleared with typechecks by the Cogent expression

```
CLEAR(Heap,CPtr (CPtr 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 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 2.6.7, function pointer types are mapped by Gencot to abstract types of the form `CFunPtr_EncFuntyp` where `EncFuntyp` encodes a C function type, or `CFunInc_EncRestyp` where `EncRestyp` encodes the C result type. 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,funptr). fun -> #funptr
```

where `fun` is a Cogent function type and `funptr` is the Cogent abstract type representing the function pointer.

Invoking a function of type `fun` is supported by translating the function pointer of type `#funptr` to the Cogent function (equivalent to the enumeration value) which then can be invoked in the usual way in Cogent. This translation must be implemented in antiquoted C by comparing the function pointer to the fixed list of the pointers to all known functions of the corresponding type in the Cogent compilation unit and returning the enumeration value for the function which matches. The problem here is which value to return if no known function matches the pointer. In particular, this may happen for the `NULL` pointer.

The straightforward solution would be to generally return a value of type `Result fun ()` and return the unit value `()` in case of an unknown function pointer. However, that would require to use a `match` expression to check the result whenever a function pointer is invoked. Moreover, it is usually not clear how to react when the unit value is detected, this may be difficult if values of linear types are present which must not be discarded.

In a C program a valid function pointer always results from applying the address operator `&` to a defined function. Whenever a function pointer is invoked at runtime, it is either such a valid pointer or it is invalid (including `NULL`) which results in undefined behavior. All valid function pointers can be covered by the translation, if all C functions of the corresponding type to which an address operator is applied (which can be statically detected) are either included in the Cogent compilation unit or are referenced as external (see Section 2.1.2). In the former case there will be a corresponding Cogent function definition, in the latter case there will be an abstract Cogent function definition. In both cases the Cogent compiler generates an enumeration value and a dispatcher for invoking the function. If the list of known function pointers includes all these functions, all valid pointers are covered.

Since the enumeration values for representing Cogent functions are integer values in C, it is possible to return an arbitrary integer value in case of an invalid pointer, i.e., a pointer which does not match any function in the list. The dispatchers for function invocation generated by the Cogent compiler are implemented by invoking one of the known functions, if an unknown enumeration value is specified. Thus using an arbitrary enumeration value results in defined but unintended behavior. Together, this way a C program with possible undefined behavior is translated to a Cogent program with defined behavior,

which is equivalent with that of the C program if no undefined behavior occurs. This property can be proved for the shallow embedding in Isabelle, by proving that the function pointer always matches one of the known functions.

Gencot supports the translation of function pointers to Cogent functions by the polymorphic abstract function

```
fromFunPtr: all(fun,funptr). #funptr -> fun
```

Since the enumeration values for Cogent functions start at 0, it returns the value `-1` for all unknown function pointers. That is not visible in Cogent and corresponds to one of the valid Cogent functions due to the behavior of the invocation dispatcher.

The C program may contain a test for being not `NULL` before it invokes a function pointer. To be able to translate this Gencot provides the polymorphic abstract function

```
nullFunPtr: all(funptr). #funptr -> Bool
```

Note, that this does not guarantee that the function pointer is valid, if the result is `False`.

Note also that `toFunPtr` always gets a known Cogent function as input and returns the corresponding valid function pointer.

Gencot defines in `inlude/gencot/CPointer.cogent` the preprocessor macros

```
TOFUNPTR(ENCFT)
FROMFUNPTR(ENCFT)
```

which expand to typed instances of both functions, where `ENCFT` is the encoding of the C function type.

Hence for example for the C type

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

the following function instances are provided

```
toFunPtr[(U32,U16) -> (CPtr U32),CFunPtr_FXU32XU16X_P_U32]
fromFunPtr[(U32,U16) -> (CPtr U32),CFunPtr_FXU32XU16X_P_U32]
```

which can be specified using the macros as

```
TOFUNPTR(FXU32XU16X_P_U32)
FROMFUNPTR(FXU32XU16X_P_U32)
```

For incomplete function pointer types of the form `#CFunInc_ENCT` Gencot cannot provide instances of `fromFunPtr` and `toFunPtr`, since it cannot determine the corresponding Cogent function type. Therefore such function pointers cannot be used in any way in Cogent, they can only be passed through.

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

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

2.7.10 `MayNull` Pointer Types

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, Gencot reflects this in Cogent by marking the type with `MayNull` (see Section 2.6.7). The way how to work with 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. In all these cases the pointer needs no `MayNull` wrapper type, since the surrounding type already distinguishes between `NULL` values and valid values.

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. In all these cases the `MayNull` marker type is required.

Since `MayNull` is an abstract type no predefined operations can be applied to its values. Gencot only defines 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:<E). () -> MayNull a
roNull:    all(a). () -> (MayNull a)!
mayNull:   all(a:<E). 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`. For both the type `a` must be escapeable, otherwise it would be possible to wrap a value of type `P!` which cannot escape from a banged context as a value of type `MayNull (P!)` which can escape. The operations `roNull` and `roMayNull` must be used for readonly pointers, they return a result which is again readonly. This cannot be done by applying the bang operator to the result of `null` or `mayNull` since then the readonly result value cannot escape the banged context.

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 con-

straints 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`.

Based on the abstract functions the function

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

is defined for an explicit test for the null pointer. It could be implemented in Cogent based on function `roNotNull` but is implemented in C for efficiency reasons, avoiding the intermediate use of a value of type `Option a`.

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, clearing, and disposing are available.

The type constructors for modification function (see Section 2.7.4) 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:<E).
   ModFun (MayNull a) a a
modifyNull: all(a:<E,arg).
   ModPartFun (MayNull a) a arg arg
modifyNullDflt: all(a:<E,arg:<D,out:<DSE).
   ModPartFun (MayNull a) a arg out
```

Again, instances of the functions are only defined if `a` is a linear type. Additionally it must be escapeable, since otherwise no values of type `Maynull a` can be constructed. The operations are not supported if `a` is not escapeable, since then the wrapped type would be `(MayNull a)!` and modifying its values would contradict the meaning of the bang operator. Like `isNull` the functions could be implemented in Cogent but are implemented in C for efficiency.

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 2.6.4 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 `EVT(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[EVT(R)]
dispose[EVT(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/CStruct.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 EVT(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 EVT(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 EVT(S) arg out,arg),out)
```

which applies a clearing function of type `ClrFun S EVT(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[EVT(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 `#C(FunPtr Funtyp)` 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).

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

```
getFld_f : all(rec,fld). rec! -> fld!
setFld_f : all(rec,fld:<D). ModFun rec fld ()
exchngFld_f : all(rec,fld). ModFun rec fld fld
modifyFld_f : 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 `getFld_f` 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 `setFld_f` 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 `exchngFld_f` 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 `modifyFld_f` 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.

Defining the access functions as polymorphic in the record type has the drawback that they cannot be implemented in Cogent, although every single instance for a specific record type can be implemented in Cogent. So, an alternative approach is to define the access functions for every record type `Rec` and every field `f` of `Rec` with type `Ftp` as

```
getFld_f_InRec : Rec! -> Ftp!
setFld_f_InRec : ModFun Rec Ftp ()
```

```
exchngFld_f_InRec : ModFun Rec Ftp Ftp
modifyFld_f_InRec : all(arg,out). ChgPartFun Rec Ftp arg out
```

where `setFld_f_InRec` is only defined if `Ftp` is discardable.

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

```
getrefFld_f : all(rec,pfld). rec! -> pfld!
modrefFld_f : 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 `modrefFld_f` is invoked in the form

```
(r',o) = modrefFld_f(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) type `CArr<size> El` which is a record type wrapping the array (see Section 2.6.5). 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 2.6.5). 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 `CArrXX 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 globally or on the heap. In C, arrays can also be introduced by defining them as 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.

Since Gencot array types are implemented as Cogent records, the empty-value type corresponding to type `CArr<size> El` is generated by the macro call `EVT(CArr<size> El)` as usual.

For every Gencot array type `CArr<size> El` used in the Cogent program Gencot automatically generates instances of the functions `create` and `dispose` of the form:

```
create[EVT(CArr<size> El)]
dispose[EVT(CArr<size> El)]
```

Both functions also support multidimensional array types of the form `CArr<size1> #(CArr<size2> El)`.

Modifying Arrays

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

When C array types were mapped using abstract functions all accesses to array elements must be performed using abstract functions. Since Cogent builtin array types are used for mapping C array types, the index operator `"@"` can be used instead. In the current Cogent version (Oct 2020) the index operator does not yet work for boxed array types, therefore Gencot still uses abstract functions for array element access.

The type constructors for modification functions (see Section 2.7.4) 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 (although Cogent builtin array types support it).

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 `#(CArr<size> El)`. 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 `EVT(CArr16 U32)` can be initialized by setting all elements to `0` (the `defaultVal[U32]()`) with typechecks by the Cogent expression

```
INIT(Simp,CArr16 U32) (a,())
```

and an array `a` of pointers to integers with Cogent type `CArr16 (CPtr U32)` can be cleared with typechecks by the Cogent expression

```
CLEAR(Heap,CArr16 (CPtr 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

```
initEltsParCmb: all(evt,vvt,epe,vpe,arg:<S,out).
   IniFun evt vvt (IniFun epe vpe arg out, arg, (out,out)->out) out
clearEltsParCmb: all(vvt,evt,vpe,epe,arg:<S,out).
   ClrFun vvt evt (ClrFun vpe epe arg out, arg, (out,out)->out) out
initEltsPar: all(evt,vvt,epe,vpe,arg:<S).
   IniFun evt vvt (IniFun epe vpe arg (), arg) ()
clearEltsPar: all(vvt,evt,vpe,epe,arg:<S).
   ClrFun vvt evt (ClrFun vpe epe arg (), arg) ()
initEltsSeq: all(evt,vvt,epe,vpe,arg).
   IniFun evt vvt (IniFun epe vpe arg arg, arg) arg
clearEltsSeq: all(vvt,evt,vpe,epe,arg).
   ClrFun vvt evt (ClrFun vpe epe 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` in parallel 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 `initEltsParCmb` or `clearEltsParCmb`. 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.

All three initialization functions are only defined for the case that no error can occur. In particular, the function `initHeap` cannot be used as element initialization function. If this is required, a corresponding array initialization function must be implemented manually.

For every array type `CArr<size> El` Gencot provides corresponding instances of all six functions. In these instances all types besides `A` and `O` are uniquely determined by the array size and the element type, although this cannot be expressed by Cogent type constraints. Therefore Gencot defines in `include/gencot/CArray.cogent` the preprocessor macros

```
INITelts(<k>,<ek>,<size>,El,A,O)
CLEARelts(<k>,<ek>,<size>,El,A,O)
```

which expand to the corresponding instance specifications shown above and

```
INITTYPEelts(<k>,<ek>,<size>,El,A,O)
CLEARTYPEelts(<k>,<ek>,<size>,El,A,O)
```

which expand to their function types. Parameter `<k>` is either `ParCmb`, `Par`, or `Seq`. The additional parameter `<ek>` is required for technical reasons and specifies the kind of the element type, as for macro `CPTR` (see Section 2.6.7). It must be `U` if the element is an unboxed record type or an unboxed Gencot array type, and it must be empty otherwise. If `<k>` is not `ParCmb` the last parameter `O` is ignored and may be empty.

For example an array `a` of type `EVT(CArr16 U32)` can be initialized by setting all elements to `5` with typechecks by the Cogent expression

```
INITelts(Par,,16,U32,U32,) (a,(INIT(Full,CPtr U32),#{cont=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 `initEltsSeq` or `clearEltsSeq`, 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. The general approach here is to use the property that Gencot array types are never empty and the element at index 0 safely exists and can be used as default if the given index is invalid.

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) (idx,el)
modifyArr : all(arr,idx,el,arg,out).
  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,out).
  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 defined if `arr` is a Gencot array type. Type `idx` must be one of `U8, U16, U32, U64` according to the `<size>` of the array. Types `el` and `pel` must be as determined by `arr`. `arg` and `out` may be arbitrary types.

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. The index is always part of the result so that the additional output has the same type as the additional input, so that `exchngArr` can be used as element function also for `modifyArr` and `modrefArr`.

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 instead applies the element change function to the element with index `0`. This corresponds to the fourth solution described in Section 2.7.6 and supports arbitrary types of additional input and result.

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.

Function `modrefArr` works like `modifyArr` but uses an element modification function and passes a pointer to the element to it, so that the element is modified in-place.

The types of `modifyArr` and `modrefArr` 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 `CArr<n1> #(CArr<n2> ...)` the element type is again an (unboxed) array type. When `getrefArr` and `modrefArr` are used for the access in the outer array all access functions described here can be used for the access in the inner array.

Working with Unboxed Arrays

As for unboxed record types, Cogent provides a value constructor for unboxed array types. Both together can be used to construct values of unboxed Gencot

array types. For the type `#(CArr3 U16)` a value can be constructed by the expression

```
#{arr3 = [1,2,3]}
```

Elements of an unboxed array can be accessed using the index operator `"@"`. Thus, if the element type is not linear, elements of a Gencot array type can be accessed directly in Cogent. The second element of a value `a` of type `#(CArr3 U16)` can be accessed by the expression

```
a.arr3@2
```

If the element type is linear, the unboxed array is linear as well and the dot operator cannot be used. Instead, the array must be taken from the wrapping record, which means, the whole array must be copied even for accessing a single element. Therefore Gencot also defines and implements abstract functions for working with unboxed Gencot array types.

Gencot does not support Gencot array types with taken elements other than empty-value types for boxed arrays. Therefore no initialization or clearing operations are applicable to unboxed Gencot array types.

As for boxed Gencot array types Gencot only supports functions for accessing single elements of unboxed arrays. It supports the functions `getArr`, `getArrChk`, `setArr`, `exchngArr`, and `modifyArr`. The latter four functions return a modified copy of the array. As described in Section 2.7.4 the `ModFun` type is interpreted accordingly for unboxed values. Note that an unboxed array may still be of linear type, if the elements are of linear type. Then all the type restrictions defined for the access functions are also required for the unboxed arrays. Together with these restrictions the semantics of these functions guarantees that elements of linear type are neither shared nor discarded.

The `getref` and `modref` functions are not supported. They would return or pass a pointer to an element. Since unboxed Gencot arrays are stack allocated these would be pointers into the stack which are not supported by the Isabelle C parser.

2.7.13 Explicitly Sized Arrays

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 `p` in the form `p[i]`. The semantics is to access an element in memory at the specified offset after the element referenced by the pointer. To use this in a safe way, the number of array elements must be known.

Gencot provides support for this kind of working with arrays, if the number of elements is specified explicitly as an integer value. Basically, the element pointer and the array size are combined to a pair. The corresponding type is called an “explicitly sized array” type. Gencot uses the generic type `CArrES` to construct types for explicitly sized arrays, such as `CArrES El` for an explicitly sized array with elements of type `El`.

For the Isabelle shallow embedding the direct use of an element pointer in the pair would correspond to a pair consisting of the first element and the size. Thus, explicitly sized arrays would be considered equal, if they have the same size and the same first element. To prevent this, the element pointer type is replaced by the type `(CArrPtr El)` where `CArrPtr` is a specific generic type introduced for this purpose. As described in Section 2.6.7 for the marker type `MayNull`, defining `CArrPtr` as an abstract type can lead to violation of the Cogent type system guarantees, therefore it is defined like `CPtr` using a wrapper record:

```
type CArrPtr el = { arr: el }
```

Like `CPtr` this definition maps to a C type which is binary compatible to pointer to type `el` in C. In the shallow embedding it is mapped to type `list`. Based on this type definition it would be possible to access the first array element in Cogent, but it is not intended to be used in this way and will lead to code which cannot be verified.

Note that if `CArrPtr` is instead used as a marker type for the pointer type, the situation would be more complex: for a primitive element type an array pointer type would have the form `(CArrPtr (CPtr U32))`, for a struct element type `r` it would have the form `(CArrPtr Cogent_Struct_r)`. Both forms have to be treated differently in the shallow embedding.

As a C pointer, the pointer may be `NULL`. To express this, the type `MayNull` (see Section 2.6.7) may be applied to `CArrPtr el`. However, in explicitly sized arrays it is always assumed that the pointer is not `NULL`. If an arbitrary pointer is used to construct an explicitly sized array it should have type `MayNull (CArrPtr el)` and should be tested with `notNull` before.

The corresponding type definition for explicitly sized arrays is

```
type CArrES el = (CArrPtr el, U64)
```

To be able to define the functions `getArr`, `modifyArr`, and `modrefArr` in the same way as for fixed size arrays, explicitly sized arrays must not be empty, i.e., the size must not be `0`. This property must be proven for all uses of explicitly sized arrays.

In the same way as the macro `CARR` for fixed-sized arrays (see Section 2.6.5) Gencot provides in `include/gencot/CArray.cogent` the preprocessor macro

```
CAES(<ek>,el)
```

where `<ek>` describes the kind of element type as for the macro `CPTR` (see Section 2.6.7), i.e. it is `U` if `el` is an unboxed type and it is empty otherwise. The form `CAES(,El)` expands to `CArrES El`, the form `CAES(U,El)` expands to `CArrES #El`.

For every instance of type `CArrES` Gencot provides operations as described in the following sections.

Whenever a pointer `p` of type `(CArrPtr el)` to an element is given in a program and an expression `s` is known which specifies the corresponding number of elements in the sequence, the corresponding explicitly sized array value can be constructed manually as `(p,s)` in Cogent. To make this construction more explicit, Gencot provides in `include/gencot/CArray.cogent` the preprocessor macro

```
MKCAES(p,s)
```

which expands to the pair `(p,s)`.

Converting Arrays

Gencot provides support for converting array values between explicitly sized arrays and the fixed size arrays of types `CArr<size> El`. It supports the polymorphic abstract functions

```
toCAES: all(arr,aes). arr -> aes
fromCAES: all(aes,arr). aes -> arr
rotoCAES: all(arr,aes). arr! -> aes!
rofromCAES: all(aes,arr). aes! -> arr!
```

where `arr` is the fixed size array type and `aes` is the explicitly sized array type. Gencot provides instances for all cases where `arr` and `aes` have the same element type. Since this constraint cannot be represented in Cogent, Gencot provides the following preprocessor macros in `include/gencot/CArray.cogent`:

```
TOCAES(<size>,<ek>,El)
FROMCAES(<size>,<ek>,El)
ROTOCAES(<size>,<ek>,El)
ROFROMCAES(<size>,<ek>,El)
```

where `<size>` is the number of elements in the fixed size array type, `<ek>` is the kind of element type as above, and `El` is the element type. As example, the macro call `TOCAES(8,,U32)` expands to

```
toCAES[CArr8 U32,CArrES U32]
```

and the macro call `ROFROMCAES(5,,CPtr U16)` expands to

```
rofromCAES[CArrES (CPtr U16),CArr5 (CPtr U16)]
```

Function `toCAES` returns the pair of the pointer to the first array element and the constant known size, this always succeeds. Function `fromCAES` simply returns the casted pointer. This may be wrong, so in the shallow embedding it must be proven that the sizes are the same. An alternative would be to define `fromCAES` so that it compares the explicit size in the argument to the known fixed size of the result and, if they are equal it returns the element pointer, otherwise it returns an error value with the original input. However, the original input is linear and cannot be discarded and in the context of the invocation, the linear value of the fixed size array must be consumed, therefore such a definition would be very difficult to be used in Cogent.

Creating and Disposing Explicitly Sized Arrays

The empty-value type corresponding to an explicitly sized array type `(CArrES El)` cannot be constructed by `EVT(CArrES El)`, because that would take all fields from the *pair*, including the size, which must be present also for the empty-value type. Instead, Gencot uses the type `(EVT(CArrPtr El),U64)` as empty-value type for explicitly sized arrays. For specifying the empty-value type Gencot provides the preprocessor macro

```
EVT_CAES(<ek>,El)
```

where parameter `<ek>` is as described for `CAES` above.

For allocating an explicitly sized array the size must be specified, since it is not known as part of the type. Gencot provides the polymorphic abstract function

```
createCAES: all(evp). (U64,Heap) -> Result (evp,Heap) Heap
```

for all cases where `evp` is a type of the form `EVT_CAES(<ek>,El)`, i.e., an empty-value type for an explicitly sized array. For disposing an explicitly sized array the usual function `dispose` from Section 2.7.3 is used, since an `evp` value already contains all information required for applying function `free`.

Modifying Explicitly Sized Arrays

Explicitly sized arrays can be modified in the same way as fixed size arrays. In particular, all type constructors for modification functions (see Section 2.7.4) can be applied to explicitly sized array types with the usual intended semantics.

Initializing and Clearing Explicitly Sized Arrays

Since the array size is known at runtime, all conceptual initialization and clearing operations defined for fixed size arrays can also be provided for explicitly sized arrays. These are the general operations defined in Section 2.7.5 and the array-specific operations defined in Section 2.7.12.

The functions `initFull` and `clearFull` are not supported, since they are defined using the unboxed type `#vvt` to represent the content. For an explicitly sized array type the unboxed type is the same tuple type. Thus `initFull` has not the intended semantics of moving data from the stack to the heap. The other functions work as described.

Since the way how the empty-value type is constructed is different for explicitly sized arrays, the macros defined in Section 2.7.5 cannot be used. Gencot instead provides the macros

```
INIT_CAES(<k>,<ek>,El)
CLEAR_CAES(<k>,<ek>,El)
INITTYPE_CAES(<k>,<ek>,El)
CLEARTYPE_CAES(<k>,<ek>,El)
```

where `<k>` is one of `Heap` or `Simp`, and `<ek>` and `El` are as for `CAES` above. As for fixed size arrays, the functions treat all array elements in the same way.

The functions `initEltsParCmb`, `clearEltsParCmb`, `initEltsPar`, `clearEltsPar`, `initEltsSeq`, `clearEltsSeq` treat elements individually. They are all supported for explicitly sized array types `vvt`. Gencot provides the macros

```
INITelts_CAES(<k>,<ek>,El,A,O)
CLEARelts_CAES(<k>,<ek>,El,A,O)
INITTYPEelts_CAES(<k>,<ek>,El,A,O)
CLEARTYPEelts_CAES(<k>,<ek>,El,A,O)
```

corresponding to `INITelts`, `CLEARelts`, `INITTYPEelts`, and `CLEARTYPEelts`. As for them `<k>` is one of `ParCmb`, `Par`, or `Seq` and `A` and `O` are arbitrary types for the additional input and output of the element function. `O` is ignored if `<k>` is not `ParCmb`.

Accessing Array Elements

All element access functions defined for fixed size arrays in Section 2.7.12 are also supported for explicitly sized arrays, i.e., their type variable `arr` can be an explicitly sized array type. The index type `idx` can be as described (one of U8, U16, U32, U64).

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. Declarations embedded in a body are processed when the body is translated. They are translated to a binding of a local variable to the value specified by the initializer or to a default value (see Section 2.10).

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 2.6.4.

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 2.6.3.

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.

2.9 Processing C Function and Object 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 2.6.6. In a C function definition the types for all parameters must be specified in the parameter list, if old-style function definitions are ignored.

2.9.1 Function Bodies

In C the function body consists of a compound statement. In Cogent the function body consists of an expression. Gencot translates the C code to Cogent code as described in Section 2.10.

As described in Section 2.6.6 Gencot translates all C functions to Cogent functions with a single parameter. If the C function has more than one parameter it is a tuple of the C parameters. To make it possible to use the original C parameter names in the translated function body Gencot generates a Cogent pattern for the 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. 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 item property Global-State (see Section 2.6.1) has been used to introduce additional function parameters, the corresponding parameter names are added to the tuple after the last C parameter name. If the item property Heap-Use has been set for the function, an additional parameter named `heap` or `globheap<n>` (see 2.6.1) is added to the tuple after possible Global-State parameters. If the item property Input-Output has been set for the function, an additional parameter named `io` or `globio<n>` (see 2.6.1) is added to the tuple after possible Global-State parameters and the Heap-Use parameter.

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.

The generated Cogent function definition has the form

```
<name> :: (<ptype1>, ..., <ptypen>) -> <restype>
<name> (<pname1>, ..., <pnamen>) =
  <translated C function body>
```

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

Gencot inserts origin markers in the Cogent code generated for the function body. It also inserts them in embedded untranslated C code. Thus, comments and preprocessor directives will be reinserted into the code. However, since the structure of the Cogent code may substantially differ from that of the C code, comments and directives may be misplaced or omitted, this must be checked and handled manually.

2.9.3 Entry Wrappers

The C function definitions are also used for generating the entry wrappers as described in Section 2.1.2. For every definition of a C function with external linkage an entry wrapper is generated in antiquoted C.

The main task of the entry wrapper is to convert the seperate function arguments to an argument tuple which is passed to the corresponding Cogent function. If the C function has only one argument it is passed directly. If the C function has no argument the unit value must be passed to the Cogent function.

The entry wrapper has the same interface (name, arguments, argument types, result type) as the original C function. However, since the wrapper is a part of the Cogent compilation unit, the argument and result types used in the original function definition are not available. They must be replaced by the binary compatible types generated from their translation to Cogent types (see Section 2.6). This is accomplished by using the Cogent types in antiquoted form.

If the original C function definition has the signature

```
<type0> fnam(<type1> arg1, ..., <typen> argn)
```

the entry wrapper has the signature

```
$ty:(<cogt0>) fnam($ty:(<cogt1>) arg1, ..., $ty:(<cogtn>) argn)
```

where `<cogti>` is the translation of `<typei>` to Cogent. It would be possible to generate synthetic parameter names for the wrapper, however, since the original parameter names must always be specified in the original function definition, Gencot uses them in the wrapper signature for simplicity and better readability.

If there is more than one function argument (`n > 1`), the wrapper must convert them to the C implementation of the Cogent tuple value used as argument for the Cogent function (see Section 2.6.6). Cogent tuples are implemented as C struct types with members named `"p1"`, `"p2"`, ... in the order of the tuple components. Since tuple types are always unboxed, the C structs are allocated on the stack. Thus the body of an entry wrapper with `n > 1` has the form

```
$ty:((<cogt1>,...,<cogtn>)) arg =
  {.p1=arg1,..., .pn=argn};
return cogent_fnam(arg);
```

If `n = 1` no tuple needs to be constructed and the wrapper body has the form

```
return cogent_fnam(arg1);
```

If `n = 0` the C implementation of the Cogent unit value must be constructed. It is a C struct with a single member named `"dummy"` of type `int`. The corresponding wrapper body has the form

```
$ty:(()) arg = {.dummy=0};
return cogent_fnam(arg);
```

If the C function has no result, i.e., `<type0>` is `void`, no result is returned by the entry wrapper. Then the last statement in its body has the form

```
cogent_fnam(arg);
```

If the C function has the Heap-Use property (see Section 2.6.1), a component of type `Heap` is added to the Cogent function's argument tuple and the result is converted to a tuple where the original result is the first component. In C the heap is accessed globally, therefore these components can be ignored. However, the wrapper implementation has to take care of them. Type `Heap` is implemented by Gencot simply by the C type `int` and a `0` is passed as its value. The entry wrapper body for a function with Heap-Use property has the form

```
$ty:((<cogt1>,...,<cogtn>,Heap)) arg =
  {.p1=arg1,..., .pn=argn, .pn+1=0};
return cogent_fnam(arg).p1;
```

if `n > 0` and

```
return cogent_fnam(0).p1;
```

if `n = 0`.

If the C function has the Input-Output property (see Section 2.6.1), a component of type `SysState` is added in the same way as for Heap-Use, but after that for Heap-Use, if present. Type `SysState` is also implemented by Gencot by the C type `int` and a `0` is passed as its value.

If atleast one of the C function arguments has the Add-Result property, the result is also converted to a tuple with the original result as its first component. In this case the last statement in the wrapper's body also has the form

```
return cogent_fnam(arg).p1;
```

if the C function's result is not `void`.

If the C function has virtual parameters with Global-State properties declared in the item properties, references to the corresponding global variables are passed to these parameters by the entry wrapper. For a parameter with Global-State property the corresponding variable is determined by searching all known toplevel items for an item with a Global-State property with the same numerical argument. Since every global variable must use a different numerical argument, this search will result in a single toplevel item identifier.

If the function does not access the global variable and uses the virtual parameter only to pass the variable reference to invoked function, it may be the case that the variable is not visible in the C source file, because it is neither defined nor declared there. However, the name used in (antiquoted) C for the variable can always be constructed from the item identifier alone. No other information about the variable is needed to generate the entry wrapper.

Assume that a C function has two virtual parameters with Global-State properties with numerical arguments `0` and `3`. The global variable associated with argument `0` has name `gvar` and external linkage, the global variable associated with argument `3` has name `table` and is defined in source file `scan.c` with internal linkage. Then the entry wrapper body has the form

```
$ty:((<cogt1>,...,<cogtn>,GlobState,GlobState3)) arg =
  {.p1=arg1,..., .pn=argn, .pn+1=&gvar, .pn+2=&local_scan_table};
return cogent_fnam(arg).p1;
```

Together, the information required to generate the entry wrapper is the same as that for translating the function's type to Cogent: the argument types and the result type must be translated to Cogent and the function name must be mapped to determine the name of the Cogent function.

2.9.4 Object Definitions

Gencot processes object declarations with block-scope ("local variables") as described in Section 2.10.1.

A toplevel ("external") object definition (with "file-scope") either has an initializer or is a "tentative definition" according to the C standard. Such an object definition defines a global variable. Gencot treats global variables as described in Section 2.1.2.

If the global variable has a Global-State property (see Section 2.6.1) Gencot introduces a type synonym of the form `GlobState<i>`. The type synonym definition is generated in the translated Cogent source at the place where the object definition was in the C source and is marked with the corresponding origin. The type synonym name is determined from the numerical argument of the Global-State property, the right-hand side of the type definition is the translation of the derived pointer type of the variable's type, translated according to Section 2.6.

If the global variable has the Const-Val property (see Section 2.6.1) Gencot introduces an abstract parameterless access function in Cogent in the form

```
<name> : () -> <type>
```

where `<name>` is the mapped variable name and `<type>` is the translation of the variable's defined type. If the defined type is a struct, union, or array type, the translation of the derived pointer type is used instead, with a bang operator applied to make it readonly. The access function definition is generated in the translated Cogent source at the place where the object definition was in the C source and is marked with the corresponding origin.

Independent from its item properties, the object definition is translated to antiquoted C and added to the file containing the entry wrappers. In this way all global variables are still present in the Cogent compilation unit. The translated definition uses as type the antiquoted Cogent type which results from translating the type specified in the object definition. For an object with external linkage the name is the original name, so that the object can still be referred from outside the Cogent compilation unit, and the linkage is external. For an object with internal linkage the name is mapped as described in Section 2.1.1 so that it is unique in the Cogent compilation unit, and the linkage is internal. If an initializer is present in the original definition it is transferred to the antiquoted definition, mapping all referenced names of global variables with internal linkage.

The entry wrappers and object definitions are placed in the antiquoted C file in the same order as the original function and object definitions. Additionally, all global variables are declared together with all external functions in a file included before all entry wrapper files, so that entry wrappers defined before the variable can still pass a pointer to it to the wrapped Cogent function.

If the global variable has the Const-Val property Gencot additionally creates an implementation of the access function in antiquoted C and puts it immediately after the variable definition. Depending on the variable's type it returns its value or a pointer to the variable.

2.10 Processing C Statements and Expressions

In C statements and expressions occur in the bodies of function definitions and in initializers of object definitions. Gencot translates both to corresponding

Cogent constructs, as described in Section 2.9. In both cases the C statements and expressions must be translated to Cogent expressions. This means a transformation from imperative constructs with possible side effects in a global state to purely functional constructs where all effects are explicitly represented in the construct itself. It also means a transformation from arbitrary treatment of pointers to the Cogent uniqueness type system where pointers are never shared or discarded.

The first issue is handled by Gencot. In some cases a translation is not (yet) implemented. In such cases the generated Cogent code contains a dummy expression which documents the reason for not translating it automatically, together with the corresponding C code as a comment. So the generated Cogent code is always syntactically correct and can be parsed by the Cogent compiler, although it may not reflect the complete original C code.

The second issue is not handled by Gencot. If the C code involves sharing or discarding of pointers the translation will result in Cogent code which violates the uniqueness type assumptions. This will be detected by the Cogent compiler in its type checking phase and must be handled manually, either by modifying and retranslating the C code, or by modifying the Cogent code.

2.10.1 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.

Global variables can be turned to parameters or access functions using the Global-State and Const-Val properties (see Section 2.6.1). If neither property has been specified for a global variable, Gencot does not translate accesses to it and inserts a dummy expression instead.

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.

Gencot assumes that 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.

If the assumption is not true the Cogent Compiler will detect this when it typechecks the translated code. See Section 4.7.1 for proposals how to handle such cases manually.

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. If `init` is an expression this is valid C code. Otherwise, if `init` is an initializer for a struct or array, the statement is not valid in C, but Gencot 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 2.7.12, or the initialization can be done by several applications of operation `set` (see Section 2.7.6).

For a declaration without an initializer Gencot inserts an assignment with a default value. If the variable has a regular type, the Gencot operation `defaultVal` (see Section 2.7.2) is used. If the variable has linear type (is a pointer or directly contains pointers) the contained pointers are translated to `MayNull` types (see Section 2.6.7) and are initialized to value `null()` (see Section 2.7.10). If a NotNull property (see Section 2.6.1) is used to avoid the `MayNull` type Gencot will not translate the declaration if it has no initializer and will generate a dummy expression instead.

Together, this way all declarations are replaced by statements and 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 (including pseudo assignment statements corresponding to record and array initializers).

2.10.2 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 4.7.4.

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(modifyFld_a(s,(modifyArr,(i,setFld_x,v)))))
```

where `modifyFld_a` and `setFld_x` are abstract modification functions for the fields `a` and `x`, respectively, as described in Section 2.7.11.

This translation is correct, but it is inefficient because `modifyFld_a` will copy the whole array `a` only to access one element of it by applying `modifyArr` to the unboxed array. Using `modref` (see Section 2.7.4) this can be improved to

```
(v,s) = let v = 5
  in (v,fst(modrefFld_a(s,(modifyArr,(i,setFld_x,v)))))
```

here `modrefFld_a` retrieves only the pointer to `a` and applies `modifyArr` to the boxed array. If the array element would be a larger data structure it could be further improved by using `modrefArr` instead of `modifyArr`, here it does not pay because the array element is already a single pointer.

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 `modifyFld_mi`, however it prevents the improvement described above by using `modref`.

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.

Again, this translation may be improved by using `modref` instead of `modify` where appropriate.

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}...} = v1
and ...
and vn{mn1=...{mnkn}...} = vn
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} = s
and t{n} = t
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) =
  modifyFld_a(s,(modifyArr,(i,modifyFld_x,(f,(5,z)))))
  in (v,s)
```

If index `i` is invalid, according to the definition of `modifyArr` the function call `f(5,s->a[0]->x,z)` is executed instead.

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. If `e0` evaluates to a value of numerical type the translation of the conditional expression is

```
(v,v1,...,vk) =
  let (x,x1,...,xn) = expr0
  and (v,u1,...,uq) =
    if x /= 0 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.

If the outermost operator of `e0` is a "logical" operator (one of `<`, `<=`, `>`, `>=`, `==`, `!=`, `&&`, `||`) it can be translated in a way that the first component in the result of `expr0` is of type Bool. Then, instead of `if x /= 0 then` the condition is written as `if x then`.

In C the value used for testing the condition may be of any scalar type, i.e., a numerical or pointer type. In case of a pointer type the condition corresponds to testing the pointer for being not `NULL`. The translated expression `expr0` will return a value of linear type as first component. Since it may be null (otherwise the conditional expression could be replaced by `e1`) it must be translated so that `expr0` returns a value of type `MayNull a` (see Section 2.7.10) as its first component. Since the use as condition in Cogent would discard the value, it must be readonly, i.e., of type `(MayNull a)!`. Then, instead of `if x /= 0 then` the condition must be written as

```
if not isNull(x) then
```

using function `isNull` as described in Section 2.7.10. If the first component of the result of `expr0` is not already of readonly type it must be made readonly in the condition:

```
if not isNull(x) !x then
```

Often in C, after testing x successfully for not being `NULL`, the pointer `x` is used in `e1` by dereferencing it. In Cogent this is not possible, since a value of type `MayNull a` cannot be dereferenced. Instead, a value of type `a` is required. This is made accessible by the functions `notNull` and `roNotNull` (see Section 2.7.10). Then the translation of the conditional expression has the form

```
(v,v1,...,vk) =
  let (x,x1,...,xn) = expr0
  and (v,u1,...,uq) =
    notNull(x)
    | Some x ->
```

```
      let (y,y1,...,ym) = expr1
      in (y,u1,...,uq)
    | None ->
      let (z,z1,...,zp) = expr2
      in (z,u1,...,uq)
in (v,v1,...,vk)
```

Here the `x` introduced by matching `Some x` is of type `a` and can be used in `expr1` to access and modify it. If `expr1` only reads `x` and should not consume it, it must be introduced as readonly by replacing the line `notNull(x)` by

```
roNotNull(x) !x
```

Then the `x` introduced by matching `Some x` is of type `a!`. Function `roNotNull` must also be used when the outer `x` is already of the readonly type `(MayNull a)!`.

In the following patterns we only show the form `if x /= 0 then`, it must be replaced as needed.

Usually, the condition `e0` has no side effects, the the translation can be simplified to

```
(v,u1,...,uq) =
  if expr0 /= 0 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 /= 0 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)
```

For these translations the way how the Cogent compiler is used is relevant. When it 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 { v = b; rest}
else { v = c; rest}
```

duplicating the translated code for `rest` (which is called "a-normal form" in Cogent). 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 Cogent compiler must be used with the flag `-fnormalisation=knf` which translates to "k-normal form"

```
if a { v = b}
else { v = c}
rest
```

Alternatively, 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`, even if a-normal form is used.

2.10.3 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 by adding a component to the `pattern`, so that the translation becomes

```
(c,v1,...,vn) = expr
```

The variable `c` is a new variable which is not already bound in the context of the expression, it is used to bind the information about control flow modification by 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 single variable `c`. In the following descriptions we always use a pattern of the form `(c,v1,...,vn)` with the meaning that for `n=0` it is the single variable `c`.

The value bound to `c` is a tuple of type

```
(Bool, Bool, Option T)
```

where `T` is the original result type of the surrounding function. In a value `(cc,cb,res)` the component `cc` is true if the statement contains a `return`, `break`, or `continue` statement outside of a `switch` or loop statement. The component `cb` is true if the statement contains a `return` or `break` statement outside of a `switch` or loop statement. The component `res` is the value `None` if the statement contains no `return` statement, otherwise it is the value `Some v` where `v` is the value to be returned by the function.

The translation of `goto` statement and labels is not supported, this must be done manually.

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. Simple statements cannot modify the control flow, they always bind the variable `c` to the value `(False,False,None)` which we abbreviate as `ffn` in the following.

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 2.10.2. Let its translation be `(v,v1,...,vn) = expr`. Then the binding for the statement `e;` is

```
(c,v1,...,vn) =
  let (v,v1,...,vn) = expr
  in (ffn,v1,...,vn)
```

The result `v` of `e` is discarded, the same considerations apply here as described for expressions using the comma operator in Section 2.10.2.

If `expr` is a tuple `(e0,e1,...,en)` the binding can further be simplified to

```
(c,v1,...,vn) = (ffn,e1,...,en)
```

As an example, the translation of the simple statement `i++;` is

```
(c,i) = let (v,i) = (i,i+1) in (ffn,i)
```

which can be simplified to

```
(c,i) = (ffn,i+1)
```

The empty C statement `;` is translated as

```
c = ffn
```

Jump Statements

A jump statement is a `return` statement, `break` statement, or `continue` statement.

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 with `()` as first component.

The translation of a return statement of the form `return;` is the binding

```
c = (True,True,Some ())
```

If `e` has no side effects, the translation of statement `return e;` is

```
c = (True,True,Some expr)
```

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

```
(c,v1,...,vn) =
  let (v,v1,...,vn) = expr
  in ((True,True,Some v), v1,...,vn)
```

which can be simplified if `expr` is the tuple `(e0,e1,...,en)` to

```
(c,v1,...,vn) = ((True,True,Some e0),e1,...,en)
```

A C `break` statement has the form `break;`. It ends the next surrounding `switch` or loop statement, otherwise it may not be used. It is translated to

```
c = (True,True,None)
```

A C `continue` statement has the form `continue;`. It ends the body of the next surrounding loop statement, otherwise it may not be used. It is translated to

```
c = (True,False,None)
```

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 `(c1,v1,...,vn) = expr1` be the translation of `s1` and `(c2,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

```
(c,u1,...,uk) =
  if expr /= 0 then
    let (c1,v1,...,vn) = expr1
    in (c1,u1,...,uk)
  else
    let (c2,w1,...,wm) = expr2
    in (c2,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 2.10.2. In particular, the condition test `if expr /= 0 then` must be replaced by `if expr then` or `if not isNull(expr) then` or the forms with `notNull` or `roNotNull` as needed.

The first form of the conditional statement is translated as

```
(c,v1,...vn) =
  if expr /= 0 then expr1
  else ((False,False,None),v1,...vn)
```

If expression `e` has side effects the translation is extended as for the conditional expression.

As an example the translation of the conditional statement `if (i==0) return a; else a++;` is the binding

```
(c,a) =
  if i==0 then
    let c = (True,True,Some a)
    in (c,a)
  else
    let (c,a) = ((False,False,None),a+1)
    in (c,a)
```

which can be simplified to

```
(c,a) =
  if i==0 then ((True,True,Some a),a)
  else ((False,False,None),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 2.10.1: 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 `(c,v1,...,vn) = expr1` be the translation of statement `s1` and `(c,w1,...,wm) = expr2` be the translation of statement `s2`. Then the translation of `{ s1 s2 }` is

```
(c,u1,...,uk) =
  let ((cc,cb,res),v1,...,vn) = expr1
  in if cc then ((cc,cb,res),u1,...,uk)
  else let (c,w1,...,wm) = expr2
  in (c,u1,...,uk)
```

where `u1,...,uk` is the union of `v1,...,vn` and `w1,...,wm` in some order without the local variables declared in the block.

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 4.7.4.

As an example, the compound statement `{ if (i==0) return a; else a++; b = a; }` is translated to the binding

```
(c,a,b) =
  let ((cc,cb,res),a) =
    if i==0 then ((True,True,Some a),a)
    else ((False,False,None),a+1)
  in if cc then ((cc,cb,res),a,b)
  else let (c,b) = ((False,False,None),a)
  in (c,a,b)
```

If `s1` contains no jump statement the variable `cc` is bound to `False` and the translation can be simplified to the binding

```
(c,u1,...,uk) =
  let (c,v1,...,vn) = expr1
  and (c,w1,...,wm) = expr2
  in (c,u1,...,uk)
```

As an example, the compound statement `{ int i = 0; a[i++] = 5; b = i + 3; }` is translated to the simplified binding

```
(c,a,b) =
  let (c,i) = (ffn,0)
  and (c,i,a) = (ffn,i+1,setArr(a,i,5))
  and (c,b) = (ffn,i+3)
  in (c,a,b)
```

which can be further simplified to

```
(c,a,b) =
  let i = 0
  and (i,a) = (i+1,setArr(a,i,5))
  and b = i+3
  in (ffn,a,b)
```

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 `(c,v1,...,vn) = expr`, the body is translated to the expression

```
let (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, `c` contains the value `Some ()`.

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 4.7.4. 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 ((_,_,res),v1,...,vn) = expr
in res | None -> (defaultVal(),pm1,...,pmn)
       | Some v -> (v,pm1,...,pmn)
```

where `defaultVal()` is as defined in Section 2.7.2. If the type of the original function result `v` is not regular (in C: contains a pointer) `defaultVal` cannot be used, in this case a different expression for the `None` alternative must be specified manually. Note that in this case the `None` alternative is never used, if the C program is correct. Nevertheless, for Cogent to be correct some result value must be specified also for this case, it is irrelevant which value is used.

Switch Statements

A `switch` statement in C specifies a "control expression" of integer type and a body statement. Control jumps to a `case` statement in the body marked with the value of the control expression or to a `default` statement. If neither is present in the body it is not executed.

The `case` and `default` statements may occur anywhere in the body other than in a nested `switch` statement, in particular they may occur in nested

conditional statements and in loop statements. In these cases the jump is past the condition or loop initialization. The resulting behavior is clearly defined in the C standard, however, it is difficult to be transferred to a functional specification and usually involves code copying. The jump targets may also occur in a nested compound statement so that the jump may be past some declarations into an inner scope, this is also difficult to transfer.

However, there is a simple form of `switch` statements which can be transferred in a rather straightforward way. It is the case where the body is a compound statement and all `case` and `default` statements occur directly in the sequence of contained statements. These `switch` statements have the form

```
switch (e) {
  ...
case c1: s11 ... s1k1
...
case cn: sn1 ... snkn
default: s01 ... s0k0
}
```

where the `default` statement may have any other position or may be omitted. The C standard requires that all `ci` constants are pairwise distinct. In practical applications most `switch` statements are of this form.

If there are declarations among the `sij` or before the first `case` statement the `switch` statement may cause a jump past the declaration which means the declared object exists but is not initialized, which is also difficult to transfer. Therefore we assume that there are no declarations in the body.

Then the `switch` statement of the form shown above can be rewritten to the following statement

```
{ t v = e;
  if (v == c1) {s11 ... s1k1}
  ...
  if (v == c1 || ... || v == cn) {sn1 ... snkn}
  {s01 ... s0k0}
}
```

and can be translated as described in the corresponding sections. Here `v` is an otherwise unused variable and `t` is the type of `e`. It is needed even if `e` is a single variable, because this variable could be modified in some cases but in the following case conditions the original value must be used.

Note that the additional disjunctions and the omitted condition for the default part cover the feature of "fall through" in a `switch` statement: the execution of following cases is only prevented by an explicit `break` or `return` statement. If the `default` statement is placed before other `case` statements it also needs a condition which prevents its execution for the cases following it.

Since `break` statements which are not enclosed by a nested loop or `switch` statement are consumed by the `switch` statement and not propagated to the context, the control flow tuple must be reset by the switch statement. If it is bound to `(cc,cb,res)` by the body, it must be rebound by the `switch` statement to

```
(res /= None || (cc && not cb),res /= None, res)
```

which will only preserve the effect of `return` and `continue` statements. A `continue` statement does not affect the `switch` statement but may be present and affect a surrounding loop.

As an example, the `switch` statement

```
switch (i) {
  case 1: return 0;
  case 2: i++;
  case 3: return i+10;
  default: return i;
}
```

can be rewritten to

```
{
  if (v==1) return 0;
  if (v==1 || v==2) i++;
  if (v==1 || v==2 || v==3) return i+10;
  return i;
}
```

which is translated to

```
(c,i) = let v = i in
  let (cc,cb,res) =
    if v==1 then (True,True,Some 0)
            else (False,False,None)
  in if cc then ((cc,cb,res),i) else
    let ((cc,cb,res),i) =
      if v==1 || v==2 then ((False,False,None),i+1)
                      else ((False,False,None),i)
    in if cc then ((cc,cb,res),i) else
      let c =
        let (cc,cb,res) =
          if v==1 || v==2 || v==3
          then (True,True,Some i+10)
          else (False,False,None)
        in if cc then (cc,cb,res)
                 else (True,True,Some i)
      in (c,i)
```

which could be simplified to

```
(c,i) =
  ((True,True,
    if i==1 then Some 0
    else if i==2 then Some 13
    else if i==3 then Some 13
    else Some i),
   if i==2 then 3 else i)
```

For Loops

A `for` loop has the form `for (ee1; ee2; ee3) s` with optional expressions `eei` and a statement `s`. The first expression `ee1` can also be a declaration. The `for` loop can always be transformed to a `while` loop of the form

```
ee1;
while (ee2) { s ee3; }
```

therefore `ee2` must evaluate to a scalar value, as for the condition in a conditional expression or statement.

Since `while` loops are not directly supported by Cogent and need additional work for proofs when translated by using an abstract function, we do not use this transformation in general, instead, for certain forms of `for` loops we translate them using functions from the Cogent standard library. This is possible, whenever a `for` loop is a "counted" loop of the form

```
for ( w=e1; w<e2; w=e3) s
```

with a counting variable `w` of an unsigned integer type. The only free variable in `e3` must be `w` and `w` must not occur free in `e2`. The `e1, e2, e3` must not have side effects.

If a `for` loop is not of this form it may be possible to rewrite it in C to that form. Specifically, this can be done if it is possible to rewrite the expressions `eei` as follows:

```
ee1: ee1', w=e1
ee2: w<e2 && ee2'
ee3: ee3', w=e3
```

Then the loop `for (ee1; ee2; ee3) s` can be rewritten as the counted loop

```
ee1';
for (w=e1; w<e2; w=e3) {
  if (!ee2') break;
  s ee3'
}
```

In other cases, especially if some of the `eei` are empty, it may still be possible to rewrite the loop as counted by moving parts from before the loop or from the loop body into the `ei`.

Let `expri` be the translation of the `ei` and let `(c,v1,...,vn) = expr` be the translation of statement `s`. Let `w1,...,wm` be the variables occurring free in `expr` other than `w` and the `vi`. Then the translation of the counted loop

```
for (w=e1; w<e2; w=e3) s
```

is

```
(c, v1,...,vn) =
let ((v1,...,vn),lr) =
  seq32_stepf #{
    frm = expr1, to = expr2,
    stepf = \w => expr3,
```

```
      acc = (v1,...,vn), obsv = (w1,...,wm),
      f = \#{acc = (v1,...,vn), obsv = (w1,...,wm),
             idx = w} =>
        let ((cc,cb,res),v1,...,vn) = expr
        in ((v1,...,vn),
            if cb then Break (res) else Iterate ())
      }
  in lr | Iterate () -> ((False,False,None), v1,...,vn)
        | Break (None) -> ((False,False,None), v1,...,vn)
        | Break (res) -> ((True,True,res), v1,...,vn)
```

The function `seq32_stepf` is defined in the Cogent standard library in `gum/common/common.cogent`.

The variables `w,w1,...,wm,v1,...,vn` either occur free in the loop body or are modified in the loop body. In both cases they must be declared in C before the counted loop, so they are defined in Cogent and can be used when `acc` and `obsv` are set. The `w1,...,wm` must not be of linear type. Since they are used in the loop body, depending on how often the body is executed they could either be used several times or not at all. If a `wi` is of linear type the loop cannot be translated in this way and must be handled manually.

Since all `w1,...,wm` have a regular or readonly type the tuple `(w1,...,wm)` is a valid value for both `obsv` fields which have the readonly type `obsv!`.

The translation correctly handles all `return`, `break`, and `continue` statements which may occur in the body.

Note that the translation is only possible if the type of the counting variable `w` can be represented in Cogent by the type `U32`.

Since `seq32_stepf` is an abstract function the Cogent compiler does not generate a refinement proof for it. The refinement proof for `seq32_stepf` can only be successful if the loop terminates. This depends on the expression `e3`. As usual, if it strictly counts the variable `w` up, and some additional assumptions about the modulo calculation for `U32` are valid, termination can be proved.

If the expression `w=e3` can be rewritten as `w+=e3'` where `w` does not occur free in `e3'` the counted loop can be translated using function `seq32` instead of `seq32_stepf`. The translation has the same form as above, with the line setting `stepf` replaced by

```
      step = expr3',
```

where `expr3'` is the translation of `e3'`. Here, arbitrary other variables may occur free in `expr3'`.

In this case the counting variable `w` may also have a type which can be represented in Cogent by the type `U64`. Then the translation has the same form but uses the function `seq64` instead of `seq32`.

If the expression `w=e3` can be rewritten as `w-=e3'` where `w` does not occur free in `e3'` and the expression `ee2` can be rewritten as `w>e2 && ee2'` the counted loop can be translated as for `w+=e3'` using the function `seq32_rev` instead of `seq32`.

If the translation can be done using `seq32` and the body does not contain any `break` or `return` statement the translation can be simplified using the function `seq32_simpl` in the form

```
(c, v1,...,vn) =
let w = expr1
and (v1,...,vn,w1,...,wm,w) =
  seq32_simple #{
    frm = w, to = expr2, step = expr3',
    acc = (v1,...,vn,w1,...,wm,w),
    f = \(v1,...,vn,w1,...,wm,w) =>
      let (_,v1,...,vn) = expr
      in (v1,...,vn,w1,...,wm,w+expr3')
    }
in ((False,False,None), v1,...,vn)
```

Note that `seq32_simple` does not pass the counting variable `w` to the body, so we must add it to `acc` and count it explicitly in `f`. If the counting variable `w` is not used in the body (does not occur free in `expr`) this can further be simplified to

```
(c, v1,...,vn) =
let (v1,...,vn,w1,...,wm) =
  seq32_simple #{
    frm = expr1, to = expr2, step = expr3',
    acc = (v1,...,vn,w1,...,wm),
    f = \(v1,...,vn,w1,...,wm) =>
      let (_,v1,...,vn) = expr
      in (v1,...,vn,w1,...,wm)
    }
in ((False,False,None), v1,...,vn)
```

As an example, the counted loop

```
for (i=0; i<size; i++) {
  sum += a[i];
}
```

is translated to

```
(c, sum) =
let i = 0
and (sum,a,i) =
  seq32_simple #{
    frm = i, to = size, step = 1,
    acc = (sum,a,i),
    f = \(sum,a,i) =>
      let sum = sum + getArr(a,i)
      in (sum,a,i+1)
    }
in ((False,False,None), sum)
```

where `a` must have a readonly Gencot array type.

For the counted loop

```
for (i=0; i<size; i++) {
  if (a[i] == trg) return i;
}
```

the body translates to

```
c = if getArr(a,i) == trg
    then (True,True,Some i)
    else (False,False,None)
```

where again `a` must be of a readonly Gencot array type. The translation of the loop is

```
c =
let ((),lr) =
  seq32 #{
    frm = 0, to = size, step = 1,
    acc = (), obsv = (a,trg),
    f = \#{acc = (), obsv = (a,trg), idx = i} =>
      let (cc,cb,res) =
        if getArr(a,i) == trg
        then (True,True,Some i)
        else (False,False,None)
      in ((),if cb then Break (res) else Iterate ())
    }
in lr | Iterate () -> (False,False,None)
      | Break (None) -> (False,False,None)
      | Break (res) -> (True,True,res)
```

which can be simplified to

```
c =
let ((),lr) =
  seq32 #{
    frm = 0, to = size, step = 1,
    acc = (), obsv = (a,trg),
    f = \#{obsv = (a,trg), idx = i} =>
      ((),if getArr(a,i) == trg
          then Break (Some i)
          else Iterate ())
    }
in lr | Iterate () -> (False,False,None)
      | Break (None) -> (False,False,None)
      | Break (res) -> (True,True,res)
```

Since end of 2021 there is a new abstract function `repeat` for loops in Cogent for which a general refinement proof has been developed. It is intended to be added to the Cogent standard library. It is even more general than `seq32_stepf` and can be used to translate all `for` loops for which an upper limit of the number of iterations can be calculated. This includes the terminating counted `for` loops described above.

A general `for` loop `for (ee1; ee2; ee3) s` where `ee2` has no side effects is first rewritten to

```
ee1;
for (; ee2; ee3) s
```

by moving the initialization expression or declaration before the loop.

The `repeat` function has two functions as argument: the `step` function which corresponds to a single execution of the loop body, and the `stop` function which tests whether the loop should stop prematurely. It would be straightforward to put the loop condition `ee2` into the `stop` function. However, then the result of a test which determines that a pointer is not NULL cannot be used in the `step` function to derive this property where the pointer is used. Therefore the loop is further rewritten to

```
ee1;
for (; ; ee3)
  if ee2 then s else break;
```

so that `ee2` becomes a part of the loop body and therefore of the `step` function.

Let `(c,v1,...,vn) = expr` be the translation of the statement sequence `{if ee2 then s else break; ee3}`. Let `w1,...,wm` be the variables occurring free in `expr` other than `c` and the `vi`. Then the translation of the remaining loop

```
for (; ; ee3)
  if ee2 then s else break;
```

is

```
(c, v1,...,vn) =
let ((_,_,res),v1,...,vn) = repeat #{
  n = exprmax,
  stop = \#{acc = ((_,cb,_),v1,...,vn),
            obsv = (w1,...,wm)} => cb,
  step = \#{acc = (_,v1,...,vn),
            obsv = (w1,...,wm)} => expr
  acc = ((False,False,None),v1,...,vn), obsv = (w1,...,wm)
  }
in ((res /= None,res /= None,res),v1,...,vn)
```

where `exprmax` is an expression for calculating the upper limit of the number of iterations. The `repeat` function is defined in `C` using a `for` loop of the form `for (i=0; i<n; i++)`, so it iterates atmost `n` times. If the loop has the form `for (w=e1;w<e2;w+=e3') s` where `w` does not occur free in `e1`, `e2`, and `e3'` the expression `exprmax` can be constructed by translating the `C` expression

```
(e2 - e1) / e3'
```

In other cases the expression for the upper limit must be determined manually.

The `repeat` function passes an "accumulator" `acc` and an "observed object" `obsv` through all iterations, were `acc` may be modified and `obsv` not. The `repeat` function takes as arguments the initial values of `acc` and `obsv` and the functions `stop` and `step` which both have `acc` and `obsv` as arguments. The `stop` function is used before each iteration to determine whether a next iteration should be executed, it returns a boolean value whether to stop. The `step` function covers the effect of one loop iteration and returns the new `acc` value. In the translation the `acc` value includes the control variable `c`. The `stop` function inspects the value of its `cb` component to detect whether a `break`

or `return` statement has been executed in the body and terminates the loop in that case.

Note that this translation may not be fully equivalent to the original C code. If the expression `e3'` evaluates to a value greater 1, the increment of `w` may cause an overflow of the word arithmetics, even if `w<e2` holds before. Then the loop will not terminate in that step and possibly never. The translation described above will instead terminate safely after `n` iterations. Since this situation is normally a programming error in C this is not viewed as a problem.

While Loops

A `while` loop has the form `while (e) s` or `do s while (e);`. Since Cogent does not support recursive function definitions these loops cannot be translated in a natural way to Cogent.

To translate a `while` loop it may be rewritten in C as a counted `for` loop which can then be translated as described in the previous section. If that is not possible the loop must be put in an abstract function which is defined in C using the original `while` loop.

Alternatively, if an upper limit of the number of iterations can be determined, the loop can be translated using the abstract `repeat` function described above as follows.

Let `(c,v1,...,vn) = expr` be the translation of statement `if e then s else break;`. Let `w1,...,wm` be the variables occurring free in `expr` other than `c` and the `vi`. Then the translation of the loop `while (e) s` is

```
(c, v1,...,vn) =
let ((_,cb,_),v1,...,vn) = repeat #{
  n = exprmax,
  stop = \#{acc = ((_,cb,_),v1,...,vn),
            obsv = (w1,...,wm)} => cb,
  step = \#{acc = (_,v1,...,vn),
            obsv = (w1,...,wm)} => expr
  acc = (c,v1,...,vn), obsv = (w1,...,wm)
  }
in ((res /= None,res /= None,res),v1,...,vn)
```

where `exprmax` is an expression which calculates an upper limit for the number of loop iterations.

2.10.4 Dummy Expressions

Dummy expressions are generated when Gencot cannot translate a C declaration, statement, or expression. It consist of a Cogent expression of the form

```
gencotDummy "... reason for not translating ..."
{- ... untranslated C code ... -}
```

(see Section 2.7.1) with the C code contained as a comment.

The C code is intended for the programmer who has to deal with it manually. Therefore it should be readable, even if it is a larger C fragment. Therefore Gencot provides the following features for it:

- generate origin markers so that comments and preprocessor directives are inserted,
- map C names to Cogent names so that they can be related to the surrounding Cogent code.

Origin Markers

To preserve comments embedded in the embedded C code it is also considered as a structured source code part and origin markers are inserted around its subparts. 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.

Mapping Names

To relate the names in the C code to the Cogent code, Gencot substitutes names occurring free in the C code. These may be names with global scope (for types, functions, tags, global variables, enum constants or preprocessor constants) or names of parameters and local variables. 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.

Names with local scope (either local variables or types/tags) are also mapped by Gencot, both in their (local) definition and whenever they are used.

The mapping 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 for embedded C code. As result, an embedded 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 `#(CArr5 Struct_s1)`, respectively. This translation for the derived types must be done manually.

Global variables referenced in a function body are treated in a specific way. If the global variable has a Global-State property (see Section 2.6.1) and a virtual

parameter item with the same property has been declared for the function, the identifier of the global variable is replaced by the mapped parameter name with the dereference operator applied. This corresponds to the intended purpose of the Global-State properties: instead of accessing global variables directly from within a function, a pointer is passed as parameter and the variable is accessed by dereferencing this pointer.

If a global variable has the Const-Val property (see Section 2.6.1) its access is replaced by an invocation of the corresponding parameterless access function. If the global variable has neither of these properties declared, Gencot simply uses its mapped name, this case must be handled manually.

2.11 Function Parameter Modifications

If a function parameter of linear type is modified by a function it can be returned as part of the function result, as described in Section 2.6.6. This is done by declaring the item property Add-Result (see Section 2.6.1) for the parameter. Gencot automatically declares this property for all parameters of linear type and provides specific support to suppress its effect by either adding the Read-Only property or by omitting the Add-Result property.

The Read-Only property can only be declared, if the parameter value is not modified by the C function implementation. 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.

The Add-Result property must be omitted if a parameter is actually discarded by a function. This is only the case for the external C standard function `free` or for parameters which are directly or indirectly passed to `free`.

2.11.1 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 used by Gencot to declare the Read-Only and remove the Add-Result property for all unmodified parameters.

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.

Before Gencot uses the JSON descriptions 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.

Gencot generates the JSON descriptions from function definitions, so that it can analyse the function body for detecting parameter modifications and invocations of other functions. 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.

2.11.2 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.

2.11.3 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

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

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 item identifier of the struct or union type.

2.11.4 Described Entities

Gencot uses the information about parameter modifications to declare Read-Only properties or omit Add-Result properties for the parameters.

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 declaring item properties for parameters.

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. For all other items (in the sense of Section 2.6.1) which have a function pointer type Gencot provides a JSON description. The description is provided independent of whether the function has parameters of linear type or not. If not, it is not needed by Gencot for translating the item's type, however it may be needed as required invocation for calculating information for parameters depending on it.

If a typedef name is defined in C for a function type or a type derived from a function type, the parameters are collective items and properties can only be declared for them collectively. In this case Gencot does not provide a JSON description for the individual item(s) to which the typedef name applies, but instead a JSON description for the typedef name as a collective item. This is only done if the function type occurs directly in the typedef name's definition, if it is specified by another typedef name the JSON description is only provided for that other name.

If a type name is defined external to the Cogent compilation unit and is only used indirectly, it is resolved and not used in the Cogent code. As described for items in Section 2.6.1, then collective sub-items of the type are replaced by individual sub-items of the items declared to have that type. Gencot treats parameter modification descriptions in the same way and uses different individual descriptions, if function items use a common function type name which is resolved because it is external.

2.11.5 Heap Usage

Another information about function parameters needed by Gencot for the translation is whether the heap is used by the function and must be passed in and out as parameter and result. This is specified to Gencot by the item property Heap-Use (see Section 2.6.1). Similar to parameter modification, heap use is transitive: a function uses the heap if it directly allocates or deallocates data, or if it invokes a function which uses the heap. The transitive heap use can be detected, when Gencot calculates the transitive closure of the function invocations. Therefore, Gencot extends the JSON descriptions to include information about the heap usage and uses it to declare Heap-Use properties.

According to the C standard there is only a small fixed number of memory management functions: `malloc`, `calloc`, `realloc`, `aligned_alloc`, and `free`. Thus, Gencot can detect that a function directly needs the heap, when it invokes any of these five functions in its body. If it invokes no functions at all, it is clear that it does not use the heap.

In all other cases, when a function only invokes other functions than memory management functions, heap usage cannot be decided locally, it depends on the heap usage of the invoked function. In this case `all` invocations are required for deciding heap usage. This potentially massively enlarges the number of relevant invocations in relation to those described in Section 2.11.2. This is treated by Gencot by giving the developer the possibility to mark invocations as not relevant for heap use dependency. In this way the developer may manually prune invocation chains in order to reduce the number of source files to process. However, this must be done with the help of background knowledge, it cannot be done by locally inspecting the function body.

For invoked function pointers, like parameter modification, heap usage must always be specified manually. The same holds basically for invocations of functions external to the <package>. However, to reduce manually specifying heap use, Gencot uses a list of standard system functions which are known to not use the heap. For these function the heap use information is entered automatically.

2.12 Extending the Isabelle Code

After Gencot has translated C code to Cogent, the Cogent compiler will generate a logic based representation of the code and a proof that the generated C code refines that representation. Both the representation ("shallow embedding") and the proofs are generated in Isabelle notation.

However, the abstract types and functions used in the Cogent code are not covered by this Isabelle code. Since Gencot uses several specific abstract types and functions to represent the C types (as described in Sections 2.6 and 2.7), the corresponding representations and refinement proofs must be added to the Isabelle code to fully cover a C program translated by Gencot. Since the semantics of these types and functions is defined by Gencot, Gencot is able to automatically generate these parts of the Isabelle code.

2.12.1 Representing Gencot Types

Specific treatment is necessary whenever Gencot uses abstract types to represent a C type in Cogent. Most C types are mapped to Cogent using the normal

Cogent types in specific ways. In these cases the Cogent compiler already generates corresponding Isabelle code. However, also in some of these cases Gencot modifies that code.

Isabelle Types Generated by Cogent

The Isabelle code for representing all Cogent types is generated by Cogent in a file `X_ShallowShared_Tuples.thy` where `X` is a program specific prefix. The code consists of

- a type declaration for every abstract Cogent type, using the same type name as in Cogent.
- a record definition for every equivalence class of record types occurring in the Cogent program. Two record types are equivalent, if they have the same field names in the same order. If there is a type synonym defined for the record type in Cogent (one of) the synonym name is used for the record, otherwise a generated name of the form `T<nr>` is used. For every Cogent field name `x` the name $\texttt{x}_\texttt{f}$ is used in Isabelle.

  Additionally, Cogent tuple types are represented by records with field names `p1`, `p2`, ... and the corresponding records are defined. The record for pairs always has the name `RR`, the names for the other tuple records are determined as described above.
- a datatype definition for every variant type occurring in the Cogent program. The type name is determined as for the records. As names for the alternative constructors the Cogent tag names are used.
- a type synonym definition for every Cogent type definition. For every Cogent type name `X` the type synonym $\texttt{X}_\texttt{T}$ is defined in Isabelle.

Since Gencot uses tuple types in its predefined operations, it is necessary to know the names of the records for representing tuples in Isabelle in advance, so that Isabelle specifications can be defined by Gencot for these operations. Gencot uses the include file `Tuples.cogent` which contains type synonyms `Tup<nr>` for the tuple types from 3 to 9 components. Whenever another Gencot include file uses tuple types with more than two components it includes this file, so that the tuple records in Isabelle are named accordingly.

Gencot Array Types

As described in Section 2.6.5 Gencot represents every C array type `<el>[<size>]` in Cogent with the help of type

```
type CArr<size> el = {arr<size>: el#[<size>]}
```

as `CArr<size> <el>`. For every `<size>` occurring in the program a separate generic type of this form is defined. Cogent translates this type definition to Isabelle as

```
record 'a CArr<size> = arr<size>_f ::  'a
type_synonym 'el CArr<size>_T = "'el list CArr<size>"
```

As in Cogent the array is wrapped in a single-field record, but it is represented by a list of arbitrary length.

Gencot takes a slightly different approach where the lists are restricted to the fixed length `<size>`. To achieve this, the last line of every array type definition as above is modified to

```
type_synonym 'el CArr<size>_T = "(<size>, 'el) FixedList CArr<size>"
```

where the type `FixedList` is defined for all array types by the Isabelle type definition

```
typedef (overloaded) ('n::len, 'a) FixedList =
  "{x::'a list. length x = LENGTH('n)}"
```

so that it includes the fixed length specified by parameter `'n`.

Type `MayNull`

2.12.2 Specifying Gencot Operations

Gencot Array Operations

The Gencot array operations as specified in Section 2.7.12 are generic in three aspects

- the array size
- the array element type
- the index type

The second and third aspect are handled by usual polymorphism in Isabelle. The first aspect could be handled in the same way due to the definition of `FixedList` which takes the size as a type parameter. However, every size uses an own wrapper record. Since the size is syntactically coded into the field name, all these wrapper records are different defined types and cannot be covered by polymorphism.

The solution taken by Gencot is to use a locale `CArrFuns` to handle the wrapper records in a common way. It is defined as

```
locale CArrFuns = fixes
  fld :: "'arr => ('n::len, 'el) FixedList" and
  wrp :: "('n, 'el) FixedList => 'arr"
  ...
```

where the locale parameters `fld` and `wrp` correspond to the field access function and the constructor function of the wrapper record. The type `'arr` is the type of the wrapper record which is used to actually represent the C array type in Isabelle.

For every size used in the program Gencot creates an interpretation of the locale:

```
interpretation CArrFuns<size>:
  CArrFuns arr<size>_f CArr<size>.make
```

using the actual field access function and constructor of the wrapper record as arguments.

As for all abstract functions Cogent generates for every Gencot array function such as `getArr` a constant declaration:

```
consts getArr :: "'arr x 'idx => 'el"
```

Gencot adds the definition for every such function in three steps. The first step is to define a corresponding function for type `FixedList`:

```
definition getArr' ::
  "('n::len, 'el) FixedList => 'idx::len word => 'el"
where ...
```

This function is generic in all three aspects: the array size `'n`, the index bitlength `'idx`, and the element type `'el`.

The second step is to define a corresponding function for the wrapper record in the locale `CArrFuns`:

```
definition getArrFxd :: "'arr x ('idx::len) word => 'el"
where "getArrFxd x == let (a,i) = x in getArr' (fld a) i"
```

It simply uses the locale parameter `fld` to lift `getArr'` to the wrapper record.

The third step uses ad hoc overloading to link this function to the constant generated by Cogent:

```
adhoc_overloading getArr CArrFuns<size>.getArrFxd
```

This is done for every `<size>` after the corresponding locale interpretation.

Note that the actual array size is not specified in the locale interpretation, it still remains generic. Only when an array function is invoked for an actual array value, the type of the array includes the array size and makes it available to the function definition where it can be used to check whether the index value is valid.

Operations for Type `MayNull`

2.12.3 Specifying Cogent Operations

Cogent defines several abstract function in its standard library. However, as of February 2021 the Cogent distribution does not contain Isabelle specifications for them. Therefore Gencot provides Isabelle specifications for all such functions which are used by Gencot.

Iteration Functions

The following Cogent iteration functions are used and supported by Gencot: `seq32`, `seq32_simple`. They correspond to counted for-loops in C, where the counter is a 32 bit word. With `seq32` the loop can be terminated early by a break statement. In all cases the loop body is passed as a function argument to the loop function, making the loop functions higher-order.

The iteration functions support arbitrary step amounts for the counter. If the step amount is greater than 1 even a for-loop with a specified end value for the counter may not terminate due to overflow of the counter. This happens if the last counter value `i` is lower than the end value and `i+step` is expected to be greater than the end value but overflows the 32 bit arithmetics and restarts

with a value again lower than the end value. This may lead to counting several times to the end value and then stop, or even run the loop forever.

This behavior is always expected to be a programming error and should not occur. To prove termination Gencot uses an additional guard `seq32_term to step` which excludes this behavior depending on end value `to` and step amount `step` by

$$\texttt{step} > 0 \wedge \texttt{(unat to)} + \texttt{(unat step)} \leq 2^32$$

It also excludes the case `step = 0` which would also lead to non-termination. The guard guarantees that as long as the loop has not terminated, so that the counter is still lower than `to`, adding the `step` will not cause an overflow. That a bit stronger than necessary but this way it becomes much simpler. Note that the guard is always satisfied if `step = 1`.

Every iteration function `seq` is specified in two parts. The core specification is a recursive function `seq_imp` which counts with natural numbers from zero in steps of one, so termination and induction are straightforward. The function `seq` is then specified by an axiom which takes the guard as a premise and calculates the number of iterations from start, step, and end values of the counter with the help of a function `seq32_cnt` and invokes `seq_imp` accordingly.

2.12.4 Reasoning Support for Gencot Operations

Gencot provides some rules for reasoning about Cogent functions which use Gencot operations. These rules can be used to support a higher level reasoning instead of unfolding the actual definitions given for the Gencot operations.

Gencot Array Operations

Gencot provides a simple abstract data type for arrays of arbitrary size. It consists of the four functions

```
siz :: 'arr => nat
vld :: 'arr => nat => bool
elm :: 'arr => nat => 'el
upd :: 'arr => nat => 'el => 'arr
```

where `'arr` is the array type (actually the type of the wrapper record) and `'el` is the array element type. Function `siz` returns the number of elements, `vld` tests whether an index value is valid for an array (i.e., it is lower that the number of elements), `elm` accesses an element by an index counting from zero, and `upd` updates the array at the specified index. These functions are overloaded for all array sizes used in the actual Cogent program.

The abstract data type is specified by the following rules which hide the underlying fixed list and wrapper record and thus simplify resoning about arrays:

```
sizArr:  siz a = <size according to type of a>
vldArr:  vld a n = n < <size according to type of a>
updArr:  vld a i ==> elm (upd a i e) i = e
updArrFrame:  vld a i∧vld a j∧i ≠ j ==> elm (upd a i e) j = elm a j
```

The first two are registered as simplifier rules so that `siz` and `vld` are automatically eliminated. The last two rules are also intended for simplification, however

they introduce additional goals for the premises and must be applied explicitly with the `subst` method.

Next, for every Gecot array operation `op` two rules `opValid` and `opInvalid` are provided which reduce the operation to the abstract data type functions. The first rule covers the case of valid index values, the second the case of invalid index values. Since they also have premises they must also be applied with the `subst` method.

Finally, the rule

```
eqArr:  (a1 = a2) =∀i.  (vld a1 i) ==> ((elm a1 i) = (elm a2 i))
```

defines extensional array equality if the elements are equal for all valid indices.

2.12.5 Reasoning Support for Cogent Operations

Gencot also provides some rules for reasoning about Cogent functions which use the Cogent abstract functions described in Section 2.12.3.

Iteration Functions

Gencot provides induction rules for the Cogent iteration functions `seq32` and `seq32_simple`. They have in common that they prove a predicate for the loop result by proving that it holds before the loop starts and is preserved by every single iteration. As third premise the rules take the termination condition `seq32_term` (see Section 2.12.3).

For function `seq32_simple` the body function takes as input and output a single accumulator value. Thus we can prove predicates which only depend on this accumulator value and not on the number of iterations performed. The induction rule `seq32_simple_induct_f` can be used to prove such predicates of the form

```
(P acc)
```

for accumulator values `acc`. Here the induction step goes from `(P acc)` to `(P (f acc))` where `f` is the body function.

The accumulator value property may additionally depend on the number of iterations performed. That is often the case when the loop iterates through an array, processing one element in every iteration. The induction rule `seq32_simple_induct` can be used to prove such predicates of the form

```
(P n acc)
```

where `n` is the natural number of iterations performed. Here the induction step goes from `(P n acc)` to `(P (Suc n) (f acc))`. It has as additional assumption that the iteration counter `n` is bounded by the number of iterations performed by the loop in the conclusion. This can be useful for proving the induction step, e.g. if the loop counter is used an an array index to prove that all accesses are valid.

Function `seq32` is more complex in two aspects. First, the body function takes as argument a record `{acc,idx,obsv}` with the current loop counter value `idx` and an "observed" value `obsv` which is not modified, in addition to the accumulator value. Second, it returns a pair `(acc,res)` which together with the new accumulator value `acc` contains a variant value `res` which is either

`Iterate ()` or `Break r` with some value `r`. In the second case the loop is terminated, as when using a `break` statement in C. The pair returned by the last iteration is also the result of the loop as a whole.

Since the loop counter is always passed explicitly to the body function, Gencot provides only the induction rule `seq32_induct` which can be used to prove predicates of the form

```
(P n (acc,res) obsv)
```

which may depend on the loop counter value, the pair `(acc,res)` returned by the body function, and the observed value `obsv`. Here the induction step must handle the two cases for `res`: If it is `Iterate ()` the step goes from `(P n (acc,res) obsv)` to `(P (Suc n) (f {acc,idx,obsv}) obsv)` where `idx` is the loop counter value calculated from `n`. If `res` is `Break r` the body function is not applied anymore, hence the step goes from `(P n (acc,res) obsv)` to `(P (Suc n) (acc,res) obsv)` where only the iteration counter is increased, so that the rule can conclude `P` for the number of iterations calculated by `seq32_cnt` without taking the early termination into account. As for `seq32_simple_induct` the step has the additional assumption that the iteration counter `n` is bounded by the number of iterations performed by the loop in the conclusion.

2.12.6 Refinement Proof for Gencot Operations

2.12.7 Refinement Proof for Cogent Operations

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 3.6.8.

3.2 Item Properties

As described in Section 2.6.1 Gencot supports declaring properties for typed items in the C program. The corresponding declarations are specified in textual form as a sequence of text lines where each line may declare several properties for a single item. The item is named by a unique identifier, every property is encoded by a short identifier.

3.2.1 Item Identifiers

Item identifiers are text strings which contain no whitespace.

Global individual items are functions or objects. They can always be identified by their name. However, for items with internal linkage the name is only unique per compilation unit, whereas the item declarations span the whole Cogent compilation unit which may consist of several C compilation units. Therefore, items with external linkage are identified by their plain name (which is an unstructured C identifier), items with internal linkage are identified in the form

```
<source file name> : <item 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:myitem"`. This approach differs from that for generating the Cogent

name, as described in Section 2.1.1, where the extension is always omitted. The reason for preserving the extension here is to better inform the developer in which file the item has been defined, not for preventing name collision.

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.

Local individual items are variables ("objects") locally defined in a function body. They are identified in the form

```
? <item name>
```

which is not unique. The current version of Gencot does not try to identify local variables in a unique way, since the language-c parser does not directly provide the name of the surrounding function for them. Even if that name would be used for disambiguation, this would not help for the case of different variables with the same name in the same function. Therefore in current Gencot version it is not feasible to declare item properties for individual local variables.

Collective items are identified by certain type expressions. For a typedef name the identifier has the form

```
typedef | <typedef name>
```

For a tag name it has one of the forms

```
struct | <tag name>
union  | <tag name>
```

For a tagless struct or union a pseudo tag name is used which has either of the forms

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

where `<lnr>` is the line number in the source file where the tagless struct or union begins syntactically and the suffix is constructed from the name `x.h` or `x.c` of the source file. In `x` all minus signs and dots are substituted by an underscore.

For a derived pointer type where the base type can be used as a collective item it has the form

```
<base item identifier> *
```

Sub-items of an item are identified by appending a sub-item specification to the item's identifier. The members of a struct or union are specified in the form

```
<item identifier> . <member name>
```

The elements of an array are collectively specified in the form

```
<item identifier> / []
```

The data referenced by a pointer is specified in the form

```
<item identifier> / *
```

The result of a function is specified in the form

```
<item identifier> / ()
```

The parameters of a function are specified in either of the forms

```
<item identifier> / <parameter name>
<item identifier> / <position number>
```

where the position number of the first parameter is 1. A virtual parameter with the Global-State property declared must always be specified in the former form with the parameter name. The position is automatically determined by Gencot when the parameter is added to the Cogent function.

As an example, if `f_t` is a typedef name for a function type with a parameter `par` which is a pointer to an array of function pointers, the results of the function pointers in the array can be identified by

```
typedef|f_t/par/*/[]/*/()
```

All these syntactic rules for item identifiers are defined in Haskell module `Gencot.Items.Identifier`.

3.2.2 Item Associated Types

Item properties affect how C types are translated to Cogent types. Thus during translation for every C type the corresponding item identifier must be known to retrieve the item properties for it. Module `Gencot.Items.Types` defines the type

```
type ItemAssocType = (String,Type)
```

for pairs consisting of an item identifier and a C type. Whenever a C type is processed in a declaration it is first paired with the identifier of the declared item to an `ItemAssocType` and then processed in this form.

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.

In all these cases the type is associated with an item and can be paired with the corresponding item identifier. There are other uses of types in C, such as in `sizeof` expressions, these cases are ignored by Gencot and must be handled manually.

The language-c parser constructs for every defined or declared function the derived function type, therefore 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 can reach them from their composite type. Together we can reach all item associated types used in a C program by determining all toplevel

items (variables/functions, composite types, typedef names), associate them with their types and then transitively determining all their sub-items paired with their types (field types, parameter and result types, base types of pointer and array types).

We also associate enum types with an item id, although they are not affected by properties. However, Gencot must determine and translate enum types as described in Section 2.6.3.

Generally, Gencot translates item associated types 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 with Const-Val property to an abstract access function, using the translated type of the variable or a pointer to the variable. Gencot translates local variables to Cogent variables with associated bindings, as described in Section 3.6.12. In its current version Gencot does not translate other local declarations, such as for types and functions. Also note, that in the current Gencot version local variables have no unique item identifier, thus no properties can be declared for them.

Union types are translated without referencing the field types (see Section 2.6.4). To support a manual translation Gencot treats them in the same way as struct types and also processes the types of their fields.

Gencot usually translates derived types to a parameterized Cogent type with the base type as type argument. Hence the base type is always required as type argument. Therefore Gencot also processes the sub-items corresponding to the base types of derived types.

For functions, the associated type is either a typedef name (resolving to a derived function type) or an expression for a 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) using the parameter and result types.

3.2.3 Property Identifiers

Each property is identified by a two-letter code as follows:

Read-Only: ro

Not-Null: nn

No-String: ns

Heap-Use: hu

Input-Output: io

Add-Result: ar

Global-State: gs

The numerical argument of Global-State properties is directly appended, such as in `gs5`. As an exception, the numerical argument 0 is omitted and the plain property identifier `gs` is used.

3.2.4 Item Property Declarations

An item property declaration consists of a single text line which starts with an item identifier, followed by an arbitrary number of property identifiers in arbitrary order, separated by space sequences. For better readability a colon may be appended to the item identifier.

As an example, to declare properties for the parameters `p1, par2, param3` of a function pointer `myfunp`, introduce an additional parameter for a global variable, and declare the Heap-Use property for the function itself, five declarations are needed:

```
myfunp/*/p1:     nn ro
myfunp/*/par2:   ar mf
myfunp/*/param3: ns
myfunp/*/gstate: gs3
myfunp/*:        hu
```

When item property declarations are merged (see Section 3.12.2) property specfication can be negative be prepending a minus sign - (without separating spaces):

```
mystruct.m1:     nn -ro
```

Upon merging a negative property suppresses the property for the item, independent where it is specified. Thus negative propertiy specifications have higher priority than normal ones.

3.2.5 Internal Representation

When used the textual form is converted to an internal Haskell representation of type

```
type ItemProperties = Map String [String]
```

defined in module `Gencot.Util.Properties`. It maps item property identifiers to the list of identifiers of declared properties (including the minus sign for negative properties. If several declarations for the same item are present in the textual form they are combined by uniting the declared properties. Repeated property specifications are removed, the order of the properties in the list is arbitrary. Negative properties and corresponding normal properties are not normalized, they can coexist in the same list.

The `ItemProperties` map is added as component to the user state of the `FTrav` monad. The monadic actions

```
getItems :: (String -> [String] -> Bool) -> FTrav [String]
getProperties :: String -> FTrav [String]
hasProperty :: String -> String -> FTrav Bool
```

are defined in `Gencot.Traversal` for accessing the item properties. The actions `getProperties` and `hasProperty` treat negative properties literally and do not interprete them.

3.3 Parameter Modification Descriptions

As described in Section 2.11 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.

3.3.1 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_heap" : <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 3.3.2. 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 function 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 after 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.

The attribute `f_heap` specifies whether the function uses the heap. Possible values are `"yes"`, `"no"`, `"depends"`, or `"?depends"`. The last value is generated by Gencot if heap usage cannot be decided locally. The value `"depends"` is never generated by Gencot, it must be set by the developer to confirm that heap usage must be determined transitively.

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 3.3.2) 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 changed 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_heap" : "no"
, "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>
, "heap_depends" : <string>
, "num_params" : <int> or <string>
  <argument descriptions>
}
```

The attribute `"name"` specifies the function identifier of the invoked function.

The attribute `"heap_depends"` specifies whether the heap usage may depend on the invocation of this function. Possible values are `yes`, `no`, and `?`. The last value is the default generated by Gencot, it must be confirmed to `yes` or `no` by the developer. This attribute is only used if the attribute `"f_heap"` for the invoking function has the value `depends`. Otherwise all `"heap_depends"` attributes at the invocations are ignored.

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. A parameter in the list is only effective, if its description is specified as `depends`. In all other cases the parameter is ignored when it appears in lists of invocation arguments.

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.

3.3.2 Identifiers for Functions and Parameters

In the JSON description unique identifiers are needed for all described functions, so that they can be referenced by invocations. Gencot uses the item identifiers as defined in Section 3.2.1 as function identifiers. An item identifier is a function identifier, if the item has a derived function type.

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.

3.3.3 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_heap" :"depends"
    , "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"
            , "heap_depends" :"no"
            , "num_params" :3
            , "1-__dest" :[ "3-input" ]
            , "2-__src" :"readonly"
            , "3-__n" :"nonlinear"
            }
        ,
            { "name" :"app:somefun/f_sort/*"
            , "heap_depends" :"yes"
            , "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_heap" : "no"
    , "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 `memcpy`. This invocation is required, since parameter `input` of `somefun` depends on its first argument which is neither nonlinear nor readonly. However, it is not required for deciding heap use, since that is specified as not depending on `memcpy`. If the description of parameter `input` is changed to `"yes"`, the description is closed, since now the invocation `memcpy` is not required anymore.

3.3.4 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`.

3.3.5 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.

In the same way, whenever the heap use depends on an invoked function, there is a function description present for the invoked function where the heap use is either specified directly or again depends on further invocations.

The result of evaluation is a simplified parameter modification description where the value `"depends"` does not occur anymore as parameter description or heap use specification. 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 for declaring the properties Read-Only and Add-Result for the function parameters.

3.3.6 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.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 3.6.3) to a sequence of function descriptions. It specifically handles types which syntactically include a function type (called "SIFT" type here). It translates all definitions and declarations of functions and of objects with a SIFT type. It also translates all type name definitions for a SIFT type. For every definition of a compound struct or union type it translates all members with a SIFT 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 -> ItemProperties
```

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 3.3.5.

The function `convertParmods` converts the JSON representation to an item property map (see Section 3.2.5) with declarations of the properties Read-Only, Add-Result, Heap-Use, and Input-Output. The parameter modification description must have been evaluated as described in Section 3.3.5. The resulting single parameter description values are translated to properties as follows: the description `"yes"` is translated to the Add-Result property, the descriptions `"readonly"` and `"no"` are translated to the Read-Only Property. The descriptions `"no"` and `"discarded"` and the `f_result` attribute are translated to a negative Add-Result property. For the remaining description `"nonlinear"` no property is declared. The Heap-Use property is declared, if `"f_heap"` has the value `"yes"`.

3.4 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.

3.4.1 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.

3.4.2 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

3.4.3 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.

3.4.4 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 would also work, since the mapped names are used for the association, which are unique in the <package> even for declarations with internal linkage.

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

Filter `gencot-deccomments`

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

```
#DECL <name> <bline>
```

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

The filter implementation is described in Section 3.11.8.

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.

3.5 Preprocessor Directives

3.5.1 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.

3.5.2 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.

3.5.3 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 3.6.1. 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.

Both filters must map original C identifiers to their Cogent form. This requires the use of the name prefix map file, as described in Section 3.6.7. Therefore, both filters take the name of this file as an additional argument.

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).

3.5.4 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.

3.5.5 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.

3.6 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 3.6.9).

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
```

3.6.1 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.

3.6.2 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.

3.6.3 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 3.6.14.

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

The first one takes as arguments the original C source file name, 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 instead of the original C source file name and does the same reading from the file. Both functions pass the file name as first argument to the callback handler.

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

The callback handler is used by Gencot whenever it has to process information in a different order than in the source file. Then the callback handler collects all relevant information during the analysis phase and returns it in the user state. It is used for the second AST traversal for the processing by Gencot (see Section 3.6.14) by either transferring it to the user state for that traversal, or by using it as the traversed AST elements. There are two callback handlers used by Gencot this way:

- `collectGlobalStateIds`, defined in module `Gencot.Items.Types`. It collects a mapping from item ids to item definitions or declarations for all global variables, which is added to the user state of the second traversal. It is needed because a function with a Global-State parameter may be defined before the corresponding global variable. The variables type is required for processing the parameter, it must be retrieved from the definition or declaration.

- `collectTypeCarriers`, defined in module `Gencot.Util.Types`. It collects typed AST elements occurring in the Cogent compilation unit as starting points for transitively determining all used types in the second traversal.

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 3.11.1. 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 3.6.14).

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 functions

```
getOwnDeclEvents :: GlobalDecls -> (DeclEvent -> Bool) -> [DeclEvent]
getDeclEvents :: GlobalDecls -> (DeclEvent -> Bool) -> [DeclEvent]
```

The first one reduces the declarations to those directly in the input file, the second also returns declarations from included files. Both perform the preprocessing and return the list of `DeclEvent`s to be processed. As its second argument they expect a filter predicate for restricting the declarations to the `DeclEvent`s to be processed by the specific Gencot component.

3.6.4 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.

If a processor only processes items which are external to all input files, the original file name is not required for the files input to a Gencot processor. However, the processor `gencot-dvdtypes` also processes derived types which are not external. In a derived type the original file name is only used for constructing the Cogent name and item identifier for tagless compound types. Currently the only case where tagless types are processed by `gencot-dvdtypes` is when they occur in a derived function type. If it occurs as parameter type, it is mostly useless, because then the type has only prototype scope and is not available when the function is invoked. If, however, it is used as result type, a typedef name can be defined for it in another declarator of the same declaration and used anywhere in the program. Since this is a very unusual construction in a C program Gencot does not handle it. This will possibly result in wrong type names in the result of `gencot-dvdtypes` which must be manually corrected.

It would be possible to provide a list of the original file names together with the list of files to be processed and use them as described in Section 3.6.14. However, then it is only available in the userstate during analysis for a single input file. After merging the semantic maps the content of the resulting map cannot be associated with the original file name any more. It would be necessary to preserve this association as a separate data structure.

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 3.6.3, 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 3.6.14, 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 may 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 definition for it occurs in at most one of the symbol tables.

If both symbol tables contain a definition for the same identifier with external linkage Gencot compares their source file positions. If both positions are the same, the definition is in a file included by both sources and it is correct to discard one of them. If the positions are not the same, Gencot signals an error. The typical case for such included definitions are object declarations where the `extern` specifier is omitted. This is interpreted as a "tentative" definition in C and is classified as definition by the language-c analysis step, if it is not followed by a definition.

The dummy declarations generated by Gencot for parameterless macro definitions (see Section 3.6.5) have the form of such tentative definitions. Since they are generated separately for every C source file, they cannot be recognized by comparing their source file positions. To prevent them from being signaled as duplicate definitions they are explicitly specified as `extern`, then they are classified as declarations by language-c (instead as tentative definitions) and no error is signaled by Gencot. Alternatively they could be generated with internal linkage (specified as `static`), then they are removed before combining the symbol tables. However, then their names are mapped differently, as described in Section 2.1.1. Note that (manually created) dummy declarations for macros with parameters are never classified as definitions, since they have the form of a C function declaration without a body.

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 3.6.3 for a table built from a single input file.

3.6.5 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

```
extern 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.

Dummy declarations are generated with explicit external linkage. This causes them to be classified by language-c as declarations instead as (tentative) definitions, hence they are not signaled as repeated definitions when they occur in several symbol tables (see Section 3.6.4).

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.

3.6.6 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.

Translations for C Statements and Expressions

C statements and expressions are first translated to intermediate Cogent bindings, which are then further processed and finally converted to Cogent expressions. Gencot directly uses the datastructure for representing bindings in the Cogent AST for the intermediate form. Although it would be possible to define a specific data structure which is more adequate for the intermediate processing, this approach has the advantage of higher flexibility, the possibility of using the Cogent prettyprinter for debugging output and no need to transform between representations.

However, only a subset of the AST for bindings is used. Of the datatype `Binding` only the variant `Binding` is used, the variant `BindingAlts` is not used since it is intended for refutable patterns to distinguish cases. The optional type in the `Binding` variant is not used.

The irrefutable patterns in the bindings are usually nested tuples of variables, built from the constructors `PTuple` and `PVar`. Related to struct and array accesses also `take` patterns are used constructed by `PTake` and `PArrayTake`.

Extending the Cogent Surface Syntax

Gencot needs to extend the Cogent surface syntax for its generated code in three ways:

- origin markers must be supported, as described in Section 3.1,
- additional C code must be supported in Cogent dummy expressions, as described in Section 2.10.
- type information must be supported for some of the postprocessing steps, described in Section 3.7.

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 3.6.3. 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 Gencot embeds C code in the Cogent AST when it cannot translate it. This may happen for C statements and expressions, as described in Section 2.10. The C code is output in the form of a comment to inform the programmer for manual action.

In this case the corresponding Cogent code is a dummy expression which has the form of an invocation of the function `gencotDummy` (see Section 2.7.1). So it would be sufficient to support embedded C code only for such expressions. However, the Cogent AST implementation supports extensions only for all expressions, types, irrefutable patterns or toplevel constructs. Therefore Gencot extends all expressions to support embedded C code.

The embedded code origins from the C AST resulting from parsing the C source. It could either be embedded in the Cogent AST as a text string or as an AST structure. Embedding it as text is more flexible, however, it involves executing the C prettyprinter during translation, where the prettyprinting context (such as line length) is not available. Therefore it is embedded as an AST structure and prettyprinted together with the Cogent code. The embedded code may either be a C statement or a C expression. Since every C expression can be wrapped as a simple statement, Gencot always embeds C code in the form of a statement AST.

Type Information The translation of function bodies to the Cogent AST described in Section 3.6.12 uses nearly no type information from the C program and only interpretes some of the item properties. As described in Section 3.7, the rest of the translation is done in postprocessing steps which work exclusively on the Cogent AST and have no access to the parsed C program any more. However, some of the postprocessing steps need information about the expression types and some item properties, in particular the readonly processing described in Section **??**. Therefore this information must be added to the extended Cogent AST.

It would be possible to add the item associated types (see Section 3.2.2) to the Cogent AST. However, some of the type properties, such as determining whether a type is linear or readonly, requires to expand typedef names using the symbol table, and it requires to access the item properties. The symbol table and item properties are only available for monadic actions during C AST traversal (see Section 3.6.14). Thus, it would be necessary to implement all postprocessing by monadic actions. This could be avoided by pre-calculating

all required type properties and adding that information to the Cogent AST together with the C types. However, that would be inflexible and result in a rather complex extended Cogent AST.

Therefore Gencot uses another approach by adding the Cogent type, determined according to Section 2.6, to the Cogent AST. The mapping from C types to Cogent types respects all relevant item properties, so that their effects are present in the Cogent types.

However, typedef names are usually mapped to Cogent type names which hides the structure of the type associated with the typedef name. To make this information available, a global table of all generated Cogent type definitions would be needed to resolve used type names, which makes it necessary to implement the postprocessing steps as monadic actions using the type definition table as state. Alternatively, the Cogent types could be constructed by resolving all type names, so that their full structure is directly available and no global state is necessary for the postprocessing steps.

Using the second alternative would require to translate C types twice: once with type names resolved, for using the full type information during postprocessing, and once with type names preserved, for using a type as part of the generated Cogent code, where it should be as compact and similar to the C code as possible. There are even cases where types in the generated Cogent code must be constructed from the additional type information of expressions (explicit type arguments specified for polymorphic functions like `repeat`, see Section 3.6.12), therefore both forms must be added to the Cogent AST.

To reduce this complexity Gencot implements types in the Cogent AST in a way that combines both forms. Basically, type names are always fully resolved so that the type structure is available in a standalone way. Moreover, if the type or a part of it has been determined by resolvong a type name, this type name is additionally specified in the type or its part, so that it is available for the prettyprinter. When such a type is output to the Cogent code, the prettyprinter uses type names instead of outputting the full type structure whenever available.

This form of types is directly used to implement the types as part of the Cogent AST and it is added to the following other nodes in the Cogent AST: `Pattern`, `IrrefutablePattern`, and `Expr`. It cannot be added to `Binding` and `Alt`, because these nodes cannot be extended. The remaining node `TopLevel` represents toplevel definitions of types, functions, and constants, where enough type information is already available in the normal Cogent AST structure.

Extended Cogent Surface Syntax AST Although the Cogent code generated by Gencot is only a restricted subset of the full Cogent syntax, Gencot extends all parts of the Cogent AST which can be extended (i.e. are parameterized in the type definitions). This is more systematic and allows arbitrary extending the subset of the generated Cogent syntax.

Together, Gencot uses types equivalent to the following definitions to represent its extended Cogent surface AST:

```
data GenToplv =
  GenToplv (TopLevel GenType GenIrrefPatn GenExpr) Origin
data GenExpr =
  GenExpr (Expr GenType GenPatn GenIrrefPatn () GenExpr)
          Origin GenType (Maybe Stm)
```

```
data GenPatn =
  GenPatn (Pattern GenIrrefPatn)
          Origin GenType
data GenIrrefPatn =
  GenIrrefPatn (IrrefutablePattern VarName GenIrrefPatn GenExpr)
          Origin GenType
data GenType =
  GenType (Type GenExpr () GenType)
          Origin (Maybe String)
type GenBnd = Binding GenType GenPatn GenIrrefPatn GenExpr
type GenAlt = Alt GenPatn GenExpr
```

The second parameter of `Type` and the fourth parameter of `Expr` have been added in 2020 for representing Dargent layout information. Gencot does not generate that and uses the unit type instead.

The third components in `GenExpr`, `GenPatn`, and `GenIrrefPatn` are the additional Cogent type information mainly intended for postprocessing, as described above. The optional `String` in `GenType` is the alternative representation by a type name to be used by the prettyprinter whenever available.

Type `Stm` is the type for C statements as defined by the language-c-quote AST (see Section 3.6.9), used to optionally embed C code in the Cogent AST.

The types `GenBnd` and `GenAlt` are defined as abbreviations, the types of the corresponding nodes cannot be extended, because they are not parameterized for `Expr` nodes in the Cogent AST. Type `GenBnd` is used as intermediate representation for translating C statements and expressions (see Section 3.6.12).

All five wrapper types are defined as instances of class `Pretty`, basically by applying the Cogent prettyprint functionality to the wrapped Cogent AST type.

3.6.7 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.

Name Mapping Configuration

As described in Section 2.1.1 the general mapping schema is based on the replacement of name prefixes and can be configured. The configuration is specified in a "name prefix map" file. It consists of a sequence of line, each line describes the mapping of a prefix in the form

```
<prefix> <upper>|<lower>
```

where `<prefix>` is the prefix to be replaced, `<upper>` is the replacement if the resulting name must begin with an uppercase letter, and `<lower>` is the replacement if the resulting name must begin with a lowercase letter. The prefix is separated by whitespace from the replacements, any remaining content in the line is ignored. If the replacement specification does not contain a `"|"` it is used for both cases, converting the first letter to upper- or lowercase, as needed. If `<prefix>` is omitted the empty prefix is replaced, i.e., the replacement is prepended to the name.

If several lines are specified the prefixes are matched in the order of the lines and the first matching prefix is applied. Note that the empty prefix always matches, hence lines following it are ignored. Empty lines are always ignored, they do not prevent the use of subsequent lines. At the end Gencot automatically appends a line of the form

```
cogent_
```

which results in the default mapping described in Section 2.1.1 if no other line matches.

The file content is read and converted to an internal representation of the form

```
type NamePrefixMap = [(String, (String, String))]
```

which maps from the original prefix to the pair of replacements for uppercase and lowercase names. This representation is added to the monadic user state so that it is available during the monadic AST traversal (see Section 3.6.14). A list is used instead a map so that the order of the map entries is preserved.

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 monadic action

```
mapName :: Bool -> Ident -> f String
```

where `f` is a `MapNamesTrav` and `MonadTrav` (see Section 3.6.14). The first argument specifies whether the name must be uppercase. It uses the configuration from the user state to match and replace a prefix.

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, or a C struct/union/enum type reference.

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. To access the real name of the input file it must be retrieved from the user state (see Section 3.6.14). Hence, the mapping function is defined as a monadic action.

Together the function for translating struct/union/enum type references is

```
transTagName :: TypeName -> f String
```

where `f` is the same as for `mapName`.

If the definition itself is translated, it is already available and need not be retrieved from the map. However, the user state is still needed to map the generic name `<stdin>` to the true source file name. Therefore Gencot uses function `transTagName` also when translating the definition.

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 together with the real name of the input file is sufficient for translating the name. To make both 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 3.6.9.

```
transObjName :: Ident -> f String
```

where `f` is the same as for `mapName`.

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 -> f String
```

where `f` is the same as for `mapName`.

Replacing Names of Global Variables

As described in Section 2.9.1, accesses to global variables in a function body may be replaced by an access to a parameter introduced by a Global-State parameter or by an invocation of an access function, in case of a Const-Val property. To retrieve the properties of the global variable, its item identifier must be constructed. To do this, Gencot must determine whether the identifier of an object used in a function body has local or global scope, since the item identifiers for the same C identifier are different in both cases (see Section 3.2.1).

For variables the scope can be determined from the identifier's linkage. If it has global scope it must have either internal or external linkage. If it has local scope it must have no linkage. A locally declared variable with keyword `extern` has external linkage and only references a definition outside the function body, thus it has global scope. A locally declared variable with keyword `static` has no linkage, the keyword only specifies its storage class. Thus it has local scope and is never replaced by Gencot.

For every name occurring in a C function body, which syntactically may be a global variable, Gencot looks up its definition in the symbol table. If it has external or internal linkage Gencot constructs its item identifier and retrieves its declared properties. If a Global-State property has been declared, Gencot accesses the function definition in the monadic user state (see Section 3.6.14) and determines its virtual parameter items with Global-State properties. If one of them has the same Global-State property as the global variable its name is mapped, the C dereference operator `*` is applied to it, and the resulting expression is used instead of the variable name.

Otherwise, if the Const-Val property has been declared for the variable, it is replaced by an invocation of the parameterless access function. The function name is determined by mapping the variable name. Otherwise the variable name is not replaced and only mapped to the Cogent naming scheme.

3.6.8 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.

3.6.9 Generating C Expressions

For outputting the Cogent AST the prettyprint functionality must be extended to output the embedded C code. In that C code origin markers must be generated to be able to re-insert comments and preprocessor directives. Finally,

all names occurring free in the embedded C code 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 embedded C code 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`.

As described in Section 3.6.6, the extended Cogent AST uses only statements from this C AST for embedding C code. However, the wrapper functions are also generated using the C AST and they need complete C function definitions and embedded antiquoted Cogent types. Therefore the complete language-c-quote AST is reimplemented in `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 embedded C code. The function for translating from language-c AST to the Cogent AST is extended by the functions in `Gencot.C.Translate` to translate code 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 embedded C code 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 3.6.14). 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 2.6.2.

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 2.6.8: 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 2.6.3 and 2.6.4: 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.

3.6.10 Translating C Expressions

The translation produces Cogent code in a simple and straightforward manner. The resulting code is highly inefficient and large, it is intended for being improved in postprocessing steps.

General Translation of C Expressions

In C an expression can not only have side effects, side effects can even be sequenced, i.e., the result of a side effect can be observed and used for another side effect, for example by using the comma operator. To cover sequenced side effects, Gencot uses binding lists as intermediate representation for expressions. The meaning is that the rest of a list is evaluated in the context established by the first binding.

Related to accessing components of struct and array values Gencot uses `take` and `put` operations in Cogent. Components accessed by a `take` operation must usually be put back after use. To make the component value accessible as result value of an expression it is bound to a variable in between. Therefore Gencot represents a translated expression by a binding list together with a variable for accessing the result value.

In C an expression may also denote an "lvalue" which can be modified. Gencot represents modifications by binding a new value to specific variables. Therefore the representation of a translated expression also includes an optional "lvalue" variable for this case.

The type `ExprBinds` is used as intermediate representation for a translated C expression. Gencot uses a common monadic action

```
bindExpr :: CExpr -> FTrav ExprBinds
```

to translate every expression to the intermediate representation, irrespective of the expression kind.

As variable for the expression value Gencot uses variables named `v<n>'`, where `v0'` is abbreviated to `v'`. Due to the prime character this is a valid identifier in Cogent but not in C, so it cannot collide with any other variable taken and mapped from the C code. The `<n>` is a natural number increased for each (sub)expression so that the variables for all expression values are distinct. Note that, since expression values can only be used once at the position where the expression is syntactically placed, every value variable will also be used only once (in translations of assignments and increment/decrement operators it is used twice, but immediately together in a tuple expression) and could be reused afterwards for other expressions. However, the value of an expression is not

always used before another expression is evaluated, therefore the value variables are constructed so that they are unique.

The binding which binds the expression result to a variable has the general form

```
(v<n>',v1..) = expr
```

with value variable `v<n>'`. The term `v1..` abbreviates the sequence `v1`,...,`vn` of the mapped names of all C identifiers affected by side effects of the expression, their values are specified in the corresponding components of the tuple value of `expr`. Often, `expr` will be a tuple expression of the same size as the pattern, but in some cases not (e.g., for function calls or dummy expressions). The variables `v1`, ..., `vn` are called "side effect targets" and occur in the pattern in arbitrary order. If there are no side effect targets, the lhs pattern is the single variable `v<n>'` and the rhs is the expression for the value bound to it.

If Gencot cannot translate an expression it translates it to a single binding where the rhs is a dummy expression as described in Section 2.10.4. The lhs may still contain side effect targets, if Gencot detects that the expression modifies these C objects.

The binding list can be combined to a single binding by the function

```
cmbBinds :: ExprBinds -> GenBnd
```

If the value variable of the `ExprBinds` is `v<n>'` the resulting binding has the form

```
(v<n>',v1..) =
let b1 and ... and bk
in (v<n>',v1..)
```

where `b1`, ... , `bk` are the bindings in the list and the `v1..` are all side effect targets which occur in the list. Binding lists are usually only combined when their code is evaluated conditionally.

Dealing with Subexpressions

An expression with subexpressions is translated to a binding list which first binds the subexpressions to value variables and then uses these variables in the final binding as replacement for the subexpressions. The subexpressions are translated to binding lists using `bindExpr`.

The C standard defines "sequence points" for subexpression groups in some expressions. This means that side effects occurring in subexpressions before the sequence point are observable in subexpressions after the sequence point. Otherwise subexpressions are "unsequenced", then the semantics is undefined if their side effects overlap (either the target of one can be observed in another subexpression or side effects in two subexpressions have common targets).

Gencot does not translate expressions where side effects of unsequenced subexpressions overlap for an object determined by the same identifier in C. Otherwise it translates every group of unsequenced subexpressions to a binding list and concatenates these lists according to the sequencing of the groups.

The binding lists of unsequenced subexpressions are concatenated in arbitrary order. Because the subexpressions have no overlapping side effect targets,

the side effect targets are all distinct. As described above, the value variables are constructed so that they are distinct for all subexpressions. Together, no variable will be rebound by concatenating the lists.

Note that it is not sufficient to use value variables numbered from 1 to `n` for `n` subexpressions. The same variables could be used for subexpressions of the subexpressions and would be rebound when the lists are concatenated.

Note also that there can be overlapping side effects for struct components, array elements, and objects accessed through shared pointers. These overlapping side effects are not determined by Gencot, the resulting translation corresponds to one possible semantics of the C program resulting from the specific sequencing used for the unsequenced subexpressions. If unwanted, such situations must be detected and corrected manually.

Expressions for Values with Pointers

In C even an expression which is not an "lvalue expression" can cause side effects, if the expression evaluates to a value which is a pointer or contains a pointer and occurs in a context where it is possible to use the pointer to access and modify the referenced data. In the translation the possible side effect must be respected by binding a variable to the value with the pointer referencing the modified data. In general it is not clear which variable to use here.

In C a value containing a pointer is either the pointer itself, or it is a container with a component containing a pointer, i.e., a struct with a member containing a pointer, or an array where the elements contain a pointer. In C an expression for a pointer or container is either an identifier, a component accessed in another container, an application of the derefence operator `*`, a function invocation which returns the pointer or container as its result, a conditional expression, or a comma expression. A container can also be designated by a compound literal. A pointer can also be designated by an application of the `&` operator, or by "pointer arithmetics", i.e., by arithmetic operations involving pointers.

If a pointer or container is used in a context, the subexpression for the pointer or container is translated by Gencot as described above, resulting in a binding list which binds the pointer or container to a value variable `v<n>'`. The use in the context results in another binding which uses `v<n>'`. This binding must be extended by a side effect target for the pointer or container after possible modification.

If the pointer or container is specified in C by an identifier, the identifier is mapped to a Cogent variable which is bound to `v<n>'`. The mapped identifier is then used as side effect target. If there is no sharing, all subsequent uses of the pointer or container must be through the same identifier, therefore the modifications will be respected by the re-binding.

If the pointer or container is specified in C by access to a component of another container, the value variable `v<n>'` is bound to the corresponding component variable, as described in the following subsection about accessing struct members and array elements. This component variable is used as side effect target to bind the pointer or container value after modification. Since the component variable is also used in the putback operation, the surrounding container will be affected by the modification. As described below, if the pointer or container is specified by dereferencing another pointer, Gencot treats this in the same way as for a component access, using a component variable.

If the pointer or container is specified in C by any other expression, it can only be accessed afterwards if it is shared by another C object. Usually, Gencot cannot automatically translate such cases to valid Cogent code. Therefore Gencot does not bind the pointer or container after modification to a variable, it uses the error variable `err'` instead. Since the error variable is never referenced this means the pointer or container is "discarded" in Cogent. Depending on the further processing in following translation stages this may cause a corresponding error message by the Cogent compiler.

Accessing Struct Members and Array Elements

The most general way of accessing struct members and array elements in Cogent is the `take` operation. It separates the member or element from its container data structure and binds both to variables. Since the `take` operation syntactically is a pattern, the access expressions can be represented by a binding list, like all other expressions.

In general the `take` operation modifies the type of the container value. To restore it the component must be re-inserted after being used or replaced which is done by a `put` operation. A `put` operation is a Cogent expression which returns the reassembled container. Gencot binds it to the same variable which has been used to bind the remaining container in the `take` operation. The resulting binding always consists of the single variable bound to the `put` expression.

The variable to which the component is bound by the `take` operation is used by the `put` operation for the reassembling. Therefore it cannot be a value variable since it may be used more than once. Gencot uses variable names of the form `p<n>'` for this purpose, called component variables. They are distinct from all value variables and, due to the prime character, from all mapped C identifiers. The `<n>` is a natural number increased for each `take` operation so that the variables for all taken components are distinct.

The variable to which the remaining container is bound in the `take` operation is determined as described for values with pointers. Cases where there are no pointers in the container or where no modifications can occur are handled in later translation stages. If the variable is determined to be the error variable `err'` because the container has not been specified in C by a variable or a chain of nested accesses to a variable the `put` operation is replaced by a unit binding `() = ()` to prevent a free occurrence of the error variable in the `put` expression.

If the container is a struct which does not contain (directly or indirectly) a pointer it may be discarded in Cogent after accessing a component. Therefore a component access where the container is specified by the result of a function invocation, a conditional expression, a comma expression, or a compound literal can be represented in Cogent as long as the component is not modified, so that no `put` operation is required. However, in the first three cases the container is no lvalue expression in C and cannot be modified. A compound literal is an lvalue expression and can be modified, but compound literals are not (yet) supported by Gencot. Gencot translates these access expressions in the same way as all others, using a `take` operation where the remaining container is specified by the error variable and the `put` operation is replaced by a unit binding. The `take` operation will be replaced by a valid member access in later processing steps.

If the container is an array, the `put` operation needs to use the same element index as the `take` operation. In the `take` operation it is available as a value

variable. Since value variables are not reused in the body of a function, the same value variable is used in the `put` operation.

Between the `take` and `put` bindings the component variable may be rebound. This way it is possible to translate side effects modifying the component and thus its container. In such a rebinding the component variable occurs as a normal side effect target.

If it has been rebound the component must be put back before it is accessed again in the same expression and before the container has been modified in other ways. Therefore Gencot always puts it back as soon as possible, i.e., as soon as it has been bound to a value variable or has been rebound. However, a component may also be used by accessing a nested component using a nested take and put. Nested component access can be arbitrarily deep, therefore it may be translated to an arbitrarily long sequence of take bindings, followed by a component binding or rebinding, followed by a list of put bindings in the opposite order of the take bindings, so that components which are taken first are put back last.

A struct member access `e.m` has only one subexpression `e` for denoting the struct. If it is a variable or an access chain starting with a variable a read access is translated to the following binding sequence:

```
<v>{m=p<k>'} = v<m>'
v<n>' = p<k>'
<v> = <v>{m=p<k>'}
```

where `v<m>'` is the variable to which the value of `e` has been bound by the translation, `<v>` is a variable depending on the expression bound to `v<m>'` in translation of `e` as described above, and `p<k>'` is a new component variable. The member value is available as `v<n>'`.

If the struct component is used as an lvalue the component variable `p<k>'` is re-bound to the new value by additional bindings inserted between the second and third binding.

A Cogent `take` operation can be applied to boxed and to unboxed record types. In the member access `e.m` the expression `e` corresponds to an unboxed record, in the member access `e->m` it corresponds to a boxed record. Therefore the member access `e->m` is translated by Gencot in the same way as `e.m`, the difference can be detected by looking at the type of `v<m>'`.

An array element access `e1[e2]` has two unsequenced subexpressions `e1`, `e2` for denoting the array and the index value. The array access is translated by first concatenating the binding list for `e1` after `e2`. Then, if `e1` is a variable or an access chain starting with a variable a read access is translated to the following binding sequence:

```
<v> @{@v<l>'=p<k>'} = v<m>'
v<n>' = p<k>'
<v> = <v> @{@v<l>'=p<k>'}
```

where `v<m>'` is the value variable bound in the translation of `e1`, `v<l>'` is the value variable bound in the translation of `e2`, and `<v>` and `p<k>'` are as above. The element value is available as `v<n>'`.

If the array element is used as an lvalue the component variable `p<k>'` is re-bound to the new value by additional bindings inserted between the third and fourth binding.

Here the specific Cogent array `take` and `put` operations are used. In Cogent they can only be applied if `v<m>'` has a builtin array type. This is never the case for the code generated by Gencot, the mapped array types use builtin array types, but they are wrapped in record types (see Section 2.6.5). Moreover, in C an array element access can be applied to any pointer which is not a function pointer, therefore the type of `v<m>'` can result from mapping any C array or pointer type to Cogent, as described in Sections 2.6.5 and 2.6.7. The array `take` and `put` operations are only used as intermediate representations here, they will be completely replaced by other code constructs in the postprocessing steps, depending on the type of `v<m>'`.

Translating Pointer Dereferences

A pointer dereference is denoted in C by an application of the unary operator `*` to a subexpression `e` which evaluates to a pointer value. Depending on the type of the referenced value (container, function, other) and whether they can be `NULL`, pointers are treated differently by Gencot (see Section 2.6.7).

In the first translation step Gencot only differentiates between function pointers and all other pointers. Function pointers can only be dereferenced for invoking the function, this is always implemented by Gencot using the abstract function `fromFunPtr` (see Section 2.7.9). The following binding is appended to the translation of `e`:

```
v<n>' = fromFunPtr[ft,fpt] v<m>'
```

where `v<m>'` is the value variable bound in the translation of `e`. The types `ft` and `fpt` are the types of the function and the function pointer, they must be explicitly specified because `fromFunPtr` is polymorphic.

For all other pointers Gencot translates pointer dereferences as if the pointer type would be translated to a type `(CPtr T)` by accessing the `cont` component. That is, an expression `*e` is translated by `bindExpr` like `e.cont` to a binding list involving a `take` and a `put` operation as described for struct member accesses. If `e` is a variable or an access chain starting with a variable a read access is translated to the following binding sequence:

```
<v>{cont=p<k>'} = v<m>'
v<n>' = p<k>'
<v> = <v>{cont=p<k>'}
```

where `v<m>'` is the value variable bound by the translation of `e`, `<v>` is a variable depending on the expression bound to `v<n>'` in the translation of `e` as described above, and `p<k>'` is a new component variable. The referenced value is available as `v<n>'`.

Semantically this means that the referenced content is separated from the pointer in a `take` binding and is bound to `p<k>'` and then both are recombined by the `put` binding.

If the dereferenced pointer is used as an lvalue the component variable `p<k>'` is re-bound to the new value by additional bindings inserted between the second and third binding.

Note that a member access `e->m` is equivalent to `(*e).m` but the latter is translated in two steps using two nested `take` and `put` pairs. This form will be simplified by postprocessing steps.

Note also that, similar as for array element accesses, there are cases where the `take` and a `put` operations cannot actually be applied, because the mapped type of `e` is no record with a `cont` component. It may even be a mapped array type, because in C pointer dereference can be applied to arrays, which accesses the first array element. In these cases the `take` and a `put` are only intermediate representations which will be replaced by postprocessing steps.

Acessing Global Objects

An identifier referenced in a function body may denote a global object. This can only be translated in three possible cases: if the global object has a Const-Val or a Global-State property (see Section 2.6.1) or if it is a preprocessor constant (see Section 2.3). In the following, let `<id>` be the mapped name of the identifier referencing the global object.

If the global object has a Const-Val property, it is translated by Gencot as a parameterless access function which returns the value of the global object. In this case the object reference is translated to an invocation of the access function. The name of the access function is the mapped name of the C object. The resulting binding is

```
v<n>' = <id> ()
```

If the global object has a Global-State property it can be accessed if the surrounding function has a parameter with the same property. If no such parameter exists a dummy expression with an error message is generated. Since the parameter is used to pass a pointer to the global object an additional pointer dereference is generated by Gencot. A read access is translated to the following binding sequence:

```
gsp{cont=p<k>'} = v<m>'
v<n>' = p<k>'
gsp = gsp{cont=p<k>'}
```

where `gsp` is the name of the parameter with the Global-State property and `v<m>'` is the value variable bound to it and `p<k>'` is a new component variable. The value of the global object is available as `v<n>'`.

If the global object is used as an lvalue the component variable `p<k>'` is re-bound to the new value by additional bindings inserted between the second and third binding.

The global object may also be a preprocessor constant, because for them dummy declarations have been created by Gencot, as described in Section 3.6.5. These dummy declarations cause them to appear like global C objects. Since preprocessor constants are translated to Cogent value definitions they can be directly accessed in the Cogent code. Therefore they are translated to a reference to a Cogent variable of the mapped name of the preprocessor constant. The resulting binding is

```
v<n>' = <id>
```

Note, that such a Cogent variable is never bound in the context of this code, it always references the toplevel Cogent value definition.

The remaining case is a true global C object with neither of the two properties. This case cannot be translated to Cogent. Since it is not possible to

distinguish this case from the preprocessor constant, Gencot always assumes that global objects which have neither property are preprocessor constants and translates them accordingly. If that assumption is wrong, no Cogent definition or binding will exist for the generated variable reference and the Cogent compiler will signal an error. It us up to the user to add the properties to all global C objects which are accessed in translated function bodies.

Translating the Address Operator

The address operator `&` is used in C to determine a pointer to an object. If applied to a function, Gencot translates it using the abstract function `toFunPtr` (see Section 2.7.9).

Gencot does not translate other expressions which apply the address operator to a subexpression, this must be done manually (see Section 4.7.4).

Translating Function Calls

A function call `f(e1,..,em)` has the function `f` to be applied and all actual arguments as subexpressions, these are treated as usual. Then the application is translated to the following binding sequence:

```
v<n>' = v<n0>' (v<n1>',..,v<nm>')
```

where `v<n0>'` is the value variable bound in the translation of the function subexpression `f`. If the function has result type `void` in C the result has unit type in Cogent, then the unit value will become bound to `v<n>'`.

Depending on its type and on item properties for the function and its arguments, the function may take additional arguments and it may return a tuple with additional results. Also, the function may be variadic or incomplete. The following item properties must be directly processed here, since they cause the introduction of additional function arguments or result components: Heap-Use, Input-Output, Add-Result, and Global-State. The other item properties only affect the translation of types and are thus implicitly included in the translated types added to the expressions and patterns, so they can be handled in later translation stages.

The item properties Heap-Use, Input-Output, and Global-State introduce additional function arguments, for them actual values must be determined to pass to the function invocation. In all three cases the additional arguments must also be present for the surrounding function definition for which the body contains the invocation. The corresponding variable names are used as actual arguments for the invocation. In the case of Heap-Use and Input-Output it is a fixed name as described in Section 2.6.1, in the case of Global-State it is the name of the corresponding virtual parameter.

The item properties Add-Result, Heap-Use, Input-Output, and Global-State introduce additional result components. In that case the binding has the form

```
(v<n>',v1..) = v<n0>' (v<n1>',..,v<nm>')
```

with a tuple pattern. The variables `v1..` correspond to the properties Add-Result, Global-State, Heap-Use, Input-Output in that order. The variable for a component added by an Add-Result property is determined as described for expressions for values with pointers, i.e., it is either a mapped C variable name,

or a component variable, or the error variable `err'`. The variable for a component added by a Global-State, Heap-Use, or Input-Output property is the same which is used as actual argument for the corresponding parameter, i.e., it is the name of the corresponding parameter of the surrounding function.

Translating lvalue Expressions

In C the targets of side effects are denoted by "lvalue expressions" which occur as subexpressions in assignment expressions and in increment/decrement expressions. An lvalue expression has the same form as any other expression and is only determined by its position as subexpression in the surrounding expression. However, expressions which may occur as valid lvalue expressions are restricted mainly to variables, container component accesses, and pointer dereferences.

Gencot translates lvalue subexpressions in the same way as other subexpressions to a binding list using `bindExpr`. As described above, for all cases where the subexpression is a valid lvalue expression, the resulting `ExprBinds` also specifies a variable `v` which can be used to modify the lvalue by re-binding it to the new value. This variable is always either a mapped C object identifier, or a component variable.

The binding list of the translated lvalue expression always contains a binding

```
v<m>' = v
```

which makes the value of `v` available for read accesses using the value variable `v<m>'`. If the expression is used as an lvalue, the additional binding

```
(v<n>',v) = (v<k>',v<k>')
```

is inserted immediately afterwards into the binding list. Here `v<k>'` is the value variable bound by the translated expression corresponding to the new value assigned as side effect. It is also the result of the assignment expression, therefore it is additionally bound to the value variable `v<n>'`. Note that this is the only case where a value variable can be used twice, however, it is always used in the same binding.

Of course, appending this binding makes the binding for the read access obsolete, because `v<m>'` is never used. However, such unused bindings are removed in later processing steps anyways, so Gencot does not care about it here.

As an example, for the assignment expression `x = 5` the concatenated binding lists for the subexpressions yield the list

```
v1' = 5
v' = x
```

which is extended for the translated assignment expression by appending the binding

```
(v2',x) = (v1',v1')
```

so that `v2'` represents the result value of the assignment expression and `x` is the modified side effect target.

The translation of `a[i].m = 0` yields the binding list

```
v2' = 0
v1' = i
(a @{@v1'=p1'},i1') = (a,v1')
v' = p1'
p1'{m=p2'} = v'
v3' = p2'
(v4',p2') = (v2',v2')
p1' = p1'{m=p2'}
a = a @{@i1'=p1'}
```

with the putback bindings immediately after the binding for the assignment.

Translating Expressions with Side Effects

C Expressions with side effects are assignments and application of increment/decrement operators. The lvalue subexpression is always translated as described in the previous section. The cases differ in how the value variable `v<k>'` for the new value is determined.

For an assignment expression there are two subexpressions, the subexpression `e1` for the lvalue and the subexpression `e2` for the rhs. Let `v<j>'` be the value variable bound in the translation of `e1` and `v<k>'` be that for `e2`. The binding list for the assignment expression is constructed by appending the list for `e1` after that for `e2`. Then an additional binding is appended which is

```
v<m>' = v<k>'
```

for a plain assignment `e1 = e2` and has the form

```
v<m>' = v<j>' + v<k>'
```

for an assignment with an operator, such as `e1 += e2`. After that the binding for the lvalue modification

```
(v<n>',v) = (v<m>',v<m>')
```

is appended to the main list, where `v` is the variable to be set by the lvalue, as described in the previous section.

For an increment/decrement operator such as `e++` there is only one subexpression `e`. Let `v<j>'` be the value variable bound in the translation of `e`. An additional binding list for the literal `1` is constructed and prepended to that for `e`. If it binds the value variable `v<k>'` the additinal binding appended to the main list of `e` is

```
v<m>' = v<j>' + v<k>'
```

Then for a postfix increment/decrement operator, such as `e++` the binding for the lvalue modification is

```
(v<n>',v) = (v,v<m>')
```

and for a prefix increment/decrement operator, such as `++e` it is

```
(v<n>',v) = (v<m>',v<m>')
```

Conditional Evaluation

In C there are cases where subexpressions are only evaluated conditionally. These cases are the logical operators `&&` and `||` where the evaluation of the second operand depends on the value of the first operand and the conditional operator where the evaluation of the second and third operands depend on the value of the first operand.

The side effects of a conditionally evaluated subexpression occur only if it is evaluated. In the translation Gencot uses a binding for all side effect targets to a common conditional expression which either returns the tuple of modified values or the tuple of the original side effect targets.

The logical operators are conceptially rewritten as conditional expressions:

```
e1 && e2 -> e1 ? e2 : False
e1 || e2 -> e1 ? True : e2
```

A conditional expression `e0 ? e1 : e2` is translated to a binding list as follows. First the three subexpressions are translated to binding lists `bs0`, `bs1`, and `bs2` by `bindExpr`. Let `v<j>'`, `v<m>'`, and `v<k>'` be the value variable to which the result of `e0`, `e1`, `e2`, respectively, is bound in the translation. Let `x1..` and `y1..` are the side effect targets in the translations of `e1` and `e2`, respectively, and let `z1..` be the union of both in some order. The binding list of the translated expression is the list of `e0` to which the binding

```
(v<n>',z1..) =
if v<j>'
then let bs1 in (v<m>',z1..)
else let bs2 in (v<k>',z1..)
```

is appended. According to the type of `v<j>'` the condition in the conditional expression is adjusted in later stages.

In most realistic cases the conditionally evaluated subexpressions have no side effects. Then it is irrelevant whether they are evaluated or not. It would be possible to appended the lists `bs1` and `bs2` to the list of `e0` outside of the conditional expression and replace the `let` expression in the corresponding condition branch by the tuple `(v<m>',z1..)` or `(v<k>',z1..)`, respectively. However, then implications of the condition cannot be used in `bs1` and `bs2`. At least NULL test conditions are exploited during postprocessing to change the type of tested values from `MayNull` wrapped to unwrapped. To make this possible the lists `bs1` and `bs2` are always placed in the condition branches, as shown above.

As an example, the expression `x && (v = i)` is translated to

```
v' = x
(v4',v) =
  if v' then
    let v2' = i
    and (v1',v) = (v2',v2')
    in (v1',v)
  else let v3' = False in (v3',v)
```

which will be simplified to `(v4',v) = if x then (i/=0,i) else (False,v)` in later stages, whereas the expression `(v = i) && x` is translated to

```
v1' = i
(v',v) = (v1',v1')
v4' = if v'
  then let v2' = x in v2'
  else let v3' = False in v3'
```

which will be simplified to `(v = i) and (v4' = if i/=0 then x else False)` in later stages.

3.6.11 Additional Type Information

Although the translation described in Section 3.6.10 is mostly independent of any type information, additional type information must be provided in the resulting Cogent AST to be used during postprocessing (see Section 3.6.6). This is done by constructing a Cogent type for every expression and pattern in the bindings of the Cogent AST and also for all subexpressions and subpatterns. These types are inserted into the AST elements for expressions and patterns.

The type information is either taken from the C program or it is constructed according to the structure of an expression or pattern from the types of sub components. Additionally, the item properties have to be taken into account, which affect the type translation.

The analysis module of the language-c parser provides type information in two ways: either from the symbol map using the monadic action `lookupObject` (see Section3.6.14), where type information is provided for all defined or declared identifiers, or from a monadic action `tExpr` which is used to typecheck expressions during the analysis phase.

The action `tExpr` is defined by language-c as

```
tExpr :: [StmtCtx] -> ExprSide -> CExpr -> m Type
```

where `m` is a `MonadTrav` (see Section3.6.14). The `StmtCtx` is only used for "statement expressions" which are a GNU extension of C which is not supported by Gencot, therefore an arbitrary value can be used for it. The `ExprSide` specifies whether the expression occurs as lvalue or rvalue and only affects the admissible types. Here the value for rvalue is used since no restrictions apply if expressions are used as rvalue. If the typecheck is successful, `tExpr` returns the C type of the checked expression in the same form as used in the symbol map. Since Gencot assumes a correct C program the typecheck functionality is irrelevant, only the resulting type is used.

Expressions for Constants and Operator Application

For constants and the results of operator applications the type information is always directly determined from the C program. Item properties are not relevant, because constants and operators are no items, so no properties can be specified for them. The type of operator results may depend on the type of the parameters, however, C operators only exist for arithmetic types, which are not affected by item properties.

For constants language-c distinguishes in the C AST between Integer, Char, Float, and String constants. Float constants are not supported by Gencot. Char constants in C have type `int`, which is mapped to Cogent type `U32` (see

Section 2.6.2). For String constants Cogent type `String` is used. For Integer constants and operator applications the C type is determined using `tExpr`, then it is mapped to a Cogent type as described in Section 2.6.2 which results in one of the types `U8`, `U16`, `U32`, or `U64`.

In C there is no specific type for boolean values, they are represented by values of integer or pointer type with `False` being represented by `0` and `True` by arbitrary other values. In Cogent there is type `Bool` which is used for the result of equational, relational, and boolean operators and as expected type for boolean operators and the condition in conditional expressions.

Since the expected types for operators and conditional expressions are not made explicit in the Cogent AST, type `Bool` is only used for the result of equational, relational, and boolean operators. Cases of expected type `Bool` are handled in postprocessing steps.

Expressions for Variables

For an identifier first the C type is determined using `tExpr` (which internally looks it up in the symbol map). Since every identifier may denote an "item" with declared item properties on which its Cogent type may depend (see Section 3.2), next its associated item id must be determined, then both are combined to an `ItemAssocType` (see Section 3.2.2) and mapped to a Cogent type by `transType` (see Section 3.11.3).

This way, the type of a preprocessor constant is determined according to the dummy declaration generated for it. This will always be `int` because the dummy declarations are generated in this form (see Section 3.6.5). A special case is the constant `NULL`. It is detected by its name, then its type is set to `MayNull CVoidPtr`. This is necessary to determine the correct type for a conditional expression, as described below, if one branch is the `NULL` value.

The item id of an identifier depends on whether the identifier denotes a toplevel object or function, a parameter of the surrounding function definition, or a local variable declared in the function body. However, this information is no more available after the analysis by language-c. Looking the identifier up in the symbol map only allows to distinguish toplevel definitions from local declarations, but neither parameters from local variables, nor different local variables with the same name.

Therefore Gencot implements its own "item id table" for this purpose: it maintains a stack of local contexts where each context maps identifiers to their item id. Only local identifiers in function definitions are managed here, the function parameters are in the outermost context and at the end of a function definition all contexts are removed. See Section 3.6.14 for the corresponding monadic actions.

Component Access

As described in Section 3.6.10, accesses to container components (struct members, array elements, pointer dereferences) are translated to pairs of `take` and `put` operations. In Cogent both operations modify the container type, reflecting the fact that the component has been removed or inserted.

During translation Gencot does not apply these type modifications in the additional type information, the same type is always associated to a container.

The reason is that the additional type information is not exposed in the final Cogent program, it is only used by Gencot for postprocessing. There it is not needed, because `take` and `put` operations are always generated in pairs with no other `take` and `put` operations for the same container in between. During postprocessing a `take` operation may be converted to a readonly component access (which does not modify the container type) and a `put` operation may be removed, because the component has not been modified, in these cases the type modifications would have to be reverted.

The Cogent type of the component can always be determined from the Cogent type of the container, because it is a syntactic part of the container type.

In C, if the component is an (embedded) array, component access does not result in a copy of the array, instead, it results in a pointer to the first array element. In Cogent this corresponds to a value of the boxed mapped array type. Since embedded array components are always represented by the unboxed array type (see Section 2.6.5), the type of the component variable is adjusted to the boxed component type, whenever the component is of an unboxed array type. This is not correct for Cogent `take` and `put` operations, they will be replaced during postprocessing usually by `getref` and `modref` applications (see Section 2.7.6).

The boxed component type is not wrapped as `MayNull` because the access of an array as component always yields a valid pointer, if the pointer to the container is valid. It may only be readonly if the container has a readonly type. An array which is embedded in a container of linear type is always itself linear. If a Read-Only property has been declared for the component it will only affect the types of sub-components, because an embedded component is translated to have unboxed type.

Function Pointers

When function pointers are used to invoke the referenced function they must be dereferenced. This may be done explicitly, using the indirection operator `*`, or implicitly by simply applying the function pointer to arguments. In Cogent the dereferencing must always be explicit using the abstract function `fromFunPtr` (see Section 2.7.9). For the corresponding expression the function type must be associated.

However, since Gencot currently represents function pointer types as abstract types with encoded type names (see Section 2.6.7), it is not possible to determine the function type from the function pointer type in Cogent. As solution, Gencot first determines the C type from the function pointer expression using `tExpr`, then determines the C function type from it, and finally translates that to Cogent using `transType`.

For this translation Gencot needs the item id of the function pointer. However, the function pointer need not be specified by an identifier in C, it may be an arbitrary expression like a struct member access or the result of a function application. Therefore the item id is constructed according to the expression structure from the item id of a subexpression which is an identifier (and which must always be present for a function pointer expression).

When Gencot is modified to use structured Cogent types for function pointers this approach is not needed anymore, then the Cogent function type can be directly constructed from the Cogent function pointer type.

Function Application

In a function application the Cogent type of the function is determined by translating its C type using `tExpr` and `transType`. If the function is specified by a function pointer the function type is determined as above, otherwise the function is specified by its name, so that the item id can be determined immediately. Since all relevant item properties are respected by the translation, the Cogent type includes all additional arguments (for Global-State, Heap-Use, and Input-Output properties) and all additional result components (for Global-State, Heap-Use, Input-Output, and Add-Result properties).

The type of the function application expression in Cogent is simply taken from the function type. The type of the function argument (which is a tuple type of the argument types or a single argument type) is instead constructed from the types of the actual argument expressions. This may result in type differences between the actual argument type and the formal argument type specified as part of the function type. These differences may origin from similar differences in the C program, which are resolved in C by automatic type conversion, or they may origin from item properties which are applied to the arguments and/or the function in different ways. These type differences are resolved by Gencot in postprocessing stages.

Conditional Expressions

A conditional C expression is translated to a conditional expression in Cogent. The type of the conditional expression must be the same as the type for both branches. However, the branch types may differ, due to differences in the C program or to differences in item property applications.

The C type of the conditional expression includes the resolved differences in C, but does not resolve the differences caused by item properties, therefore it cannot be used to determine the Cogent type of the conditional expression. Instead, it is constructed from the Cogent types of the branches. This is based on the fact that type differences caused by item property application are always resolved to one of both types in a deterministic manner. For example, a readonly type and a linear type are always resolved to the readonly type, because a value of linear type can be converted to readonly by banging it, but not vice versa. This makes it possible to pre-calculate during translation the type which will later result from postprocessing the type differences for both branches. This type is used for the conditional expression.

Patterns

The patterns used in the generated bindings to bind expression results always consist of single variables, `take` patterns, or (flat) tuples of these. For variables which result from translating a C identifier, the type is determined as described above. For component variables the type is determined by the component type, which is determined by the container type, as described above. For value variables the associated type is always the type of the expression bound to it. Since value variables are reused, their associated type may change as soon as they are re-bound to another expression. For a `take` pattern the type is the same as the container type, it is not modified to a "taken" type, as described above.

The type of a tuple pattern is simply constructed as the tuple type of the types of the sub patterns.

3.6.12 Translating C Statements

Like C expressions, C statements are mainly translated to Cogent bindings.

General Translation of C Statements

The translation of C statements differs from that of C expressions in two major ways. A C statement has no result value, therefore its translation never binds a value variable. But a C statement may cause jumps, these are translated to conditional evaluation with the help of an additional "control variable" `c` as described in Section 2.10.3.

However, the concept described there is modified with respect to the control variable and the return value. Instead of using type `(Bool, Bool, Option T)` for the control variable the implementation uses an integer type and encodes nonlocal jumps by the values `3` (`return`), `2` (`break`), `1` (`continue`), and `0` (no jump). In case of a `return` jump the third component only represents the fact that a jump has been caused. The return value is represented separately by binding a value of type `T` to a special "result variable". Gencot uses the name `r'` for the result variable, it is automatically distinct from the names of all other variables used in the translation. It is normally treated like an additional side effect target.

The reason for this modification is as follows. If a value of the option type is maintained, code for the `None`-case must be generated, although this case never occurs if the C program is syntactically correct. If the result type `T` is linear, the code for the `None`-case will always violate the Cogent type constraints, because it either uses a default value (which does not exist for linear types) or it discards the value (which is not allowed). When using the result variable the `None`-case may correspond to the situation where the result variable occurs free in the function body. This would also be an error for the Cogent compiler, however, if the C program is syntactically correct all these occurrences can be eliminated from the Cogent code. In its initial translation stage Gencot generates these free occurrences of the result variable, they are eliminated in later stages.

Instead of using a tripel of boolean values Gencot encodes the possible nonlocal jumps in an integer value, since that is more compact and simplifies the later stages.

Values of expressions which are syntactically a part of a statement can never be used in other statements. Therefore all value variables can be reused in every statement. Gencot supports this by resetting the value variable counter `<n>` whenever it starts translating a statement.

C Statements are always evaluated in a sequenced manner or conditionally. The value of the control variable is used exactly once, immediately after evaluating the translated statement. Therefore a single control variable is sufficient for all statements. Gencot uses the name `c'` for the control variable, it is automatically distinct from the names of all other variables used in the translation.

The approach described in Section 2.10.3 always converts the bindings which result from expression translation to Cogent expressions before they are used as part of a translated statement. This means that taken container components

are put back locally in every translated expression. In later postprocessing steps Gencot may try to reuse taken components over several expressions in a statement and even over several sequential statements. To prepare for reusing taken components, Gencot never reuses component variables in different statements of the same function body.

All this implies that a translated statement can always be represented by a single binding, no additional information about a value variable and an lvalue variable is required, in contrast to translated expressions.

Gencot uses a common monadic action

```
bindStat :: CStat -> FTrav GenBnd
```

to translate every statement to a binding of the form

```
(c',v1..) = expr
```

with side effect targets `v1...` The expression `expr` covers the effect on the control variable and all side effects of the translated statement. The side effect targets may include the result variable `r'`.

If Gencot cannot translate a statement it translates it to a binding where the rhs is a dummy expression as described in Section 2.10.4. The lhs contains the control variable and may still contain side effect targets, if Gencot detects that the statement modifies these C objects.

Expression Statements and Null Statements

An expression statement has the form `e;`. It is translated by translating `e` to an `ExprBinds` with a binding list `bs`. Then the translation of the expression statement is the binding

```
(c',v1..) = let bs in (0,v1..)
```

where `v1..` are the side effect targets of `bs`. This discards the value `v<n>'` of `e`.

A null statement has the form of a single semicolon `;` and is translated to the binding

```
c' = 0
```

Jump Statements

A jump statement is a `return` statement, `break` statement, `continue` statement, or `goto` statement. Gencot does not translate `goto` statements.

For a `return` statement with an expression the expression is translated to an `ExprBinds` by `bindExpr` with a binding list `bs`. Then the translation of the return statement is the binding

```
(c',r',v1..) = let bs in (3, v<n>',v1..)
```

where `v<n>'` is the value variable bound in `bs` to the result of the expression. This binding explicitly binds the result variable `r'` as additional side effect target. The type of `r'` is always taken from the formal function result type. It may differ from the type of `v<n>'` due to automatically resolved type differences

in C or due to item property applications. Such type differences are resolved during postprocessing.

For a `return` statement without an expression only the control variable is bound, the statement is translated to the binding

```
c' = 3
```

This implies that for a C function with result type `void` the result variable will never be bound in the Cogent code. The result value of the Cogent function will be generated in later stages.

A `break` statement is translated to the binding

```
c' = 2
```

A C `continue` statement is translated to the binding

```
c' = 1
```

Local Declarations

Local declarations are declarations which occur as an item in a compound statement. Gencot only translates local object declarations, it does not translate local type definitions and static assertions.

A local object declaration consists of a sequence of declarators. Every declarator declares a single object identifier and may specify an initializer for the object's initial value. An initializer is either an expression or an initializer for an aggregate or union type. Gencot translates every initializer in the same way as an expression to an `ExprBinds`.

A declarator is translated to a pair consisting of a variable name and an `ExprBinds`. The variable name is the declared identifier mapped to a Cogent name, as described in Section 3.6.7. If the declarator has an initializer the `ExprBinds` results from its translation. Otherwise an `ExprBinds` with the single binding `v' = defaultVal ()` is used.

A local object declaration is translated to the sequence of the translations of all declarators.

Compound Statements

Compound statements are blocks which contain a sequence of block items which are statements or (local) declarations. Block items are conceptionally grouped right-associatively. Thus a block item sequence is either empty, or a single block item (at the end), or it is a block item followed by a block item sequence.

A declaration declares identifiers which have the rest of the block as their scope. Thus the scope is always the block item sequence which begins with the declaration and ends at the end of the block. The declared object identifiers can occur as side effect targets in their scope, but the side effects can only be observed in the scope and must not be propagated outside it.

A declaration may consist of several declarators. Every declarator declares a single identifier which has the following declarators and the block after the declaration as its scope. Thus, conceptually, the declarators can be considered to be the block items instead of the declarations.

If a declarator has an initializer, the initializer is not part of the scope of the declared identifier. A declarator may redeclare an identifier which has been declared outside of the block associated with the declarator, in this case the identifier is shadowed in the scope of the redeclared identifier.

Gencot translates every block item sequence to a Cogent binding of the same form as that for a statement. It covers the effect to the control variable and it covers all side effects to variables not declared in the sequence.

The translation of a compound statement is the translation of its block item sequence.

An empty block item sequence is translated like a sequence with a single null statement to the binding

```
c' = 0
```

For a block item sequence consisting of a single statement the translation of the sequence is identical to that of the statement.

A block item sequence consisting of a statement `s` followed by a nonempty sequence `bis` is translated as follows. First `s` and `bis` are translated to bindings `(c',x1..) = expr1` and `(c',y1..) = expr2`. Let `z1..` be the union of the side effect targets `x1..` and `y1..` in some order. The block item sequence is translated to the binding

```
(c',z1..) = let (c',x1..) = expr1
in if c' > 0
   then (c',z1..)
   else let (c',y1..) = expr2 in (c',z1..)
```

A block item sequence consisting of a single declaration is translated by appending a null statement.

A block item sequence consisting of a declaration `d` followed by a nonempty sequence `bis` is translated by translating the declaration to a sequence of pairs of a variable and a binding. The sequence `bis` is translated to a binding. Then these parts are combined starting with the last pair and the binding.

As described above a pair consists of a variable `v` which corresponds to the declared identifier and an `ExprBinds` with a binding list `bs` which binds the value of the initializer or the default value to `v<n>'` and has the side effect targets `v1...` It is combined with a binding `(c',y1..) = expr2` as follows. Let `u1..` be `y1..` without `v`. Let `z1..` be the union of `v1..` and `u1...` Then the combined binding is

```
(c',z1..) =
  let bs
  and (c',u1..) =
    let v = v<n>'
    and (c',y1..) = expr2
    in (c',u1..)
  in (c',z1..)
```

This translation also accounts for the case that `v` is a side effect target of its own initializer so that the side effect is correctly applied to a shadowed instance of `v`.

Function Bodies

A C function body is always a compound statement. It is translated by `bindStat` to a binding of the form

```
(c',v1..) = expr
```

From this binding the Cogent expression

```
let (c',v1..) = expr
in <result expression>
```

is constructed as body expression of the translated Cogent function. The `<result expression>` depends on the item properties Add-Result, Global-State, Heap-Use, and Input-Output.

In the special case of the function named `main` in C the default result `0` is added by prepending a binding for the result variable `r'`:

```
let r' = 0
and (c',v1..) = expr
in <result expression>
```

so that `r'` is bound even if the function does not contain `return` statements.

If no additional result components are caused by the item properties, the `<result expression>` is

```
()
```

if the C function result type is `void`, and

```
r'
```

otherwise.

If the item properties cause additional result components the `<result expression>` is

```
((), v1..)
```

or

```
(r', v1..)
```

respectively, where the variables `v1..` correspond to the properties Add-Result, Global-State, Heap-Use, and Input-Output in that order.

The control variable `c'` is never used in the `<result expression>`, it may only be used in sub-statement. Its binding in the function body will be removed during postprocessing.

Selection Statements

A selection statement is a conditional statement or a switch statement.

A conditional statement of the form `if (e) s1 else s2` is translated similar to a conditional expression. First the subexpression `e` is translated to an `ExprBinds` with a binding list `bs` which binds the value of `e` to `v<n>'` and has the side effect targets `v1...` The substatements are translated to bindings `(c',x1..) = expr1` and `(c',y1..) = expr2`. Let `z1..` be the union of the side effect targets `v1..`, `x1..`, and `y1..` in some order. The conditional statement is translated to the binding

```
(c',z1..) = let bs
in if v<n>'
   then let (c',x1..) = expr1 in (c',z1..)
   else let (c',y1..) = expr2 in (c',z1..)
```

For a conditional statement of the form `if (e) s1` the expression `e` and statement `s1` are translated as above. Let `z1..` be the union of the side effect targets `v1..` and `x1..` in some order. The conditional statement is translated to the binding

```
(c',z1..) = let bs
in if v<n>'
   then let (c',x1..) = expr1 in (c',z1..)
   else let c' = 0 in (c',z1..)
```

A switch statement of the form `switch (e) s` is only translated if it has the restricted form described in Section 2.10.3: The statement `s` must be a compound statement `{s1 ... sn}` where no `si` is a declaration and all `case` and `default` statements associated with the `switch` statement occur among the `si`.

The translation has the form described in Section 2.10.3, however, the C code is not rewritten but translated directly. This is done by processing the body `s` in a specific way which handles the `case` and `default` statements among the `si`. At all other positions `case` and `default` statements are translated to dummy expressions. Declarations among the `si` are also translated to dummy expressions. Statements before the first `case` or `default` statement in the `si` are silently omitted.

The value of `e` must be assigned to a specific "switch variable". Nested switch statements are translated encapsulated in subexpressions where the switch variable shadows that from all surrounding switch statements, therefore a single switch variable is sufficient for all statements. Gencot uses the name `s'` for the switch variable, it is automatically distinct from the names of all other variables used in the translation.

The case labels are in real programs mostly literals, however, syntactically they are expressions. In the Cogent code a case label may be used in several places, therefore Gencot evaluates them once at the beginning of the switch statement. This is possible because the case labels must be constant expressions which implies that they can be evaluated independent of their context. After evaluation Gencot immediately compares the result to the value of the switch variable and binds the result to specific "case variables". For a default statement the case variable is bound to `True`. Gencot uses the names `s1'`, ... for the case variables, which are automatically distinct from the names of all other variables used in the translation. Like the switch variable they can be reused for nested switch statements where they shadow the case variables from surrounding switch statements.

Every `case` or `default` statement `sk` of the form `lbl: sk0` in the `si` is grouped together with all following statements `sk1 ... skmk` which are no `case` or `default` statement to the compound statement `{sk0 sk1 ... skmk}`. Let `g1 ... gp` be the sequence of these compound statements and let `lbl1 ... lblp` be the corresponding labels in the `case` or `default` statements. For

each `case` statement let the label `lblk` be `case ck`, i.e., `ck` is the constant used in it.

Every compound statement `gk` is translated to a binding `(c',x1..) = exprk` by applying `bindStat` to it. Then this binding is extended to a binding `bk` of the form

```
(c',x1..) =
  if condk
  then exprk
  else (0,x1..)
```

which corresponds to the C statement `if (condk) then {sk0 sk1 ... skmk}`. The condition `condk` is determined as follows. If there is no `default` statement before `sk` the condition is

```
s1' || ... || s<k>'
```

Otherwise the condition is

```
not (s<k+1>' || ... || s<p>')
```

For `k = p` this condition is equivalent to `True` and the binding `(c',x1..) = exprp` is used directly. Note that if `sk` is the default statement the case variable `s<k>'` (which has been bound to `True`) is never used.

The resulting sequence of bindings `b1 ... bp` is combined to a binding `(c',x1..) = exprb` by processing it as described for a compound statement, as if the bindings resulted from translating statements which are the block items in the compound statement. The controlling expression `e` is translated to an `ExprBinds` and combined by `cmbBinds` to a binding `(v<n>',v1..) = expr`. The case label expressions `e1`, ..., `ep` are translated to `ExprBinds` and combined by `cmbBinds` to bindings `v<n>' = exprk` (there are no side effect targets because the `ei` are constant expressions). Let `z1..` be the union of the side effect targets `v1..` and `x1..` in some order. Then the translation of the `switch` statement is the binding

```
(c',z1..) = let (s',v1..) = expr
  and (s1',...,s<p>') = (s' == expr1,...,s' == exprp)
  and (c',x1..) = exprb
  in (if c'=2 then 0 else c',z1..)
```

where the new control value reflects that a switch statement consumes a `break` jump (represented as 2) but passes a `continue` or `return` jump to its context.

If a branch in a selection statement uses a `return` statement its translation binds the result variable `r'`. Then all other branches must also bind `r'` so that the selection statement can bind it as a whole. If another branch does not use a `return` statement (atleast not in all its sub-cases) this will result in a reference to an unbound instance of `r'`. This will be detected statically by the Cogent compiler and signalled as an error.

It would be possible to avoid this by binding `r'` to a default value at the beginning of each translated function. However, the Gencot `default` operation (see Section 2.7.2) does not support default values for arbitrary result types. Therefore Gencot does not handle this case, it must be handled by modifying the C program so that for every selection statement either all branches are left

by a `return` statement or none. Note that this is not necessary for `break` and `continue` statements.

In C this situation is often handled by sequentially executing a `return` statement after the selection statement which will handle all branches not left by a `return` statement. Then the C program can be modified by moving this `return` statement into all branches. A specific situation is a `switch` statement without a default case, because it always has an "invisible" empty default case. If all explicit cases are left by a `return`, there is still the implicit default case for which this is not the case. Thus it must be changed by adding an explicit default case which is also left by a `return` statement.

Iteration Statements

An iteration statement is a `for` loop or a `while` loop.

Gencot only translates specific `for` loops. These loops are translated using the abstract function `repeat` as described in Section 2.10.3. A `for` loop is only translated for some cases where an upper limit for the number of iterations can be determined automatically from the loop structure.

In a `for` loop the initialization part can be a declaration or a subexpression. If it is a declaration, the `for` loop is the scope of the declared variable(s).

The general translation of a loop `for (c1; e2; e3) s` with a clause `c1` which is either a declaration or an expression and where `e2` has no side effects is as follows. The statement `if e2 then s else break;` is translated to a binding `(c',x1..) = expr` by applying `bindStat` to it. The expression `e3` is translated to a binding list `bs3` by applying `bindExpr` to it. Conceptually, `e3` is appended as an expression statement to `s`, by combining all bindings to the binding `(c',y1..) = exprstep` defined by

```
(c',y1..) = let (c',x1..) = expr
in if c' > 1
   then (c',y1..)
   else let bs3
        in (0,y1..)
```

where `y1..` is the union of the side effect targets in `bs3` and `x1...` Note that the condition uses `c' > 1` (`break` or `return`) instead of `c' > 0` (`break`, `return`, or `continue`) to correctly translate the semantics of `continue` statements in the body. A `continue` statement causes the body to abort, but still executes `e3`, whereas `break` or `return` statements end the loop without executing `e3`.

Then the binding `(c',y1..) = exprloop` is constructed defined by

```
(c',y1..) =
let (c',y1..) = repeat #{
  n = exprmax,
  stop = \#{acc = (c',y1..), obsv = (w1..)}
         => c' > 1,
  step = \#{acc = (_,y1..), obsv = (w1..)}
         => exprstep
  acc = (0,y1..), obsv = (w1..)
  }
in (if c'=2 then 0 else c',y1..)
```

where `w1..` are all variables occurring free in `exprstep` which are not part of `y1..`, in some order. The expression `exprmax` for the upper limit of number of iterations is constructed as described below.

Here the new control value is determined like for a switch statement, by consuming the effect of a `break` statement. Note that `continue` statements are already handled in `exprstep` above, where in the `else` case the control value is set to `0` which consumes the effect of a `continue` statement. As a consequence the control value returned by `repeat` can never be `1` so that it is not necessary to handle this case here.

Finally, if `c1` is not empty its translation is prepended to the binding `(c',y1..) = exprloop` in the way as described above for a block item sequence. If `c1` is an expression it is translated to a binding list `bs1` by applying `bindExpr` to it. Then the translation of the `for` statement is the binding

```
(c',z1..) =
let bs1
and (c',y1..) = exprloop in (c',z1..)
```

where `z1..` is the union of `y1..` and the side effect targets in `bs1` in some order. This corresponds to a simplified sequence of `bs1` and the `exprloop` binding exploiting the fact that the first results from an expression and can therefore not contain a jump.

If `c1` is a declaration it is translated to a sequence of pairs of a variable and a binding. Then this sequence is combined with the binding `(c',y1..) = exprloop` beginning with the last pair, as described above for a declaration in a block item sequence.

Gencot translates a loop `for (c1; e2; e3) s` in this way only if it can construct an expression `exprmax` for the upper limit of number of iterations. For this, the loop must have the following properties:

- The expression `e2` must be a relation `v rel evl` where `v` is a variable not occurring free in `evl` and `rel` is one of the relation operators `<`, `<=`, `>`, `>=`, `!=`, or `e2` must be a conjunction which contains such a relation. The body statement `s` must not modify `v` or `evl` and the expression `e3` may only modify `v` but not `evl`.

- The expression `e3` must be an assignment expression for the same variable `v` or it must be a sequence of comma expressions which contains exactly one such assignment.

- If `rel` is `<` or `<=` the assignment in `e3` must have one of the forms `v++`, `v+=stp`, `v=v+stp`, or `v=stp+v` where `stp` is an integer literal. In this case let `emax` be the expression `evl`, if `rel` is `<=` let `emax` be `evl+1`.

- If `rel` is `>` or `>=` the assignment in `e3` must have one of the forms `v--`, `v-=stp`, or `v=v-stp` where `stp` is an integer literal. The clause `c1` must be an assignment expression of the form `v=ini` or a sequence of comma expressions which contains such an assignment, or it must be a declaration with a declarator for the variable `v` and initialization expression `ini`. In this case let `emax` be the expression `ini`, if `rel` is `>=` let `emax` be `ini+1`.

- If `rel` is `!=` the expression `evl` must be an integer literal and the clause `c1` must be an assignment expression of the form `v=ini` or a sequence of

comma expressions which contains such an assignment, or it must be a declaration with a declarator for the variable `v` and initialization expression `ini`. In both cases `ini` must be an integer literal. If `ini` is lower than `evl` the assignment in `e3` must have one of the forms `v++`, `v+=1`, `v=v+1`, or `v=1+v`, then let `emax` be the expression `evl`. If `ini` is greater than `evl` the assignment in `e3` must have one of the forms `v--`, `v-=1`, or `v=v-1`, then let `emax` be the expression `ini`.

If the loop satisfies these properties the resulting C expression `emax` is translated to a binding list `bs (v<n>',v1..) = expr` by applying `bindExpr` to it. Then the expression `exprmax` is

```
let bs in v<n>'
```

where `v<n>'` is the value variable bound in bs to the result of the expression. Note that side effect targets could occur in `bs` if `emax` is `ini`. In this case the side effects are already handled by the translation of the loop and can be ignored here.

The motivation for this approach is as follows. To determine an upper limit for the number of iterations a "counting" variable `v` must be present. In `e2` it must be compared to a value `evl` which is constant for all iterations, therefore neither `v` nor a variable modified by the body may occur free in it. To safely determine the counting direction, `stp` must be a literal, otherwise its evaluation could result in a negative number which would reverse the counting direction. Also, `v` must only be modified by `e3` and not by the body `s`.

If the variable counts up and the comparison in `e2` uses `<` or `<=` the comparison value `evl` yields an upper limit for the number of iterations, independent of the step size and the initial value of `v`. Therefore `c1` may even be omitted. If the variable counts down and the comparison in `e2` uses `>` or `>=` the initial value `ini` yields an upper limit for the number of iterations, independent of the step size and the comparison value `evl`.

If the comparison uses relation `!=` the situation is similar, however `v` must be counted in steps of 1 to safely hit the comparison value and it may not wrap around zero during the iterations so that the upper limit is valid.

To decide the first property the side effects of the body statement `s` must be known. However, the side effect targets `x1..` which result from translating `s` are in general neither correct nor complete. Due to translating struct and array accesses using `take` and `put` operations all structs and arrays where a member or element is accessed occur as side effect targets. Read-only accesses are only detected and removed in later stages. It may also be the case that `s` modifies only a part of a struct and another part is accessed in the comparison value `evl`.

To check whether a counting variable `v` is modified in the body `s` Gencot checks whether `v` occurs on the lhs of an assignment in `s`. The only way how `v` can be modified in `s` without being detected in this way is if it is modified through a pointer retrieved with the help of the address operator `&` (either in `s` or before the loop) which must be handled manually anyways.

To check whether `s` or `e3` can modify the comparison value `evl` Gencot checks whether any non-literal operand occurring in `evl` occurs on the lhs of an assignment in `s` or `e3`. In particular, if a struct member occurs in `evl`, assignments to other members of the same struct in `s` or `e3` will not prevent the translation of the loop.

The variables `w1..` which occur free in `exprstep` or `expr2` can either be determined from the C source (variables free in `e2`, `e3`, or `s`) or from the translated Cogent source. Since Cogent does not support accessing defined constants in a lambda expression these must be included. This is automatically done, because in Cogent constants are syntactically referenced in the same way as variables. In C, as described in Section 3.6.5, Gencot generates dummy declarations for preprocessor constants, so syntactically they are variables as well. Since all other information used here for translating `for` statements is taken from the C source, Gencot also determines the variables `w1..` from the C source. This implies that they must be explicitly mapped to Cogent names. This mapping must be done at a place where all declarations in the context of the loop and also a declaration which occurs as clause `c1` in the loop are visible, since the mapping depends on the linkage of the name which must be looked up in the symbol table.

As an example, the loop `for (int i=0;i<b;i++) a*=2;` is translated according to these rules to the (simplified) binding

```
(c',i,a) =
  let v' = 0
  and (c',a) =
    let i = v'
    and (c',i,a) =
      let (c',i,a) = repeat #{
        n = b,
        stop = \#{acc = (c',i,a), obsv = b}
         => c' > 1
        step = \#{acc = (_,i,a), obsv = b}
         => let v1' = i<b
            in if v1'
               then let (c',a) = (0,2*a)
                    in if c' > 1
                       then (c',i,a)
                       else let (v2',i) = (i,i+1)
                            in (0,i,a)
               else let c' = 2
                    in (c',i,a)
        acc = (0,i,a), obsv = b
        }
      in (if c' = 2 then 0 else c',i,a)
    in (c',a)
  in (c',i,a)
```

which can be further simplified to

```
(c',i,a) =
let (i,a) = repeat #{
  n = b,
  stop = \#{acc = (i,_), obsv = b} => False,
  step = \#{acc = (i,a)} => (i+1,2*a)
  acc = (0,a), obsv = b
  }
in (0,i,a)
```

3.6.13 Generating Antiquoted C Code

In addition to the Cogent code Gencot also generates antiquoted C code. This is implemented by generating a C AST and then prettyprinting it. For this the reimplemented language-c-quote AST described in Section 3.6.9 from module `Gencot.C.Ast` is used together with the reimplemented prettyprinter from module Gencot.C.Output.

Embedding Cogent Code

The antiquotation feature from language-c-quote is originally intended to embed Haskell code in a C source. However, the Cogent compiler uses it to allow embedded Cogent code in some specific places.

Antiquoted parts are represented in the language-c-quote AST (and its reimplementation) by alternative syntactic entities consisting of a single string containing the literate embedded code. The prettyprinter (and its reimplementation) adds the antiquotation marker such as `$ty:` and wraps it in parentheses, if necessary.

To embed Cogent code which exists as a Cogent AST fragment, it must be prettyprinted with the Cogent prettyprinter and the resulting string must be added as an antiquoted entity to the C AST. Usually, the embedded Cogent code is rather small, such as a single type. Therefore Gencot prettyprints it in a compact form without formatting or indentation. For a Cogent AST fragment `<cogent>` this is done by

```
(displayS $ renderCompact $ pretty <cogent>) ""
```

where all three functions are used from module `Text.PrettyPrint.ANSI.Leijen`. Function `pretty` is the prettyprint function, `renderCompact` does the rendering in compact form, and `displayS` converts the result to the standard Haskell type `ShowS` which is `String -> String` and must be applied to the empty string to get the Cogent code rendered as a string.

This functionality is provided by function

```
showCogent :: Pretty a => a -> String
```

defined in module `Gencot.Cogent.Output`.

Origin Markers

Since the reimplemented language-c-quote AST is used, every syntactic entity has an associated origin information.

Since the antiquoted C code is generated, normally no origin information is available and must be specified as empty. However, if there is a corresponding entity in a processed C source file it may be useful to specify its origin and use it for inserting conditional preprocessor directives from the processed source file.

Outputting the Generated Code

The generated C AST is output using function `ppr` from module `Text.PrettyPrint.Mainland`. In case of a list of C AST entities the function `pprList` is used instead. The resulting `Doc` must be rendered to a string with function `pretty` from the same module, then it can be output with the standard Haskell function `putStrLn`.

As described in Section 3.6.9 a large value must be specified as the document width to compensate for the fast column increase caused by the hidden newlines. Therefore the standard rendering output actions defined by `Text.PrettyPrint.Mainland` such as `putDoc` cannot be used since they always set the document width to 80.

Gencot defines in `Gencot.C.Output` the function

```
showTopLevels :: [Definition] -> String
showTopLevels defs = pretty 2000 $ pprList defs
```

which can be used to output a list `<antis>` of antiquoted C AST definitions by

```
putStrLn $ showTopLevels <antis>
```

3.6.14 Traversing the C AST

The package language-c uses a monad `MonadTrav` for traversing and analysing the C AST. The monad is defined in module `Language.C.Analysis.TravMonad` and mainly provides the symbol table, error messages, 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.

The package language-c provides the function

```
runTrav :: forall s a. s -> Trav s a -> Either [CError] (a, TravState s)
```

for running a monadic action of type `Trav s a` where `s` is the type of the user state and `a` is the type of the result. The function either returns a list of errors, or it returns the result together with the modified monadic state of type `TravState s`.

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 with a different type of user state 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.

Symbol Table

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.

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
```

Additionally, there are actions to enter and leave a scope and actions for inserting definitions.

Error Messages

The monad supports nonlocal jumps by throwing and catching errors, however, this feature is not used by Gencot. Gencot is implemented so that it always can proceed if it encounters an erroneous situation, if the processed C program is correct. Gencot only registers error messages in the monad and outputs the list of all error messages at the end of execution.

An error can be recorded in the monad using the action

```
recordError :: Language.C.Data.Error.Error e => e -> m ()
```

The class `Error` covers all specific error types, it provides for every error a level (warning, error, fatal), a source code position and a sequence of error message line strings. Predefined members of the class are the types `UnsupportedFeature` with position information and `UserError` with error message only, both have level error. Also predefined is the member type `CError` which is a wrapper for all other member types of `Error`. When an error is displayed, the information is shown together with the error class name as short message.

The function `runTrav` distinguishes between errors of level error or fatal and errors of level warning. If there is atleast one error of the former case it only returns the error list and no result or modified state. If there are only errors of level warning, they are returned as part of the modified monadic state and may be retrieved using the function

```
travErrors :: TravState s -> [CError]
```

Since Gencot always proceeds upon errors and produces a result, it uses only errors of level warning.

Gencot defines its own error types with names of the form `XError` with level warning and a constructor function

```
xError :: (Pos a) => String -> a -> XError
```

Class `Pos` includes all elements of the C AST, the second argument is used to determine the position.

User State

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.

A second information is the name mapping configuration, as described in Section 3.6.7. It is used by all filters and processors which translate to Cogent. It is supported by the class

```
class (FileNameTrav m) => MapNamesTrav m where
  matchPrefix :: String -> m (String, (String,String))
```

The method `matchPrefix` returns for a name the first matching prefix together with the replacements. It applies the default matching rule, so it always returns a result. The class is defined as subclass of `FileNameTrav`, although the method does not use `getFileName`, because all clients of `matchPrefix` also use `getFileName`.

Another information is the set of external type names directly used in the Cogent compilation unit. All other external type names are resolved during translation, as described in Section 2.1.2.

This is supported by defining in module `Gencot.Traversal` the class `TypeNamesTrav` as

```
class (Monad m) => TypeNamesTrav m where
  stopResolvTypeName :: Ident -> m Bool
```

so that the method `stopResolvTypeName` can be used to test a type name whether it should be resolved or not. The list can be deactivated using a flag in the user state. This is useful, if only code is processed which belongs to the Cogent compilation unit. Then all referenced external type names are directly used and no additional information about the external type names is required, the list can be deactivated and be empty.

Other information needed during C AST traversal are:

- The item property map (see Section 3.2.5). It is used for translating all types, as described in Section 2.6.1.

- A list of tag definitions which are processed in advance as nested, as described in Section 3.6.3. 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 current function definition when traversing a function body. This is not available in the symbol table.
- A table for looking up the item ids of local variables and parameters, as described in Section 3.6.11.
- Counters for generated Cogent variable names, as described in Section 3.6.10.
- A string with configuration settings for the Cogent translation.

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, NamePrefixMap, [SUERef], ItemProperties,
   (Bool,[String]), Maybe IdentDecl, LocalItemIdTable,
   (Int,Int), GlobItemMap, String)
```

where `String` is the type used for storing the original C source file name in the user state, `NamePrefixMap` is the type for the name prefix map described in Section 3.6.7, `ItemProperties` is the type for storing the item property map, and `LocalItemIdTable` is the type for maintaining the table of item ids for local identifiers. The third component is the list for maintaining tag definitions processed as nested, the fifth component is the list of directly used external type names together with the flag for activating or deactivating the list, the sixth component is the function definition whenever traversing a function body, the eighth component is a pair of variable counters, the nineth component is the mapping from item ids to definitions for all global variables, collected by the uhandler as described in Section 3.6.3, and the last component is the translation configuration string.

`FTrav` is an instance of `FileNameTrav` and `TypeNamesTrav`. As execution function for the monadic actions the functions

```
runFTrav :: DefTable -> (
   String, NamePrefixMap, ItemProperties, (Bool,[String]),
   GlobItemMap, String)
  -> 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, the name prefix map, the item property map, the list of directly used external type names, the item id map for global variables, and the configuration string as arguments to initialize the state. The tag definition list and the item id table are always initialized as empty, the function definition is set to `Nothing`, the variable counters are set to 0. The second function leaves also the other five components empty and deactivated. The functions are themselves `IO` actions and print error messages generated during traversal to the standard error stream.

To maintain the list of tag definitions processed as nested two monadic actions are defined:

```
markTagAsNested :: SUERef -> FTrav ()
isMarkedAsNested :: SUERef -> FTrav Bool
```

The item property map can be accessed by the monadic actions

```
getItems :: (String -> [String] -> Bool) -> FTrav [String]
getProperties :: String -> FTrav [String]
hasProperty :: String -> String -> FTrav Bool
```

where the last `String` argument is an item identifier. The first action takes a predicate on the item id and the property list as argument and returns the list of item ids for which the declaration satisfies the predicate. The second action returns the list of all properties declared for the item, the third action tests whether the property named as first argument is declared for the item.

The current function definition can be maintained by the monadic actions

```
getFunDef :: FTrav (Maybe FunDef)
setFunDef :: FunDef -> FTrav ()
clrFunDef :: FTrav ()
```

The table of item ids for local identifiers can be maintained by the monadic actions

```
enterItemScope :: FTrav ()
leaveItemScope :: FTrav ()
registerItemId :: String -> String -> FTrav ()
getItemId :: String -> FTrav String
```

The first two actions must be invoked whenever a nested C block is entered or left, respectively.

The variable counters can be maintained by the monadic action

```
resetVarCounters :: FTrav ()
resetValCounter :: FTrav ()
getValCounter :: FTrav Int
getCmpCounter :: FTrav Int
```

The first counter is used for value variables, the second for component variables (see Section 3.6.10). The first action resets both counters to 0, the second action resets only the value variable counter. The other two actions return the current counter value and increment the counter.

3.6.15 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. This struct or union type must have a declared tag, otherwise the invocation is ignored and not inserted into the call graph.

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 3.6.14.

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. The corresponding execution function is

```
runCTrav :: CallGraph -> DefTable -> (String,(Bool,[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 classes `FileNameTrav` and `TypeNamesTrav`, so the own source file name can be accessed by `getFileName` and external type names can be tested for being directly used by `stopResolvTypeName` (see Section 3.6.14). The corresponding information is passed as third argument to `runCTrav`.

3.7 Postprocessing Cogent Code

The Cogent code generated for C expressions and statements in the first translation phase as described in Section 3.6.10 and 3.6.12 is in general neither correct nor efficient. Therefore it must be improved, which is done by postprocessing. The postprocessing is done directly in the Cogent AST.

This approach taken by Gencot has several advantages over generating the Cogent code in a single phase. First, the actual translation step ist rather simple and straightforward. Second, the postprocessing is done on a restricted subset of a purely functional language where there is no difference between statements and expressions, so it tends to be simpler. Third, it can be separated into arbitrary many different processing steps which can be freely combined, since they all process the same data structures (the Cogent AST). The drawback is that the code generation is not very efficient, because it first builds a quite voluminous code which is then simplified by the postprocessing. However, the quality of the resulting code has been considered more important than the performance of the Gencot translation.

In the following sections the postprocessing steps are described independently of each other and the last section describes how they are combined.

Every postprocessing step corresponds to a transformation from a Cogent expression to a Cogent expression. All required information is already present in the processed expression, in particular, the Cogent types which have been added as described in Section 3.6.11. The C symbol table and other global state information is not used. However, a postprocessing step may detect error situations. In this case usually a dummy expression is inserted into the Cogent AST, as described in Section 2.10.4, and an error message is registered for

display. For the latter the language-c `Trav` monad is used, as described in Section 3.6.14. Since no other global information is required, the monad is always used with an empty user state and an empty symbol table. The type `ETrav` is defined in module `Gencot.Cogent.Post.Util` for monads of this kind.

All postprocessing steps are defined in submodules of `Gencot.Cogent.Post`, they are either implemented by a monadic action of the form

```
Xproc :: GenExpr -> ETrav GenExpr
```

or, if no error messages are generated, by a Haskell function of the form

```
Xproc :: GenExpr -> GenExpr
```

Postprocessing is applied to all expressions which occur in the generated Cogent program. These are function body expressions, the expressions in constant definitions, and the expressions in array type size specifications.

3.7.1 Restricted Form of the Cogent AST

The Cogent AST generated by the translation described in Sections 3.6.10 and 3.6.12 is highly restricted: it only uses some of the constructs and uses them in specific restricted form. The postprocessing implementation exploits these restrictions and most postprocessing steps preserve them for being exploited by following steps.

The Cogent AST types are defined in the Cogent implementation in module `Cogent.Surface`. Syntactic expressions are defined by the datatype `Expr`. Gencot only uses the following expression variants: `Unitel` for the unit expression, `IntLit, BoolLit, CharLit, StringLit` for literals, `Var` for variable references, `Tuple` for tuple expressions, `PrimOp, App` for applications of operators and functions, `Upcast` for application of the `upcast` operator, `Match, If` for conditional expressions, `Let` for expressions with variable bindings, `Lam` for lambda expressions, `TLApp` for specifying instances of polymorphic functions, `Member` for record member accesses, `Put, ArrayPut` for record and array put operations, and `UnboxedRecord` for unboxed record structures.

The variants `BoolLit, Upcast, Match, Member` are only inserted during postprocessing. The variant `TLApp` is only used for specifying instances of the Gencot operations which are usually polymorphic. The variants `UnboxedRecord` and `Lam` are only used for specifying the arguments of the `repeat` operation in translations of loops.

Bindings in `Let` expressions mainly consist of a pattern defined by the datatype `IrrefutablePattern` and the expression bound to the pattern. Gencot only uses the following pattern variants: `PUnitel` for the unit pattern, `PVar` for a variable, `PTuple` for a tuple pattern, `PTake` and `PArrayTake` for `take` operation patterns, and `PUnderscore` for the wildcard pattern.

Additional restrictions are the following. Here a "variable tuple" means either a `Tuple` expression where all sub expressions are `Var` expressions or a single `Var` expression. A "control tuple" means either a `Tuple` expression where the first sub expression is an `IntLit` expression and the other sub expressions are `Var` expressions or a single `IntLit` expression. A "variable pattern" means a `PVar`, `PUnitel`, or `PUnderscore` pattern or a `PTuple` pattern with components of these variants. A "take pattern" means a `PTake` or `PArrayTake`.

All patterns are either variable patterns or take patterns.

The body of a `Let` expression is either a variable or control tuple or an `If` or `Match` expression. The branch of an `If` or `Match` expression is either a variable or control tuple or a `Let` expression. The body of a `Lam` expression is a `Let` expression or an operator application.

The arguments of a `PrimOp` expression and the argument of an `Upcast` expression are single `Var` expressions. The argument of an `App` expression is a variable tuple. The only exception are applications of the Cogent standard library function `repeat` which is used for translating loops and where the argument is an `UnboxedRecord`.

In a `Put` or `ArrayPut` expression the container expression is a `Var` expression and the list of put specifications contains a single element where the put value is specified by a `Var` expression. In an `ArrayPut` the put index is also specified by a `Var` expression. In a take pattern the list of take specifications contains a single element where the taken value is specified by a `PVar` pattern. In an `ArrayTake` the take index is specified by a `Var` expression.

Together, the structure of arbitrary deep syntax trees only consists of `Let`, `If`, and `Match` expressions where all leaf expressions are variable or control tuples. All other expression variants only occur as bound expression in a binding and have as subexpressions only variable and control tuples with the following exceptions:

- in an `App` expression the function subexpression may be a `TLApp` expression and the argument subexpression may be an `UnboxedRecord` expression,
- in an `UnboxedRecord` expression the subexpressions may be arbitrary expressions,
- in a `Lam` expressions the body subexpression may be a `Let` or `PrimOp` expression.

In all these exception cases the type of the expression is fully determined by the type of the subexpression, so no type clashes may occur between the subexpressions and their context. Expressions for unboxed records are only used in the translation of loops as argument for the `repeat` operation. The type of the record is always determined from the types of the specified component values, the type of the `repeat` instance is determined from this record type.

For simplicity, Gencot strictly avoids introducing additional value variables or other variables during postprocessing.

3.7.2 Evaluating Constant Expressions

In several cases the original translation phase or postprocessing steps result in constant expressions built from predefined operators. Such expressions can be statically evaluated and the resulting constant then may enable other postprocessing steps. Therefore a constant expression evaluation is defined as auxiliary function for other postprocessing steps. It never detects and signals an error. Therefore it is implemented by the function

```
evalproc :: GenExpr -> GenExpr
```

defined in module `Gencot.Cogent.Post.Simplify`.

The `evalproc` step is not applied on its own to arbitrary expressions. The reason is that a constant expression may be intentionally present in the C program to show how a value is calculated. In that case the translation should also result in a constant expression in Cogent. Only if the evaluation is useful for other postprocessing steps it is applied.

The `evalproc` step only processes operator expressions. For them it recurses into the arguments. If all arguments are constants it evaluates the operator and replaces the expression by the resulting constant. All other forms of expressions are left unmodified, in particular, `evalproc` does not recurse into subexpressions which are not operator applications.

3.7.3 Simplifying Operator Application

If an expression cannot be completely evaluated statically, there are cases where it can be simplified. The following cases are implemented by Gencot postprocessing.

If the first argument of a boolean operation evaluates to a constant the operation can be simplified according to the rules

```
True  || e --> True
False || e --> e
True  && e --> e
False && e --> False
```

Currently only the first argument is treated this way, because only that case occurs in actual examples of translation and postprocessing, mainly for the translation of `switch` statements.

Simplifying operator expressions using these rules never detects and signals an error, therefore is implemented by the function

```
opproc :: GenExpr -> GenExpr
```

defined in module `Gencot.Cogent.Post.Simplify`.

3.7.4 Simplifying `let`-Expressions

One of the simplest and most straightforward postprocessing steps is substitution of bound variables by the expression bound to them. In Cogent variables can be bound by `let` expressions and by `match` and `lambda` expressions. Variables bound in `match` and `lambda` expressions can usually only be substituted in special cases, therefore this processing step only substitutes variables bound in `let` expressions.

The basic transformation is for an expression

```
let v = expr1 in expr2
```

to replace it by `expr2` where every free occurrence of `v` is substituted by `expr1`. This is only possible if after the substitution all free variables in `expr1` are still free in the resulting expression, i.e., they are not "drawn under a binding" in `expr2`. This could be avoided by consistent renaming of variables bound in `expr2`. Gencot never renames variables and does not substitute in this case.

The Gencot translation may produce code where the types of `v` and `expr1` differ. Then the type differences are resolved by postprocessing, as described in Section 3.7.6. If the substitution would be performed before, it may be more difficult to resolve the difference, because the variable and its type have been eliminated and the type difference may now occur between `expr1` and its context in `expr2` at several places. Therefore Gencot performs the substitution only if the types of `v` and `expr1` are the same.

This scheme can directly be extended to expressions of the form `let v1 = e1 and ... vn = en in e` using the equivalence to an expression of the form `let v1 = e1 in let ... in let vn = en in e`.

As of February 2022, Cogent does not support closures for lambda expression. This means that a lambda expression must not contain free variables, therefore lambda expressions in `expr2` are never inspected for substituting.

In a Cogent `let` expression instead of a variable `v` an (irrefutable) pattern `p` can be used:

```
let p = expr1 in expr2
```

An irrefutable pattern is a variable or wildcard or it is a pattern for a tuple, record, array or the unit value were the components are again irrefutable patterns. In other words, it is a complex structure of variables which is bound to an expression `expr1` of a type for which the values have a corresponding structure. Every variable may occur only once in a pattern.

Currently, the translation phase only creates bindings with patterns which are either a single variable, or a flat tuple pattern where all components are variables, or a take pattern where all sub patterns are variables. The postprocessing does not introduce more complex patterns. However, to make the code more robust, these restrictions are not assumed for binding processing, the processing is always implemented to work for arbitrary patterns.

If the pattern occurs as a whole in `expr2` it can be substituted by `expr1` as described above. If only parts of the pattern occur (such as a single variable) it depends on the structure of `expr1` whether such a part can be substituted. Gencot tries to substitute as much parts as possible and only retains those parts of the pattern for which a substitution is not possible.

If the substitution is successful the `let` expression is replaced by `expr2` which may again be a `let` expression or any other kind of expression. Therefore the simplification may reduce the number of `let` expressions and may replace a `let` expression by an expression of another kind.

Substitution of bound variables may lead to exponentially larger code, which must be avoided. Gencot uses an expression metrics which roughly measures the size of the printed expression in the Cogent surface syntax. A binding is only substituted if the resulting expression is not much larger than the original `let` expression.

Simplifying a `let` expression by substitution can reduce the variables which occur free in it. This is the case if no parts of the pattern `p` occur free in `expr2`, then `expr1` is removed and all variables which only occur free in `expr1` are removed with it.

Simplifying `let` expressions can never detect and signal an error, therefore it is implemented by the function

```
letproc :: GenExpr -> GenExpr
```

defined in module `Gencot.Cogent.Post.Simplify`.

Processing Subexpressions

When an expression `let p = expr1 in expr2` is simplified, first the subexpressions `expr1` and `expr2` are simplified by processing all contained `let` expressions. Simplifying `expr1` has the following advantages for the substitution:

- The resulting expression usually is smaller. Then its substitution into `expr2` leads to a lower increase of size and may be allowed whereas substitution of the original `expr1` would not be accepted.
- Simplifying may reduce the free variables so that it may be possible to substitute it in more places than the original expression without drawing free variables under a binding.
- If `expr1` is again a `let` expression the pattern can only be substituted as a whole. After simplification it may have a form which corresponds more with the pattern so that also parts of the pattern can be substituted.

Simplifying `expr2` has the following advantages for the substitution:

- Simplifying may reduce the free variables so that there are fewer places for substituting the pattern or parts of it. This may allow substitutions of patterns which where not possible in the original `expr2`. It may also allow substitutions which would have lead to a too large growth of the original `expr2`.

Since an expression `let p1 = expr1 and p2 = expr2 in expr3` is equivalent to `let p1 = expr1 in (let p2 = expr2 in expr3)` this means that a sequence of bindings connected by `and` in a `let` expression is processed from its end backwards.

Pattern Substitution

If (after its simplification) `expr1` has the same structure as the pattern `p` the binding could be split and the parts could be substituted independently. This would correspond to the transformation of the binding `p = expr1` to the sequence

```
p1 = expr11 and ... pn = expr1n
```

where the `pi` are the subpatterns of `p` and the `expr1i` are the corresponding subexpressions of `expr1`. Note that the variables in the `pi` are pairwise disjunct since every variable may occur only once in `p`. Then the sequence could be processed from its end, as described above.

However, the transformation is only correct, if no variable in `pi` occurs free in an expression `expr1j` with `j > i`, otherwise the transformation would draw it under the binding `pi = expr1i`. It could be tried to sort the bindings to minimize this problem but in general it cannot be avoided. Additionally, there may be cases where `expr1` even after its simplification has no structure corresponding with that of `p`, which also prevents the transformation.

For this reason, instead of transforming the binding and substituting it sequentially, Gencot deconstructs the pattern while searching for matches. Whenever it searches the binding `p = expr1` in an expression (starting with `expr2`) it determines the variables occurring free in the expression and "reduces" the binding to these. Reducing a binding to a set of variables is done by first replacing in `p` all variables which are not in the set by a wildcard (underscore) pattern. This is always possible. Then it is tried to remove as much wildcard parts from the binding, this is only possible if the corresponding part can be removed from `expr1`.

If `p` is a tuple pattern where some components are wildcards they can be removed if `expr1` is a corresponding tuple expression. If `expr1` is an application of a function to an argument the wildcard parts cannot be removed, since the tuple returned by the function is not available. If `expr1` is a `let` or `if` expression or a `match` expression the binding cannot be reduced or split directly. In these cases Gencot constructs the reduced or split binding by recursively reducing or splitting the subexpressions (the let-body, the if-branches, the match-alternatives).

If no variable in the set occurs in the pattern all variables are replaced by wildcards and the binding is reduced to the empty binding, represented by the "unit binding" `() = ()`.

If after reducing it the binding is not empty, it is matched with the expression, which is successful if it has the same structure with the same variables (which is only possible if the pattern contains no wildcards). If it matches the search stops, if not the binding is searched recursively in all subexpressions (by first reducing it to the subexpression). When the search reaches a single variable, the binding has been reduced to that variable. If the pattern consists of the same variable it matches there.

Otherwise the pattern is empty or it contains the variable together with other parts which means that it cannot be used to substitute the variable at that position in the expression. In this case at least the binding for that variable must be retained. Gencot determines all variables in `expr2` for which that is the case. Then it reduces the original binding `p = expr1` to that set to determine the binding `p' = expr1'` which must be retained in the `let` expression. Only if no such variable exists the binding can be completely removed from the `let` expression.

If `p` contains a `take` pattern for a record or array it can only match a `put` subexpression of exactly the same form in `expr2`. That is only present if the component is taken and put back without modifying it or the remaining record or array. In this case the `put` expression is replaced by the part of `expr1` corresponding to the `take` pattern. Wildcards in a `take` pattern can only be removed if the corresponding part of `expr1` is a `put` expression.

Even if a part of the binding successfully matches in `expr2`, substitution may be prevented because it would draw a free variable in `expr1` under a binding. Such a binding may be the retained part of the original binding or it may be a binding in a `let` subexpression which has been retained during the simplification of `expr2` or it may be a binding in a `match` expression which is not processed by the simplification. These cases are handled by splitting the binding for which matches are searched whenever it is drawn under another binding in a (maximal) part allowed to draw under the binding and a rest which must be retained. The rest is still searched under the binding to determine whether the variables in it

occur at all, if that is not the case it is not required in the expression and is not retained.

A binding is split according to a set of variables not to occur free by first determining all bound variables so that the corresponding part of the bound expression contains a free occurrence of a variable from the set. Then the binding is reduced to this set and to its complement, yielding the binding to retain and the binding for substitution.

The substitution and binding simplification is implemented in two phases. In the first phase the matches for the pattern are searched in `expr2`, resulting in the part to be retained and for every matching part the number of successful matches The matching parts are actually substituted in `expr2` in the second phase. If the retained part is not empty it is prefixed to the result.

Banged Variables

A binding may have "banged" variables, i.e., be of the form

```
p = expr1 !v1 ... !vn
```

which makes the variables `v1,...,vn` readonly in `expr1`. Banged variables are introduced by readonly processing (see **??**). Simplification of `let`-expressions may be executed after readonly processing, therefore it must deal with banged variables.

In Cogent banged variables can only appear at specific places in the code: in a `let` binding, in a `match`-expression, and in the condition of an `if`-expression. Therefore it is in general not possible to substitute (a part of) the pattern `p` at arbitrary occurrences in `expr2` by `expr1` together with the banged variables. Gencot never tries, it retains those parts of the binding in which the banged variables occur free in `expr1`.

The same mechanism for splitting a binding used to prevent drawing variables under a binding is used for banged variables. Whenever a binding is split, the banged variables are added to the set of variables not to occur free.

Growth Restriction

As size metrics for an expression the number of characters appearing in its surface representation is used. It could be determined by actually prettyprinting the expression and measuring the size of the resulting string. However, it is assumed to be more efficient to traverse the expression and calculate the size from the number of characters in the names and literals and in the keywords, special characters and separating blanks needed for constructing composed expressions.

After the first phase the metrics of the `let` expression is calculated. Since for each matching part to be substituted it has been determined in the first phase how often it occurs in `expr2` the metrics for the simplified expression can be calculated and it is known how much each subpattern contributes to its size. If the size is larger than for the original expression and its growth exceeds a fixed limit factor additional binding parts are determined which are retained. Beginning with the binding part with the largest contribution, parts are retained until the growth is below the limit.

Instead of only taking its contribution to the size into account, binding parts could also be selected according to the kind of variables they contain. Gencot

uses different kinds of variables in its generated code (see Section 3.6.10): value variables, component variables, the control and result variables, and variables corresponding to C object names. These could be prioritized as follows: first as many value variables are substituted as possible in a complete expression, then the control variables, then the component variables and finally the C object names. In this way the most "technical" variables are substituted before the more "semantical".

The reference metrics is calculated for the expression after simplifying its subexpressions. This means that the growth limit factor applies to every subexpression simplification step separately. This has two implications. First, a subexpression simplification may strongly reduce the size of the subexpression and that may also reduce the size of the `let` expression, which becomes the reference for its own simplification. Thus it is not possible to tolerate a larger growth after strongly reducing the size for subexpressions. Alternatively, the reference size could be measured before simplifying the subexpressions. In the code generated by Gencot there are typically large nestings of `let` expressions with unnecessary "chain bindings". It is assumed that it does not yield good results when these unnecessary large expressions are used as reference, therefore Gencot uses the first approach.

Second, in the worst case each simplification step grows the expression by the limit factor which still results in an overall exponential growth relative to the number of subexpressions. Therefore the growth limit factor should not be much larger than 1. The effect of this factor and a good selection for it must be determined by practical tests. Alternatively the factor could be specified as an input parameter for Gencot so that it can be selected specifically for every translated C program.

Variables of Linear Type

If a variable of linear type occurs free in `expr1` and `expr1` is substituted more than once in `expr2` this usually results in a "double use" of the variable, which is not allowed in Cogent. Therefore Gencot tests for this situation and retains the (parts of) the binding which would cause a double use error when substituted.

More specifically, a double use error only results if the expression is substituted more than once in the same alternative execution branch, i.e., in the same branch of an `if` expression or the same alternative of a `match` expression. When Gencot counts matches of (sub)patterns of `p` in `expr2` it also counts the maximal number of matches in an alternative execution branch. It then uses this count to decide whether the corresponding part of the binding must be retained to avoid double use errors.

Pre-Simplifying Unused Variables

If in an expression

```
let p = expr1 in expr2
```

a variable bound in `p` does not occur free in `expr2` it can be removed from the binding `p = expr1` as described above, without substituting it in `expr2`. This case is much simpler than the general case described above in several aspects:

- it is not necessary to search for matchings in `expr2`,

- it is not possible to draw free variables in `expr1` under a binding,
- the expression will never grow by the simplification,
- type differences between `p` and `expr1` do not matter.

The Gencot translation produces many such cases of unused variables, because it translates expressions without regarding whether their values are used in the context. For example, the value of an assignment expression is normally not used in the context. On the other hand unused variables may prevent resolving a type difference by banging, if the variable has non-escapeable type (possibly caused by the banging itself) and leaves the scope of the banging. Therefore it is crucial to remove bindings of unused variables early during postprocessing, in particular, before trying to apply banging, whereas normal simplification of `let` expression is only done after resolving type differences. Early removal of unused variables is called pre-simplification here.

An exception is made for `put` operations. Every `put` binding is always generated together with a preceding `take` binding and both are usually also processed together (see Section 3.7.15). If only the component variable of a `take` binding is used but not the container variable, pre-simplification would remove the `put` binding and would preserve the `take` binding. Since the `put` bindings do not affect the banging possibilities, they are excluded from pre-simplification. Since every `put` binding uses both the container and component variable of the corresponding `take` binding, that will always be preserved as well.

Pre-simplification is facilitated by implementing the removal of unused variables in a separate postprocessing step

```
presimp :: GenExpr -> GenExpr
```

also defined in module `Gencot.Cogent.Post.Simplify`. Thus it can be applied independently from `letproc`. Note, that `letproc` will still remove variables which have become unused by other postprocessing steps after applying `presimp`.

3.7.5 Simplifying If-Expressions

The most straightforward simplification of a conditional expression is replacing it by one of the branches, if the condition can be statically evaluated. Gencot tries this, by recursing into the condition and then using `evalproc` (see Section 3.7.2) for evaluating the condition, before it recurses into the branches. This avoids processing a branch which is removed afterwards. If the condition cannot be statically evaluated, both branches are processed recursively.

Other Used Transformations

Afterwards, the following additional simplification rules are applied, if possible.

- If both branches are the same expression, the conditional expression is replaced by this expression.
- If both branches statically evaluate to a boolean constant, the conditional expression is replaced by the condition or the negated condition according to the rules

```
if c then True else False --> c
if c then False else True --> not c
```

- The condition is substituted by `True` in the `then` branch and by `False` in the `else` branch. This may enable additional simplifications in subsequent iterations (see Section 3.7.16).

The following rule is applied to operator expressions where the first argument is a conditional expression and the second argument can be statically evaluated:

```
(if c then e1 else e2) <op> e -->
if c then e1 <op> e else e2 <op> e
```

This transformation may enable the static evaluation of the resulting branches.

All these rules have been selected, because they specifically apply to conditional expressions resulting from the Gencot translation and postprocessing, in particular for the control variable.

Simplifying `if` expressions using these rules never detects and signals errors, therefore it is implemented by the function

```
ifproc :: GenExpr -> GenExpr
```

defined in module `Gencot.Cogent.Post.Simplify`.

Unused Transformations

Other rules would transform conditional expressions where one branch can be statically evaluated to a boolean value according to

```
if c then True else e  --> c || e
if c then False else e --> not c && e
if c then e else True  --> not c || e
if c then e else False --> c && e
```

This is not used, because it removes boolean constants which may be useful for static evaluation after other processing steps.

Other rules for simplifying conditionals with boolean values are

```
if c then e else not e --> c == e
if c then not e else e --> c /= e
```

These are not used because they turn conditional expressions into equations, which currently are not processed any further.

If a branch is again a conditional expression, the following transformation is possible:

```
if c then (if c' then e1 else e2) else e -->
if c' then (if c then e1 else e)
       else (if c then e2 else e)
```

It is not used because it duplicates expression `e` and does not reduce the structure of conditional expressions, so there is the danger of cyclic transformation.

Another rule would split conditional tuples into a tuple of conditionals according to

```
if c then (t1,..,tn) else (e1,..,en) -->
(if c then t1 else e1, .., if c then tn else en)
```

which would allow further simplification for all components with `ti = ei` and it could allow additional substitutions by `letproc`, because the components can be substituted or omitted separately. However, if both are not applicable, it tends to enlarge the expression by copying the condition `c`. Therefore the effect on `letproc` has been implemented there by the way how conditional expressions are split and the rule is not used here for `ifproc`.

The substitution of the condition in the branches could be generalized by substituting values for variables which can be inferred from the condition, such as in

```
if i == 0 then e1 else e2
```

where it is possible to substitute `i` by `0` in `e1`. Even more general, the condition can be interpreted as a set of equations for its free variables, if it can be solved for some of them they can be substituted by their solution in the first branch. It has not yet been considered whether such substitutions would have an effect that would pay for the additional complexity.

If the condition is itself a conditional expression, it may be possible to derive substitutions, although the condition does not occur as a whole in the branches. As an example, the following transformation could be applied:

```
if (if c then x else y)
   then (if c then (if x then e1 else e2) else e3)
   else e4
-->
if (if c then x else y)
   then (if c then e1 else e3)
   else e4
```

It has not yet been considered, whether such transformations would be useful for cases occurring during translations of C programs.

3.7.6 General Type Difference Processing

Even if the translated C program is type-correct, the Gencot translation may result in type incompatibilities in the generated Cogent code. These originate from three possible sources:

- Type differences in the C program which are automatically resolved by the C compiler by conversions.
- Application of item properties which affect the item's type during the Gencot translation.
- Type bool and arithmetic types, which are different in Cogent but not in C.

The first case only occurs for C operators. They are conceptually polymorphic, the actual function is determined by the type of the argument(s). For binary operators the argument types may be different, then in some cases they

are adapted to a common type which then determines the function of the operator. The same holds for the ternary conditional operator for the two branches. Operators in C are either arithmetic, relational or comparison operators or the conditional operator. The first two kinds are only applied to numeric arguments. Since Gencot does not support floating point types only integer arguments are relevant here. The comparison operators `==` and `!=` can also be applied to pointer arguments and the branches of a conditional operator can be of arbitrary type. In some cases the type differences can also be resolved in Cogent by applying conversion functions, in other cases Gencot does not support a translation and signals an error.

The second case may occur for the item properties Read-Only, Not-Null, and No-String. All three only affect items of pointer types. Since they may be specified manually, differences may be errors which should be detected and signaled by Gencot. In some cases it is also possible to resolve differences by inserting conversions during postprocessing.

The third case may occur whenever an expression or context is recognized as boolean, which is the case for the result of comparison operators and for the condition in conditional expressions and statements.

Gencot preserves the restricted form of the Cogent AST described in Section 3.7.1 until all these type clashes have been resolved. Therefore type clashes need only be detected and handled in the restricted form of expressions. This form mainly consists of a tree structure built from `Let`, `If`, and `Match` expressions where all leaf expressions are variables, possibly wrapped in tuples. In these cases (except for conditions in `If`-expressions and branches in `If` and `Match` expressions) and also for the exceptions listed in Section 3.7.1 the type of an expression is always determined from the type(s) of the subexpressions, therefore no type clashes can occur.

The remaining cases where type clashes may occur are conditions in `If`, `Match`, and bindings in `Let`:

1. For a branch in an `If` or `Match` the type may be different from the expected type which may be determined by both branches together.

2. For a condition subexpression the type may be different from the expected type `Bool`.

3. For the bound expression in a binding the type of a subexpression may be different from the expected type. The following cases for subexpressions exist:

   - argument of an operator (list of variables) or function application (tuple of variables), then the expected type for each variable is determined by the operator or by a formal argument type of the function,
   - container variable in a `Put` or `ArrayPut`, then the expected type is a writable not-null container with the specified component,
   - index in an `ArrayPut`, then the expected type is a numeric type,
   - condition in an `If` expression, as above.

   If the subexpression is a branch in an `If` or `Match` expression the bound expression is treated separately as in case 1 above. In all other cases of subexpressions (container of a `Member`, function in an `App`, body in a `Lam`

or `Let`) no type clashes can occur. An expression for a component value in a `put` operation is always the corresponding component variable for which the type has been determined by the component type, so no type clash can occur here.

4. For a pattern in a binding the type of a subpattern or subexpression may be different from the expected type. The following cases exist:

   - container variable in a `Take` or `ArrayTake`, then the expected type is a writable not-null container with the specified component,
   - index in an `ArrayTake`, then the expected type is a numeric type.

   All other cases of patterns with subpatterns are tuple patterns, for them the type is always determined by the type of the subpatterns, so type clashes cannot occur.

5. For a binding the type of the pattern may be different from the type of the bound expression. This may only happen for a variable pattern or a variable in a tuple pattern which is no value variable. The type of value variable, unit and wildcard patterns is always determined by the (corresponding component of) the bound expression. For a take pattern the bound expression is always a value variable bound to the container variable used in the pattern, so no type clash can occur.

In the cases 2-5 either the subexpression (cases 2-4) or the context (case 5) consists of a single variable. If it is possible to resolve the type clash by applying a conversion function to the value, this application can be added as a separate binding of the form

```
v = conv v
```

where `v` is the variable involved in the type clash and `conv` is the conversion function. The `v` on the right has its original type, the `v` on the left has the expected type. Adding the conversion in this form preserves the restricted form of the Cogent AST. This would not be the case if conversion function applications would be directly wrapped around the occurrence of the variable as subexpression in its context. This technique of inserting additional bindings is used by all postprocessing steps which resolve type clashes by value conversion.

In cases 2-4 above the binding must be inserted so that the conflicting occurrence of the variable as subexpression is in the scope of the binding, but no other occurrences. Conditions, operator or function arguments, and array indexes always origin from C subexpressions and are bound by the translation to unique value variables for each occurrence, so the variable occurs only once. Type clashes for container variables in take/put operations are never resolved by application of a conversion function. Therefore, in cases 4 and 4 the conversion binding(s) can simply be inserted before the binding containing the type clash(es). In case 2, according to the restricted form of the Cogent AST, the `If` expression can only occur as a `Let` body. Then the conversion binding(s) can be inserted at the end of the binding list of this `Let` expression, immediately before the body. In all these cases the type of the subexpression (and possibly its surrounding tuple) is changed to the expected type, thus the type clash is resolved.

In case 5 the variable in the conversion binding is that which occurs in the pattern involved in the type clash. Then the conversion binding is inserted after the binding containing the type clash. The type of the variable in the pattern and possibly the type of the surrounding tuple is changed to the type of the (component of the) bound expression, thus the type clash is resolved.

In case 1 the branch may be a variable tuple or a `Let` expression. In the first case the type clash is caused by one or more variables in the tuple and can be resolved by according conversion bindings. These bindings are inserted by replacing the branch by a `Let` expression with the conversion bindings and the original branch as body, which preserves the restricted form of the Cogent AST. In the second case the type clash is caused by the `Let` body. Due to the restricted Cogent AST it is either a variable or control tuple, then the type clash can be resolved by inserting conversion bindings at the end of the binding list immediately before the body, or it is an `If` or `Match` expression, then the clash is caused by both branches and can be resolved by recursively resolving it for both branches.

Whenever Gencot cannot resolve a type difference it signals an error and additionally replaces sub expressions with the wrong type by dummy expressions which specify the error reason and have the correct expected type according to the context of the expression. Thus, after processing them no type clashes remain in the resulting Cogent code, even if there where errors.

Resolving type clashes by applying a conversion function or signalling an error is implemented in a generic way by the monadic action (because it may signal errors)

```
resolveExpr :: ConvVarFun -> GenExpr -> ETrav GenExpr
```

defined in module `Gencot.Cogent.Post.MatchTypes`. Here the first argument is a function which specifies conversion for some type(s). Its type is defined as

```
type ConvVarFun = GenExpr -> GenType -> ETrav (Maybe GenBnd)
```

It takes an expression `e` and a type `t` as arguments. If `e` is a single variable `v` and the function supports converting its type to `t` by applying a conversion function `conv` it returns a binding

```
e' = conv e
```

where `e'` is the same variable but with type `t`. If the function determines that `e` cannot be converted to `t` it returns a binding

```
e' = gencotDummy "..."
```

where the string argument specifies the reason for the error and additionally it signals an error. In all other cases the function returns `Nothing`. The function `resolveExpr` resolves all type clashes which are handled by the `ConvVarFun` argument as described above by inserting the returned binding and modifying the type of the subexpression or context. In all other cases it leaves the type clash untouched. Therefore `resolveExpr` may be invoked several times with different `ConvVarFun` arguments, each resolving clashes for some specific types.

3.7.7 Detecting Readonly Modifications

Applying the Read-Only item property changes the item's Cogent type `t` to the readonly type `t!`. The main incompatibility for values of readonly types occurs when they are modified. Modifications can only be applied to container values, by replacing or setting a component.

As described in Section 3.6.10 container modifications are translated to a pair of `take`- and `put`-bindings with a re-binding of the component variable to a new value in between. In case of a nested component there are several `take`- and `put`-bindings surrounding the re-binding.

If the container has readonly type every modification of a component is an error, independent of the component type. Gencot detects such modifications as follows: for every `take`-binding with readonly container type all bindings of the component variable in their scope which are not `put`-bindings are modifications. A `put`-binding without a re-binding of the component variable puts back the original value and does not cause a modification.

Since it is not possible to convert a readonly type back to a modifyable type these modifications cannot be corrected by applying a conversion. Hence Gencot always signals an error message and also replaces the component variable in the erroneous binding by the error variable `err'`. This will remove all such modifications, the resulting code never modifies a readonly container. Since `err'` is never referenced the binding will be eliminated when unused variables are eliminated in later postprocessing steps.

Postprocessing for detecting and removing modifications of readonly containers is implemented by the monadic action

```
romodproc :: GenExpr -> ETrav GenExpr
```

defined in module `Gencot.Cogent.Post.MatchTypes`.

3.7.8 Readonly Processing

Another incompatibility for readonly types are type clashes with the corresponding linear type. Since values of linear type can be converted to values of readonly type by applying the "bang" operation, such type clashes may be resolved by conversion so that both types are readonly.

However, the application of the bang operator in Cogent is strongly restricted. It cannot be applied to arbitrary expressions but only to single variables. It can only be applied at specific positions in the code (at bindings, at conditions in conditional expressions, and at match expressions). The effect of the bang operator is restricted to a scope determined by the position of application. No value with escape-restricted type (containing a read-only part) may leave this scope. Due to these restrictions a systematic construction of bang applications so that a maximum of readonly type clashes are resolved, is a complex optimization problem.

Determine Variables for Banging to Resolve Type Clashes

Gencot uses a specific heuristics to find bang applications which resolve readonly type clashes. Remaining clashes are then resolved by dummy expressions and signaling them as errors. In such cases it may be possible to remove readonly type clashes manually with the help of the Read-Only item property.

The heuristics only bangs variables which correspond to translated C variables and parameters. If a component variable would be banged, the scope of the banging would either include the putback binding, which would cause a type clash with the type of the container component, or it would exclude it, then the value has to leave the scope because it is required for the putback binding, but that is not possible, since by banging it its type has become non-escapeable. If a value variable would be banged this makes only sense if its single use is in the scope of the banging. But then the value outside the scope is not used, it is discarded which is not allowed for a value of linear type in Cogent. If the result variable would be banged, it must always leave the scope of the banging, since its value must be returned by the function body, again, that is not possible because its type is non-escapeable by the banging. The other variable kinds (control, switch, and case variables) cannot have a linear type, therefore banging is not applicable to them.

When Gencot searches for readonly type clashes which may be resolved by banging, it maintains a map of "source variables" for value and component variables. These are the C variables which, if banged, change the type of the value variable or component variable to readonly. For a component variable the source variables are those of the container. Usually, a variable has only one source variable, but for a variable bound to a conditional expression there may be different source variables in the branches, so in general every variable has a set of source variables. If the set is empty the variable's type cannot be made readonly by banging variables, this is the case for example for a variable bound to the result of a function application.

The scope of a bang application is always an expression. In the case of a conditional, it is the condition expression, in the case of a binding it is the bound expression.

When Gencot processes an expression as possible bang scope it searches all contained readonly type clashes. If the subexpression at the type clash is of linear type and has source variables, so that its type can be converted to readonly by banging the source variables they are collected. In all other cases the type clash is signaled as an error and is resolved by replacing the sub expression by a dummy expression with the same type as for the context. If the set of collected source variables is empty all readonly type clashes have been resolved without the need to bang variables.

Otherwise in a next step the scope is traversed and for all expressions and patterns which depend on the collected source variables the type is changed to readonly. This may cause new inconsistencies: either modifications of readonly containers or readonly type clashes. The former are detected and treated as error, as described above, the latter are resolved by either collecting additional source variables for banging or by treating them as error. This step is repeated as long as it collects additional variables for banging. Since the scope only uses a finite number of variables the repetition terminates. If the collected source variables are banged and the scope is replaced by the processed scope and this scope has an escapeable type all readonly type clashes in it have been resolved.

Postprocessing for resolving readonly type clashes by banging variables is implemented by the monadic action

```
bangproc :: GenExpr -> ETrav GenExpr
```

defined in module `Gencot.Cogent.Post.MatchTypes`.

Determine Variables for Additional Banging

Values of readonly type can be freely copied and discarded in Cogent, the can be used in much more flexibel ways than values of linear type. Therefore Gencot tries to bang variables also if it is not required to resolve type clashes.

The most relevant kind of uses where banging provides an advantage is for a container when components are accessed. Therefore Gencot also applies the banging heuristics to all variables which occur as container in a `take` operation in the scope of the banging. As described above, a variable is only banged if it does not cause a readonly type clash. Therefore banging additional variables is most effective if all readonly type clashes have been resolved, so that detected type clashes are always caused by the banging and are not due to other reasons.

This is implemented by the monadic action

```
ebangproc :: GenExpr -> ETrav GenExpr
```

defined in module `Gencot.Cogent.Post.MatchTypes`.

Selecting Bang Positions and Scopes

As described in Section 3.6.10 Gencot translates C expressions in a way that they are either trivial and cannot contain a readonly type clash, or they contain atleast one binding and thus a bang position. The heuristics tentatively tries bang positions in bindings and conditions. Match expressions are not tried, they are always generated by NULL pointer processing (see Section 3.7.10) in a way that banging the match will make its result non-escapeable.

The heuristics starts with the outermost bang positions, i.e. those with the largest bang scopes. If it is successful without causing errors it uses the bang position by banging all variables in the collected set (which may be empty) and does not try inner bang positions in the scope. If it caused errors but there are no inner bang positions in the scope it uses the bang position with the processed scope and signals the corresponding errors. If there are inner bang positions the bang position is not used at all, discarding all errors caused by the tentative processing, and processing recurses into the scope expression. In this way the most effective bang positions (with the largest scope) are tried first.

A bang position may fail because the banged scope has a non-escapeable type. A condition in a conditional expression always has type `Bool` which is escapeable, hence a non-escapeable type can only result for a bang position in a binding. There it is treated as follows. First, the scope result is reduced as much as possible by removing unused parts from it. This is done by performing the pre-simplification step `presimp` described in Section 3.7.4 and the step `romodproc` described above before processing readonly type clashes. Moreover, containers which are banged in the scope are also removed, because although they may be used afterwards, they cannot be modified in the scope (otherwise the banging would cause an error), therefore their value before the scope is still valid.

Second, if the result of a single binding prevents banging because it is not escapeable, it may be possible to bang the binding together with subsequent bindings where the result of the first binding is used. Then the bindings form a common bang scope and the result of the first binding is only used inside this scope and need not leave it.

A sequence of bindings can be converted to a common binding scope by converting it to a single binding in a similar way as by the function `cmbBinds` described in Section 3.6.10. A binding sequence `b1, ..., bn` is converted to the binding

```
(v1,...,vm) =
let b1 and ... and bn
in (v1,...,vm)
```

where the variables `v1,...,vm` are all variables bound in `b1, ..., bn` and used after the binding sequence, either in subsequent bindings or in the body of the surrounding `let` expression.

When the heuristics processes a binding sequence it tries the scopes resulting from combined bindings in a similar way as it tries scopes in general: it starts with the largest scope which results from converting the whole sequence to a single binding. If not successful it removes bindings from the end of the sequence and tries the resulting smaller scopes until it reaches the single first binding in the sequence. If that also causes errors it is used as it is, then the rest of the sequence is processed in the same way. In this way all possible subsequences are tried as bang scopes.

MayNull Operations

The Gencot operations for `MayNull` types defined in Section 2.7.10 must be treated in a specific way since they are available both for linear and readonly types. Therefore, a readonly type clash for an argument or result of such an operation can be resolved by replacing the operation. The relevant operations are `mayNull`, `notNull`, and `null` with their readonly alternatives `roMayNull`, `roNotNull`, and `roNull`. For the first two, either the argument and result are both linear, or both readonly. Applications are generated by the NULL pointer processing (see Section 3.7.10), it always inserts the linear version, independent of the argument type. The third is only generated after readonly processing, it is still present in the form of the value `cogent_NULL`.

Occurrences of `NULL` in C are treated as follows. They are syntactically translated by Gencot to a reference to a variable `cogent_NULL` of type `MayNull CVoidPtr`. Instead of banging this variable, the type of every single occurrence may be converted to readonly type, if necessary. This is implemented by temporarily renaming every occurrence of `cogent_NULL` to `cogent<n>_NULL` where `<n>` is a unique number. The heuristics then adds all instances with a readonly type clash to the source variables collected for banging as usual. Thus their types and that of all variables with them as source are converted to readonly. For the actual banging these variables are omitted.

Applications of `mayNull` may occur at arbitrary places in the generated code, the result type is readonly whenever the argument type is readonly. Thus, if the argument is affected by banging variables, it transfers to the result. This is taken into account by transferring the source variables from argument to result. Then readonly type clashes between a linear result and its context can be resolved as usual. If the result is readonly and the context is linear, the corresponding type clash existed between the argument and the context before `mayNull` has been inserted, thus the clash is resolved as usual by replacing the `mayNull` application by a dummy and signaling an error. If

Applications of `notNull` occur only in a match expression in the form

```
notNull v
| Nothing -> e1
| Some v -> e2
```

where `v` is a C variable or component variable.

Here, whenever the type of `v` is readonly, the result of `notNull` and the two alternative patterns have readonly type, thus there cannot be a readonly type clash for the result of `notNull` which could cause `v` to be banged. Therefore the source variables of `v` need not be transferred to the result of `notNull`.

If the source variables of `v` are decided to be banged, the type of the result of `notNull` and the two patterns must be converted to readonly. The type for the occurrences of `v` in `e1` and `e1` will be converted as usual.

In all cases only the types of `mayNull` and `notNull` are converted, the operations are only replaced by their readonly alternatives in a later processing step.

3.7.9 Boolean Value Processing

The type `Bool` is used in the Cogent AST although it does not result from translating a C type. As described in Section 3.6.11 it is specified by the translation for all results of applications of equational, relational, and boolean operators. Moreover, it is implicitly expected for the arguments of boolean operators and for conditions in conditional expressions. The use of type `Bool` can clash with all types which result from translating a C type which can be used to represent a boolean value. These are all "scalar" types in C, which are the arithmetic types and the pointer types.

Arithmetic types are always translated to the Cogent integer types `U8, U16, U32,` and `U64`. Type clashes between them and `Bool` can always be resolved by conversion. An expression `e` of integer type can be converted to `Bool` by

```
e /= 0
```

because the type of the literal `0` is automatically adapted to the type of `e` here.

An expression `e` of type `Bool` can be converted to any arithmetic type by

```
if e then 1 else 0
```

because the literals `1` and `0` are automatically adapted to the expected arithmetic type.

Pointer types are translated to linear types and are represented by boxed record types or boxed abstract types, possibly wrapped by `MayNull`. An expression `e` of type `MayNull a` can be converted to `Bool` by comparing it with NULL:

```
e /= cogent_NULL
```

Here `cogent_NULL` denotes the Gencot translation of the C constant `NULL` of Cogent type `MayNull CVoidPtr`. This Cogent expression resulting from the conversion is the same as the translation of the C expression `expr != NULL`, if `e` is the translation of `expr`. Clashes between the type of `e` and `MayNull CVoidPtr` will be resolved by further postprocessing for pointer types as described below.

It would be possible to convert the value `False` to a NULL pointer using the Gencot basic operations `null` and `roNull` defined in Section 2.7.10. However, for the value `True` no sensible conversion exists. More general, conversion from a numerical value to a pointer would correspond to a form of pointer arithmetics which is deliberately not supported by Gencot. Therefore Gencot never converts from type `Bool` to a pointer type.

Type clashes with `Bool` are resolved as follows. A sub-expression with non-boolean type in a context expecting type `Bool` is converted by comparing with `0` or `cogent_NULL`. A sub-expression with type `Bool` in a context not expecting type `Bool` is always converted to arithmetic type `U32` as described above. If necessary, further type clashes are resolved in later postprocessing steps. If the type clash is between two operator arguments or two components of the branches of a conditional expression the conversion is always to type `Bool`, since that is always possible.

Postprocessing for boolean values in this way is implemented using function `resolveExpr` described in Section 3.7.6. Although it cannot cause errors, it is thus implemented by the monadic action

```
boolproc :: GenExpr -> ETrav GenExpr
boolproc = resolveExpr convVarBoolDiffs
```

defined in module `Gencot.Cogent.Post.MatchTypes`, where `convVarBoolDiffs` is the function of type `ConvVarFun` for boolean conversions.

3.7.10 NULL Pointer Processing

Pointer values may be NULL in C. In Cogent this property is modeled in the type using the `MayNull` wrapper type. Type clashes may result between `MayNull` types and not-null (unwrapped) types.

Values of not-null type can be converted to `MayNull` type using the Gencot provided operations `mayNull` and `roMayNull` introduced in Section 2.7.10. Values of `MayNull` type can only be converted to not-null type by testing them for being NULL using the Gencot provided functions `notNull` and `roNotNull` and then modifying the type in the corresponding conditional branch. In the other branch the value can be substituted using the Gencot provided functions `null` and `roNull`.

Values of not-null type are needed when the pointer shall be dereferenced. It would be possible for Gencot to generate the required NULL tests, but then it must generate a working code alternative for the case that the pointer is NULL. It is usually not possible to do this automatically. Therefore Gencot never generates NULL test, it only uses those which are present in the C code.

A NULL test has the form `v op cogent_NULL` or `cogent_NULL op v` where `op` is either `==` or `/=`. It origins either from translating corresponding C code or by resolving a boolean type clash by converting a value of pointer type to `Bool`, as described above. If the variable `v` has not-null type it can be immediately substituted by `True` or `False`. Otherwise the test result can be used to specialize occurrencies of `v` in conditional code which depends on the test result.

In C the test result can be stored and used later as condition to guard code parts. To detect all those code parts would require a full data flow analysis of the test result. Gencot does not perform this analysis, it only uses the test

result if the test is immediately present in the condition of a conditional expression or statement. Conditional expressions and statements are both translated to Cogent conditional expressions. Then the scopes affected by the test are syntactically available as the branches of the conditional expression.

Postprocessing for values of `MayNull` type in this way is implemented by the monadic action

```
maynullproc :: GenExpr -> ETrav GenExpr
```

defined in module `Gencot.Cogent.Post.MatchTypes`.

Splitting Conditions

A NULL test in C may occur as part of a complex boolean expression using the operators `&&`, `||`, and `!`. In this form the test cannot be used for specializing the tested variable, because the complex condition may have no direct consequence for the tested variable. Therefore, in a separate pass, Gencot splits complex conditions so that contained NULL tests are isolated as condition in a conditional subexpression.

As described in Section 3.6.10, the operators `&&`, `||` are translated to a conditional expression. Therefore a complex boolean condition is always translated to an expression using (possibly nested) conditional expressions and the `not` operator. If such an expression occurs as condition in another conditional expression this expression can be split according to the transformation

```
if (if x then x1 else x2) then y1 else y2
-->
if x then (if x1 then y1 else y2)
     else (if x2 then y1 else y2)
```

When the condition results from translating a `&&`, `||` operator, one of `x1` and `x2` is typically `True` or `False`, then the result of the splitting can immediately be simplified by removing one occurrence of `y1` or `y2`. Note however, that even then the splitting will duplicate one of the branches `y1` and `y2` of the original conditional expression.

The `not` operators can be eliminated by swapping the branches according to

```
if (not x) then y1 else y2
-->
if x then y2 else y1
```

Gencot reduces and splits complex conditions according to these rules until the remaining conditions are either a single NULL test or a (possibly complex) condition which does not contain a NULL test.

The duplication of branches caused by the splitting rule may result in an exponential growth of the code size. To mitigate this, Gencot tries to move parts of the branches before the conditional expression, before applying the splitting rule. This is done according to the transformation rule

```
if y then (let b1 in y1) else (let b2 in y2)
-->
let b1 and b2 in (if y then y1 else y2)
```

However, this is not always possible, since the variables bound in `b1` may not occur free in `let b2 in y2` and vice versa and also not in `y`. Gencot extracts as much bindings from the branches as possible. If a branch binds only value variables (which may not occur outside the branch), i.e., it has no side effects, all bindings can be extracted and the branch is reduced to a single variable reference, then the code is minimally enlarged.

Extracting bindings from the branches prevents applying the implications of the NULL test in them, which is the main goal of the NULL processing. Therefore bindings are only extracted from branches which are actually duplicated, because then they cannot be restricted to a single branch of a NULL test condition and its effects cannot be applied to them, at least not to all copies.

Gencot also takes into account, that the condition `y` may have side effects in C. Then it will be translated to a tuple binding

```
(v<n>',v1..) = e
```

where `e` may be a conditional expression or a let expression and the value valiable `v<n>'` is used as condition `y`. Then the splitting rule becomes more complex and must also deal with the side effect parts of `e`.

The code modifications done during condition splitting are rather complex. Banged variables in bindings would make it even more complex, therefore Gencot performs NULL processing before performing readonly processing which may introduce banged variables, as described in Section 3.7.8.

Transforming NULL Tests

After splitting conditions Gencot uses a second pass to transform all conditional expressions with a NULL test as condition to a Cogent match expression according to the transformation rule

```
if v /= cogent_NULL then y1 else y2
-->
notNull v | Nothing -> y2
          | Some v -> y1
```

Note that if the type of `v` is `MayNull T` the result of `notNull` has type `Option T` which can be discriminated by the match expression. In the `Some`-branch the variable `v` has type `T`, so it is known to be not NULL. In `y1` the type information in the Cogent AST is updated accordingly for every free occurrence of `v` and also after a re-binding to itself and by a `take` or `put` binding which both do not change the value of `v`.

The transformation always inserts the operation `notNull`, even if `v` has a readonly type. The readonly processing is performed after the NULL processing and may convert `v` to readonly by banging it, if it is not readonly. Therefore `notNull` is only replaced by `roNotNull` after readonly processing.

If `v` is a value variable the occurrence in the condition is its single occurrence and it will not be present in `y1`, so the transformation would have no effect. Therefore Gencot substitutes `v` by its bound expression, as long as it is a value variable. Then it is either a C variable or a component variable, or it is some other expression, such as a function application or a conditional expression. In the former case the variable may occur several times in `y1` and be affected by the NULL test implications. In the latter case only the syntactically same

expression could be affected and only, if variables used in it have not been rebound to other values. Even then the evaluation of the expression must be deterministic for the affecting being correct. Therefore Gencot only transforms NULL tests where the tested expression is a C or component variable after substituting value variables.

In the `Nothing`-branch it is known that `v` has value NULL. This is exploited by replacing `y2` by the expression

```
let v = cogent_NULL in y2
```

However, this would be interpreted by the readonly processing as a modification of `v` which could prevent banging it. Therefore this replacement is done in a separate postprocessing step which is executed after readonly processing. Then it has also been finally decided whether `v` has readonly type or not and instead of `cogent_NULL` the functions `null` or `roNull` can be used.

Resolving MayNull Clashes

In a third processing step Gencot resolves type clashes between `MayNull` wrapped types and unwrapped types. There are two cases: If the context expects an unwrapped type but the sub expression has a `MayNull` type this means that a pointer is expected to be not NULL, but this information is not present at the position where the pointer is used. The reason may be a missing NULL test in the C program or a too complex data flow dependency which is not detected by Gencot. This type clash is always resolved by replacing the sub expression by a dummy expression with the unwrapped type and signaling an error.

The second case is a context which expects a `MayNull` type but the sub expression has an unwrapped type. This means, it is known that the pointer is not NULL, but this information is not required in the context. This situation is always resolved by applying the Gencot operation `mayNull` to the sub expression which corresponds to "forgetting" the information of being not NULL.

Like for `notNull`, this transformation always inserts `mayNull`, for expressions of readonly type it is only replaced by `roMayNull` after readonly processing.

To preserve the restricted form of the Cogent AST described in Section 3.7.1 the `mayNull` application is always inserted as a separate binding of the form

```
v = mayNull[T] v
```

where `v` is the subexpression of type `T` which must be converted to `MayNull T`. The variable `v` on the left side of the binding has type `MayNull T`, therefore its type can be changed to `MayNull T` for every free occurrence in the scope of the binding.

This is implemented using function `resolveExpr` described in Section 3.7.6 together with a `ConvVarFun` which specifies the conversion between `MayNull` wrapped and unwrapped types as described above.

3.7.11 String Processing

**todo `stringproc`

3.7.12 Integer Conversion

Cogent supports the four arithmetic types `U8`, `U16`, `U32`, and `U64`. They have as values unsigned integers of the corresponding bitsize. In C values of the corresponding types are usually automatically converted according to their context. In Cogent explicit conversions are required.

If the context bitsize is larger than that of the expressions the `upcast` operator is used for conversion in all cases. If the context bitsize is lower than that of the expression the Cogent standard library provides abstract functions of the form `u32_to_u16` for conversion between two specific types. Since there is no function for converting from `U64` to `U8` this must be done in two steps via conversion to `U32`.

Gencot implements all these conversions between arithmetic types using function `resolveExpr` described in Section 3.7.6. Although it cannot cause errors, it is thus implemented by the monadic action

```
intproc :: GenExpr -> ETrav GenExpr
intproc = resolveExpr convVarIntDiffs
```

defined in module `Gencot.Cogent.Post.MatchTypes`, where `convVarIntDiffs` is the function of type `ConvVarFun` for integer conversions. In the case of conversion using the upcast operator the inserted binding uses an `Upcast` expression on the right hand side, this is the only case where Gencot generates this kind of expression.

3.7.13 Pointer Adaptation

**todo `pointerproc`

3.7.14 Processing `MayNull` Operations

The Gencot operations for values of `MayNull` type are defined in two variants for values which are readonly or not (see Section 2.7.10). The operations are introduced by the `NULL` pointer processing but the distinction between readonly and modifiable values is only known after readonly processing. Therefore the `NULL` pointer processing always generates the variants for modifiable values. They are replaced by the variants for readonly values in a separate processing step.

The operation `null` and its readonly variant `roNull` are used to represent the value `NULL` in Cogent. Gencot translates it to references of the variable `cogent_NULL`. Since a variable reference is easier to handle for the processing steps than an application of a (constant) function, `cogent_NULL` is only converted to `null` and possibly further to `roNull` after NULL pointer processing and readonly processing.

As described in Section 3.7.10 in the Nothing branch of generated `match` expressions the tested pointer variable is bound to NULL. Since that would be interpreted by readonly processing as a modification it would prevent banging for the pointer variable. Therefore the binding is only inserted after readonly processing. Then it is known whether the type is readonly or not, so the representation by the operation `null` or `roNull` can be used directly.

All these postprocessing steps cannot cause errors, hence they are implemented together by the function

```
opnullproc :: GenExpr -> GenExpr
```

defined in module `Gencot.Cogent.Post.MatchTypes`.

3.7.15 Take/Put Processing

For struct and array accesses in C (`s.m`, `s->m`, `a[i]`) and also for pointer dereferences (`*p`) the translation described in Section 3.6.10 creates pairs of take/put bindings of the form

```
<v>{m=p<k>'} = v<n>'
...
<v> = <v>{m=p<k>'}
```

(for struct access and pointer dereferences) and

```
<v> @{@v<l>'=p<k>'} = v<n>'
...
<v> = <v> @{@v<l>'=p<k>'}
```

(for array access). Here, `<v>` is the variable used for the container (struct, array, or pointer) and `p<k>'` is the component variable used to bind the component (member, element, or referenced data). The value variable `v<n>'` is the result of the expression denoting the container. Since in C this expression is always a subexpression of the access expression, its translation is positioned in one or more bindings before the `take` binding which ultimately bind `v<n>'`. Since the take/put pair only covers the C access expression the container cannot be modified by a side effect between the `take` and `put` bindings, i.e., the variable `<v>` cannot be re-bound between them.

There are several cases where the `put` binding is not present and has been replaced by a unit binding, either by the translation (see Section 3.6.10) or by postprocessing (see Sections 3.7.7 and 3.7.8). In all these cases the container is either specified by the error variable `err'` or it has readonly type.

If the container in a `take` binding is specified by the error variable, the container has been specified in C by an expression which is not a variable or an access chain starting at a variable. This case is detected by the translation phase and no corresponding `put` binding is generated. Note that the postprocessing described in Section 3.7.7 replaces occurrences of component variables in binding patterns by the error variable, but only in normal bindings and not in `take` bindings for nested accesses to components of the component.

There are three main postprocessing steps applied to take/put pairs:

- processing single `take` and `put` bindings according to the types of the container and the component.
- eliminating the `take` and/or `put` binding of a pair according to how the component is used after the `take`.
- converting take/put pairs to applications of `modify` or `modref`.

Postprocessing for `take` and `put` bindings is implemented by the function

```
tpproc :: GenExpr -> GenExpr
```

defined in module `Gencot.Cogent.Post.Takeput`.

The three substeps are implemented by the functions

```
tpsingle :: GenExpr -> ETrav GenExpr
tpelim :: GenExpr -> GenExpr
tpmodify :: GenExpr -> GenExpr
```

The first substep is implemented by a monadic action because it may signal errors. The other substeps never signal errors. The substeps are executed sequentially for a whole function body. Every substep converts or removes some take/put pairs, the following substep only processes the remaining pairs. After the third substep there may still remain take/put pairs which are usually valid Cogent code.

Processing Single `take` and `put` Bindings

The first substep detects erroneous take/put pairs and pairs where the container has readonly type. In both cases the `put` binding is simply removed. For erroneous pairs the `take` binding is replaced by a binding for the component variable to a dummy expression and a corresponding error is signaled. For a readonly container previous postprocessing steps guarantee that the component is not modified or replaced (see Sections 3.7.7 and 3.7.8), therefore the `put` binding is not needed and the `take` binding is replaced by a direct access to the component. Together, in all these cases both the `take` and the `put` bindings are removed.

Moreover, if the container is an unboxed record of nonlinear type (i.e., without linear components) the `take` binding can also be replaced by a direct access, but the `put` binding may still be needed, which is determined in the second substep. That substep, however, only processes complete take/put pairs. Therefore here the case is processed where not `put` binding is present, which is the case when the container has not been specified by a variable or access chain starting at a variable in C. As described in Section 3.6.10 the translation uses the error variable as bound container variable in the generated `take` binding. This property is used to recognize the case here.

Additionally the case is handled where a take/put pair represents a dereference `*p` where `p` has an array type in C, which has the meaning of accessing the first array element (with index 0). The `take` resulting from the translation is a record `take` for the array as container and the field `cont` as component (see Section 3.6.10). Like all arrays the container here is a wrapper record with a single field where the name encodes the number of array elements and which is always different from the name `cont`. If this situation is detected for a `take` it is replaced by an array `take` with index 0 and similar for a `put`. This is done before all other processing, so the resulting array take/put pair is processed like every other array take/put pair as described below.

The following erroneous situations are detected:

- The container has a `MayNull` wrapped type. This means it is not statically known that the container pointer is not NULL and the access may cause a NULL pointer exception. Gencot treats all these cases as error. This results in preventing all NULL pointer derefences in the generated Cogent code which can be formally verified. Only if the container type is

unwrapped, either caused by a Not-Null item property (see Section 2.6.1) or caused by a NULL test present in the C code and detected by NULL pointer postprocessing (see Section 3.7.10) the access is translated to Cogent.

- The container has type `CVoidPtr` or `CArrXX`. In the first case the C type was `void*` and no more specific type could be inferred during postprocessing. Then the access cannot be represented in Cogent since the component type is unknown. In the second case the C type is an array type where the number of elements could not be determined by Gencot. Then it is not possible to check the access whether the index is valid, this access cannot be represented in Cogent either. Since these are the only cases where the container has an abstract type, the error is detected by testing the container type for being abstract.

- The container has a linear type (either a boxed record or an unboxed record with a linear component) and is specified by the error variable in the `take` binding. This means it has been specified in C by an expression which is not a variable or an access chain starting at a variable, such as the result of a function application. Then an access to a component would always discard the remaining container since there is no variable to which the remaining container can be bound. This is not allowed in Cogent for a container of linear type and is treated by Gencot as an error. Note thatconstitutes in this case the `put` binding needs not be removed since the translation does not generate it (see Section 3.6.10). If the container has a nonlinear type (readonly or unboxed without linear component) the access is processed as described below or in the next substep.

- The take/put pair represents a dereference `*p` where `p` is a pointer to a struct in C. The `take` resulting from the translation is a record `take` for the boxed record corresponding to the struct as container and the field `cont` as component (see Section 3.6.10) with the unboxed record as its type. Even if the record has a field named `cont` its type cannot be the unboxed record which contains it. Therefore this case is recognized by Gencot by first testing whether the record has a field named `cont` and then checking its type. The case is treated as an error because the conversion from a boxed to an unboxed record cannot be represented in Cogent.

If the container has readonly type a `take` binding `<v>m=p<k>' = v<n>'` with container type `T` and component type `TC` is replaced by

- the pointer dereference application binding `p<k>' = getPtr[T,TC] v<n>'`, if `T` is a pointer type of the form `CPtr TC`,

- otherwise the `getref` application binding `p<k>' = getrefFld_m[T,TC] v<n>'` if the field `m` has unboxed type in `T` but `TC` is the corresponding boxed type,

- otherwise the member access binding `p<k>' = v<n>'.m`.

The array `take` binding `<v> @@v<l>'=p<k>' = v<n>'` is replaced by

- the `getref` application binding `p<k>' = getrefArr[T,U32,TC] (v<n>',v<l>')` if the elements have unboxed type in `T` but `TC` is the corresponding boxed type,
- otherwise the `get` application binding `p<k>' = getArr[T,U32,TC] (v<n>',v<l>')`.

In all these cases the container variable `<v>` does not occur in the replacement binding. Thus the result is also valid if the container variable was the error variable. Semantically this is correct because the remaining container after accessing the component may be discarded because its type is readonly.

The case where the container is an unboxed nonlinear record and `<v>` is the error variable is handled in the same way.

Eliminating `take` and `put` Bindings

The second substep checks how the accessed component is used after the `take` binding.

Eliminating `take` Bindings If the component has nonlinear type and is not used before or after the `put` binding the `take` binding can be removed. This corresponds in C to the case where the component value is overwritten by a plain assignment or it is neither used nor changed so that the access is a no-op. In the second case the `put` binding is removed as well. In the first case the `put` binding will be retained. This will result in a `put` binding which is not paired with a preceding `take` binding, so that it is applied to a container with a type where the component is not taken. Cogent accepts such `put` operations if the component has no linear type.

If the component is used the `take` binding cannot be removed, but if the container is an unboxed record of nonlinear type Gencot replaces the `take` binding by a member access. This may also result in a valid unpaired `put` binding as above.

If the component has linear type it would be illegal to overwrite it since that would discard it. In this case the `take` binding is retained unchanged, however, the Cogent compiler will signal an error because the taken component is discarded.

To determine whether the component is used, all free occurrences of the component variable after the `take` binding are inspected. If such an occurrence is on the right side of a binding, the free occurrences of the bound variable are transitively inspected in the same way. If the occurrence is in a tuple expression only the corresponding bound variable in the tuple pattern is inspected. A variable is considered to be used if it occurs free in the final body of the `let` expression which contains the `take` binding. The rationale for this is that normally a `let` expression corresponds to the translation of a C expression and the body represents the result of the expression (as value or side effect). So Gencot considers a component as used if it is part of the result of the expression which contains the component access. It is not guaranteed that the component is actually used in that case, e.g., the expression could be an isolated expression statement where the result value is discarded. However, such cases are assumed to be rare and, if they cause a problem, will be detected by the Cogent compiler.

If an array `take` is eliminated in this way and the corresponding array `put` binding

```
<v> = <v> @{@v<l>'=p<k>'}
```

is retained, it is converted to the `setArr` application

```
(<v>,()) = setArr[T,U32,TC] (<v>,v<l>',p<k>')
```

where `T` is the type of the container (array) and `TC` is the type of the component (array element). This is necessary because the generated array `put` bindings are always illegal Cogent code and must be replaced by Cogent array operations. The `setArr` operation is always applicable in this case because the array element has nonlinear type.

Eliminating `put` Bindings If the component variable is not re-bound before the `put` binding and the container has nonlinear type (either readonly or unboxed with no linear component) this means that the component is not modified or replaced. Then the `put` binding can be removed. Note that the case of a readonly container has already been processed in the first substep described above. Therefore here only the case of an unboxed container remains.

In this case the `take` binding is always eliminated as well or replaced by a direct access, as described above. Therefore no unpaired `take` binding remains which would change the type of the container to the type where the component is taken.

If the component variable is re-bound as container in the `put` binding of a nested access, it may depend on the elimination of that `put` binding whether the outer `put` binding may be eliminated. Therefore nested take/put pairs are processed for elimination starting with the innermost pairs. A nested `take` binding with the component as container is irrelevant when checking for rebindings of the component, since it is never retained without the corresponding `put` binding.

If the container has linear type, the `put` cannot be removed, even if the component is not re-bound. The remaining `take` binding cannot be replaced by a Gencot operation or member access in this case and would change the container type to a type with a taken component which would lead to a type error in Cogent as described above.

Converting take/put Pairs to `modify` or `modref` Expressions

In the third substep remaining take/put pairs are converted to applications of the `modify` or `modref` operations in the following cases:

- The pair consists of an array `take` and an array `put`, since these are always invalid Cogent code and must be replaced by Gencot operations.
- The container has linear type and in it the component has unboxed type but the component variable has the corresponding boxed type. This case cannot be represented by `take` and `put` operations in Cogent, it must be converted to an application of the Gencot `modref` operation.

The remaining take/put pairs after this step have the following properties: they consist of `take` and `put` bindings for records. The container type is not `MayNull` wrapped and is a boxed record which has the component as a field. The component variable has the type of this field. Together these `take` and `put`

are always valid Cogent code and are not further processed by Gencot. They could be replaced by an application of the `modify` operation, but since that must be defined specifically for the accessed field it is easier and more flexible to use the Cogent `take` and `put` operations.

If a take/put pair is converted to a `modify` or `modref` application the change function to be passed as argument to the application must be determined. It is constructed as a Cogent lambda expression. It takes as arguments the component and a tuple of additional inputs and returns the (possibly modified) component and a tuple of additional outputs. Its body is completely determined by the bindings between the `take` and the `put` binding.

The additional outputs are all variables which are bound in these bindings and are used in the bindings and the let-body after the `put` binding. Whether a variable is used is determined in the same way as described above for eliminating `take` bindings. If there is no such variable the additional output is the unit value. If there is only one such variable its value is used as additional output, otherwise all such variables organized in a tuple in some order.

The body of the change function is a `let` expression. Its bindings are the bindings between the `take` and the `put` binding. Its body is the pair consisting of the component variable and the additional output. In Cogent a tuple is always equivalent to a chain of right associatively nested pairs. Therefore in the case that the additional output is a tuple the body is flattened to a tuple whith the component variable as first element. The component variable may also be a part of the additional output, then it occurs twice in the body.

The additional inputs are all variables which are used in this body of the change function. The component variable cannot be a part of the additional input since it is first bound in the `take` binding. The additional input variables are organized in a tuple together with the component variable in the same way as the output variables. The corresponding variable tuple pattern is used as argument pattern in the change function lambda expression.

If converted to a `modify` application a sequence of bindings

```
<v> @{@v<l>'=p<k>'} = v<n>'
<bs>
<v> = <v> @{@v<l>'=p<k>'}
```

is replaced by the single binding

```
(<v>,ov..) = modifyArr[T,U32,TC,_,_]
  (<v>,v<l>',
   \(p<k>',iv..) => let <bs> in (p<k>',ov..),
   iv..)
```

where `<bs>` are the bindings between `take` and `put`, `ov..` is the sequence of additional output variables or unit `()`, `iv..` is the sequence of additional input variables or unit `()`, `T` is the container type, `TC` is the component type. The types of `iv..` and `ov..` for `modifyArr` are specified by `_`, Gencot assumes that they can always be derived by the Cogent compiler. A `modref` application for an array has the same form, only the name is different (`modrefArr`).

A `modref` application for a record take/put has the form

```
(<v>,ov..) = modrefFld_m[T,TC,_,_]
  (<v>,
```

```
  \(p<k>',iv..) => let <bs> in (p<k>',ov..),
  iv..)
```

where `m` is the accessed field. The main difference is the missing index type and variable `v<l>'`. If the container type has the form `CPtr TC` the take/put pair is always replaced by a `modify` application which has the same form with the name `modifyPtr`.

If a take/put pair is nested in another take/put pair the `put` bindings are generated immediately after each other. Between the `take` bindings there is always a binding `v<n>' = p<k>'` where `p<k>'` is the component variable of the outer pair and `v<n>'` is the container on the right side of the inner `take` binding. When processing the outer pair this binding is always eliminated by substituting `p<k>'` for `v<n>'` in the inner `take` so that also the `take` bindings follow each other immediately. From this it follows that the additional output variables of the outer pair are the same as for the inner pair and analogously for the additional input variables.

Gencot first processes the inner take. If it is replaced by a `modify` or `modref` application, the resulting single binding is the only content of the outer binding. If the outer binding is also replaced the resulting binding in case of two nested pairs for records has the form

```
(<v>,ov..) = modouter[T,TC,_,_]
  (<v>,
   \(p<k>',iv..) =>
     let (p<k>',ov..) = modinner[TC,TCC,_,_]
           (p<k>',chfun,iv..)
     in (p<k>',ov..),
   iv..)
```

where `chfun` is the change function for the inner pair, `modouter` and `modinner` are the corresponding `modify` or `modref` operations, and `TCC` is the type of the inner component. Now it is possible to extend the outer change function by an additional argument for the inner change function and pass it this way to the inner application:

```
(<v>,ov..) = modouter[T,TC,_,_]
  (<v>,
   \(p<k>',cf,iv..) =>
     let (p<k>',ov..) = modinner[TC,TCC,_,_]
           (p<k>',cf,iv..)
     in (p<k>',ov..),
   chfun, iv..)
```

Formally, the `chfun` has become an additional input to the outer change function, prepended to its own input `iv...` Thus, also the type of the additional input for `modouter` is now different from that for `modinner`, although this is not visible in the code because both are specified by `_`.

This binding can be simplified by substituting the inner binding in the `let` body followed by an eta reduction to

```
(<v>,ov..) = modouter[T,TC,_,_]
  (<v>,
```

```
modinner[TC,TCC,_,_],
chfun, iv..)
```

If the inner pair is for an array the `chfun` must be preceded by the array index used for the access, thus the index also becomes an additional input for the outer change function.

More generally, Gencot applies this conversion whenever, after processing the bindings between the outer `take` and `put`, these bindings consist of a single binding of the form

```
(p<k>',ov..) = f (p<k>',iv..)
```

even if the function `f` is not a `modify` or `modref` operation resulting from converting a take/put pair.

Gencot iterates this conversion as long as applicable. Here it is important that the types of the additional input are not specified explicitly for `modify` or `modref` functions. Every such type contains the type of the next inner change function which may lead to an exponentially increased code size if all these types are explicitly specified.

3.7.16 Combining the Steps

All postprocessing steps are designed and implemented in a standalone way so that they can be applied to arbitrary Cogent expression which are generated by the Gencot translation phase. However, to be most effective, they exploit several dependencies which must be respected by the order of execution.

Generally, the postprocessing steps recurse into the structure of the processed expression. Thus it would be possible to combine them on every recursion level by combining the non-recursive steps to a common postprocessing function which recurses into the expression structure. Alternatively each step recurses on its own and the overall simplification applies the steps sequentially to the toplevel expressions.

Some steps depend on each other in a cyclic way. These are in particular the functions `evalproc`, `letproc`, and `ifproc`. After `letproc` substitutes variables it may be possible to evaluate more expressions by `evalproc` than before. This may allow additional simplifications by `ifproc` since conditions may evaluate to a constant. These simplifications may remove code parts with occurrences of free variables which in turn may allow additional simplification by `letproc`. Therefore these steps must be applied in a loop until the expression does not change any more.

The `letproc` step works bottom up and is applied to subexpressions before it is applied to an expression. The `ifproc` step for a subexpression will only benefit if variables are substituted from the context. Thus it does not help to iterate these steps for the same subexpression. Instead, both steps recurse on their own and are iterated on the toplevel expressions.

The `evalproc` step is only used as auxiliary step in other postprocessing steps. Since the other steps recurse into the control structures like `if` and `let`, it is not necessary for `evalproc` to do this. Therefore `evalproc` only recurses into operator subexpressions.

The `opproc` step recurses into arbitrary subexpressions. It processes cases which are typically created by `ifproc`. Therefore it is executed after every toplevel execution of `ifproc`.

Many steps exploit the fact that the Cogent code has the restricted form described in Section 3.7.1. Since the combined steps `letproc`, `evalproc`, and `opproc` do not preserve this form, most other steps are executed before these. This implies that they are usually executed on a code with larger size and redundant parts, before the code is simplified. The advantage of an easier implementation on the restricted code form is prioritized by Gencot over an efficient execution of the postprocessing steps.

Moreover, variables can only be substituted by the expressions bound to them if both have the same type. As described in Section 3.7.6 this is in general not the case for the code generated by the translation phase. Therefore, all steps which process and remove type clashes by changing the additional type information for expressions must be executed before the simplification by `letproc`, until no type clashes remain. These steps are `bangproc` and `ebangproc` (3.7.8), `maynullproc` (3.7.10), `boolproc` (3.7.9), `stringproc` (3.7.11), `intproc` (3.7.12), and `pointerproc` (3.7.13) which are all implemented in the module `Gencot.Cogent.Post.MatchTypes`.

The following dependencies are taken into account for the order of these postprocessing steps:

- The `ebangproc` step is executed after `bangproc` because it is most effective if all readonly type clashes have been resolved for which that is possible.

- The `maynullproc` step includes nonlocal code transformations when it splits conditions. This may be complicated when variable bangings are present. Therefore it is executed before `bangproc`. Although it extends the generated code by introducing `match` expression which contain additional positions for variable bangings, such bangings are never possible in the generated `match` expressions. Therefore, these bang positions need not be taken into account by `bangproc`.

- The `maynullproc` step depends on recognizing NULL tests. In C a NULL test can be specified by the pointer alone, which is equivalent to the condition that it is not NULL. The `boolproc` step converts these cases into explicit NULL tests. Therefore `maynullproc` is executed after `boolproc` because then it is sufficient to recognize explicit NULL tests.

- The `intproc` step must be executed after `boolproc`, because that may convert boolean values to integer types which must then be converted further. It is delayed until after `maynullproc`, `bangproc` and `ebangproc` so that these need not process the additional bindings inserted for converting integer types.

The remaining postprocessing steps are `presimp`, (3.7.4), `romodproc` (3.7.7), `opnullproc` (3.7.14), and `tpproc` (3.7.15). For them the following dependencies are taken into account:

- The `presimp` step removes bindings of unused variables. Other than for `letproc` type clashes are not a problem for it. Therefore it may be applied before all type clashes are eliminated. Moreover, although it simplifies the Cogent code it preserves the restricted form generated by the translation phase.

- The `bangproc` step is more effective if unused values of non-escapeable type are removed, which would prevent banging a scope from which returns them. Therefore `presimp` is executed before `bangproc`. Since it generally reduces the code size and makes all other steps more efficient, `presimp` is executed as the first postprocessing step before all others.
- Like the `presimp` step the `romodproc` step removes values of non-escapeable type and makes `bangproc` more effective. The reason is that even if a container has readonly type it looks modified if accessed components are re-bound. This will prevent `bangproc` from discarding it when it is returned by a scope tried to be banged. Therefore, `romodproc` is also executed before `bangproc`.
- The `ebangproc` step is executed before `opnullproc` because the inserted NULL bindings would prevent `ebangproc` from banging the corresponding variables.
- A component may be re-bound but never used afterwards. In this case `romodproc` should not signal it as an error. Since `presimp` removes all such unused bindings, `romodproc` is executed after `presimp`, so that it need not consider the usage of re-bound components.
- The `opnullproc` step can only be executed after `maynullproc` has inserted the operations `mayNull` and `notNull`.
- The `tpproc` step depends on the types of containers and components to decide how to process take/put pairs. Therefore it must be executed after all steps which may modify the type of an expression.

The execution of every postprocessing step can be suppressed by adding a letter to the translation configuration string (see Section 3.11.3).

The resulting order of postprocessing steps with their suppress letters is

```
p presimp
r romodproc
c boolproc
n maynullproc
b bangproc
e ebangproc
i intproc
m opnullproc
t tpproc
l letproc <-
f ifproc   |
o opproc  -|
```

The last three steps are repeated until no more changes occur. This order and the suppressing of steps is implemented in module `Gencot.Cogent.Post.Proc`.

3.8 Reading C Types Generated by Cogent

For postprocessing the output generated by Cogent some information about how the Cogent compiler translates the Cogent types to C types is required. The

main reason is that Cogent does not preserve the Cogent type names, instead it generates type names of the form `t<nn>` in the C program.

Cogent translates record types, tuple types and variant types to C struct types. The Gencot array and pointer types are always mapped to a wrapper record in Cogent, so they are also translated to a C struct type. Function types are translated to enumeration types with an index number for each known function of that type. Basic types are translated using specific type names in C (`u8, u16, u32, u64, unit_t, bool_t`). Type `String` is always translated to `char *`. Abstract types are translated to C types of exactly the same name as in Cogent.

So the relevant information is that for all generated struct types and for the enumeration types generated for the function types. For struct types the information consists of the names and types of all members. For the latter the information consists of the parameter and result types and of the set of known functions.

This information is available in the main `.h` file written by the Cogent compiler. Gencot parses this file and provides the information in a textual representation which can be read and processed when needed.

3.8.1 Textual Representation of Type Information

The textual representation consists of a sequence of lines.

For the generated struct types every line describes a single struct type using the following format:

```
<name> <memnam>:<memtype> ... <memnam>:<memtype>
```

where `<name>` is the struct type name of the form `t<nn>` and each `<memnam>` is the member id. For Cogent record types the member ids are equal to the record component names. For Cogent variant types the member ids are equal to the alternative tags, with a member with id `"tag"` and type `"tag_t"` prepended. For Cogent tuple types the member ids are `p1, p2, ...`. For a Gencot array type the id of the single member is `arr<size>` (see Section 2.6.5). For a Gencot pointer type the id of the single member is `cont` (which cannot be distinguished from a record with a single component of that name).

The `<memtype>` specifies the member's type and may have three different forms. It may be a single identifier referencing a struct type (for tuples, unboxed records and variants), an enumeration (for function types), or a basic type. It may be an identifier followed by a single `"*"` (for boxed records and variants). Since in Cogent the case of pointer-to-pointer cannot be expressed a single star is always sufficient. Since the boxed form in Cogent is only available for records and variants, an identifier followed by a star normally has the form `t<nn>`. However, Gencot temporarily implements the type `MayNull b` where `b` is a boxed record by defining the typedef name `MayNull_tnn` as an alias for `tnn`, where `tnn *` is the type translation used for `b`. Thus an identifier followed by a star may also have the form `MayNull_t<nn>`.

The third form specifies a an array type as

```
<eltype>[<size>]
```

and is used for Cogent builtin array types. Since Cogent builtin array types are always wrapped in a struct, the element type `<eltype>` cannot be an array or

pointer type and is always a single identifier. The array size `<size>` is always specified by a literal positive number which is not zero (since Cogent builtin arrays cannot be empty).

The member specifications are separated by whitespace, every member specification is a single word without whitespace.

For the translated function types every line describes a single function type or a function belonging to a type. Every type description is immediately followed by the list of its functions descriptions. A type description uses the following format:

```
<name> <restype> <partype>
```

where `<name>` is the enumeration type name of the form `t<nn>`. The name and types are separated by whitespace. Note that in Cogent every function has exactly one parameter and one result.

A function belonging to a type is simply specified by its name (which is the same in Cogent and in the generated C code). So type and function specification lines can be distinguished by the number of their words.

3.8.2 Reading the C Source

The `.h` C source generated by Cogent uses C preprocessor directives to include files and to define macros. The included files contain type definitions for the translated basic Cogent types and for all abstract polymorphic types defined in the Cogent program. They are required for parsing the C code. The macros defined in the `.h` file are never used there and can be ignored. Thus Gencot processes the `.h` file using the standard C preprocessor `cpp` as described in Section 3.6.2 to execute and remove all preprocessor directives.

Then the file is parsed by a Haskell program using the language-c parser as described in Section 3.6.3. The file is read from standard input. It is parsed and then processed by the language-c analysis module using function `readFromInput` in module `Gencot.Input`. The resulting symbol table contains all globally defined identifiers.

To determine the required type information only the following definitions must be processed:

- `struct` definitions to determine all relevant struct types,
- dispatcher function definitions to determine the information about function types.

Definitions from included files are not needed, so their content is filtered from the symbol table as described in Section 3.6.3. The remaining symbol table entries are converted to a list of `DeclEvent` values and filtered according to the required entries. Together, Gencot uses the function `getDeclEvents` from module `Gencot.Input` in the same way as for reading an original C program for translating it to Cogent.

3.8.3 Generating the Struct Type Information

The struct type information is generated by a separate processing step. The `DeclEvent` list is filtered so that only the struct definitions remain. Since every

struct definition can be processed independently and the state information used for translatig a C source is not used here, no monadic traversal is required.

The processing is implemented by the function

```
procStruct :: DeclEvent -> String
```

defined in module `Gencot.Text.CTypInfo`. It is mapped to the filtered list of `DeclEvent`s and the resulting List of Strings is output as a sequence of lines.

3.8.4 Generating the Function Type Information

The function type information is generated by a separate processing step.

In its default configuration, Cogent uses the single enumeration type `untyped_func_enum` for enumerating the functions of all function types. They are defined in the form

```
typedef untyped_func_enum t<nn>;
```

This is not sufficient for retrieving the information about a function type.

Instead, the information is taken from the dispatcher functions. Cogent generates for every function type a dispatcher function of the form

```
static inline <restype> dispatch_<funtype>(
    untyped_func_enum a2, <paramtype> a3) {
  switch (a2) {
    case FUN_ENUM_<name>: return <name>(a3);
      ...
    default: return <name>(a3);
}}
```

If the type has only one known function the body only consists of the default return statement. The dispatcher function definition contains the function type name `<funtype>` and the `<restype>` and `<paramtype>`. The names of the functions can be read from the `return` statements in the body. The `DeclEvent` list is filtered so that only the dispatcher function definitions remain. Again, every dispatcher function definition can be processed independently, no monadic traversal is required.

The processing is implemented by the function

```
procFunc :: DeclEvent -> [String]
```

defined in module `Gencot.Text.CTypInfo`. It returns the list of strings consisting of the type description and all related function names. It is mapped to the filtered list of `DeclEvent`s and the resulting List is flattened and output as a sequence of lines.

3.9 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.

3.9.1 Implementing Polymorphic Functions

If a basic function is implemented in Cogent, it is automatically available for all possible argument types, according to the Cogent definition of the argument types.

Basic functions which cannot be implemented in Cogent are defined as abstract in Cogent and are implemented in antiquoted C. Antiquotation is used to specify Cogent types, in particular, the argument types of polymorphic functions. Then the Cogent mechanism for creating all monomorphic instances used in a program.

However, for some of the basic functions defined by Gencot the argument types are restricted in ways which cannot be expressed in Cogent. An example is the restriction to linear types. In Cogent the "permissions" `DS` express the restriction that an argument type must be sharable and discardable, i.e., *not* linear. But the opposite restriction cannot be expressed.

In such cases Gencot only provides function implementation instances for the restricted argument types. If the function is used with other argument types in a program this is not detected as error by the Cogent compiler. However, due to the missing C implementation the resulting C program will not compile.

Genops

Additionally, there are cases where the Cogent antiquotation mechanism is not sufficient for generating the monomorphic instances from a common polymorphic template. The antiquotation mechanism can replace Cogent type and type argument specifications by their translation to a C type, but it cannot generate other C code parts depending on these types. This is required for several of the Gencot basic functions.

To provide instances also for these functions, Gencot extends the Cogent antiquotation mechanism by a postprocessing mechanism called "Genops". It replaces specific code templates in the monomorphic function instances. The Genops templates are specified in the antiquoted C definition of the polymorphic function and contain an (antiquoted) Cogent type. When Cogent generates the monomorphic instance it replaces the type in the template by the corresponding C type. Afterwards, Genops reads the template, and expands it depending on the contained C type. If the C type results from a Cogent type for which Gencot does not provide a function instance, Genops signals an error. The required information about the generated C types is accessed by reading the struct and function type information described in Section 3.8.1.

The Genops template syntax is defined in a way that its insertion results in syntactically valid C code. Thus, Cogent can read and process the antiquoted C code with embedded Genops templates as usual and the resulting monomorphic instances are syntactically valid (plain) C code. Genops is implemented by reading the monomorphic instances, detecting and expanding the templates, and outputting the result again as valid C code.

Genops templates have the form of a C function invocation

```
genopsTemplate("XXX",(typ1) expr1, "genopsEnd")
```

where `XXX` names the template and `typ1` is an arbitrary C type embedded in the template, and `expr1` is an arbitrary C expression embedded in the template.

Both `typ1` and `expr1` are read by Genops and are used to determine the template expansion. If more than one type and/or expression is required, additional function arguments are used to specify them, each but the last followed by the string `"genopsNext"`. If only a type is required, `0` is used as expression. If only an expression is required, the cast with the type is omitted. The Genops templates always expand to a C expression, thus they are syntactically valid wherever the template may occur.

The syntax of the templates is selected for simple parsing, so that it is not required to use a parser for C types and expressions. It is assumed that the `typi` and `expri` in the template do not contain the strings `"genopsNext"` and `"genopsEnd"`, so they can be separated by searching for these strings. All templates can be found by searching for `genopsTemplate`. The `typi` and `expri` can be separated because the `typi` generated from Cogent types never contain parantheses (see Section 3.8).

Using a C parser is avoided because the result of processing an antiquoted C file by Cogent is usually only a fragment of a C program which contains types but not their definitions. The C syntax is not context free and a C parser needs to know which identifiers name types and which name values, therefore the files could not be parsed without additional information from the rest of the program.

3.9.2 Implementing Default Values

Default values are specified using the abstract polymorphic function `defaultVal` (see Section 2.7.2). This function is implemented with the help of the Genops template

```
genopsTemplate("DefaultVal",(typ) 0, "genopsEnd")
```

where `typ` is the type for which to generate the default value. The Genops mechanism replaces this template by the value as specified in Section 2.7.2.

The Cogent type is determined by its translation as C type `typ`. The primitive numerical types are translated to `u8`, `u16`, `u32`, `u64`, the unit type is translated to `unit_t`, the type `String` is translated to `char*`. Gencot function pointer types are represented by abstract types with names of the form `CFunPtr_...` and have the same name when translated to C. Tuple types, unboxed record types, and variant types, are translated to types `tNN` which are described in the struct type information (see Section 3.8.1). Unboxed Gencot array types are implemented as specific unboxed record types and thus have the same form. Tuple and record types are treated in the same way. Variant types are recognized by the first member having type `tag_t`. Gencot array types are recognized as double wrapper records with a member of array type.

Default values for aggregate types (structs and arrays, including the unit type which is translated to a struct type `unit_t`) are implemented with the help of C initializers. For these types the `"DefaultVal"` template expands to an initializer expression, it may thus only be used in an object definition. The initializer only specifies a value for the first member or element, according to the C standard this causes the remaining members or elements to be initialized according to the C default initialization, which corresponds to the Gencot default values as specified in Section 2.7.2. This also applies to objects of automatic storage duration, which are not initialized when no initializer is provided.

So the template can be used to initialize a local stack-allocated variable in a function to the default value and use it to copy the value elsewhere.

3.9.3 Implementing Create and Dispose Functions

3.9.4 Implementing Initialize and Clear Functions

The functions `initFull` and `clearFull` are implemented in antiquoted C by moving the boxed value from stack to heap and back in a single C assignment.

The functions `initHeap`, `clearHeap`, `initSimp`, and `clearSimp` must be implemented by treating each component of a struct type and each element of an array type separately. This is done with the help of type specific functions for initializing and clearing, which are generated automatically. They are invoked by two Genops templates:

```
genopsTemplate("InitStruct",(typ) expr, "genopsEnd")
genopsTemplate("ClearStruct",(typ) expr, "genopsEnd")
```

In the first case `typ` must be the translation of a Gencot empty-value type, in the second case it must be the translation of a Gencot valid-value type. In both cases `expr` must be an expression for a value of type `typ`. The templates expand to an invocation of a generated function which initializes or clears this value as described in Section 2.7.5.

For every type `t` occurring as `typ` in an `"InitStruct"` template, and also for every pointer type `t` which transitively occurs in a component or element of `typ`, Genops generates a C function `gencotInitStruct_t` which implements the initialization. Analogously Genops generates C functions `gencotClearStruct_t` for the `"ClearStruct"` templates.

The generated function definitions must be placed at toplevel before the first occurrence of the corresponding template. Since Genops does not parse the C file it uses a marker, which must be manually inserted, to determine that position. The marker has the form

```
int genopsInitClearDefinitions;
```

and is replaced by the sequence of all generated function definitions. The function definitions are ordered so that functions for components or elements of a type precede the function for the type and can be invoked there.

3.9.5 Implementing Pointer Types

3.9.6 Implementing Function Pointer Types

3.9.7 Implementing MayNull

The functions of the abstract data type defined in Section 2.7.10 are implemented in antiquoted C, no Genops templates are required.

The function `isNull` and the part access operation functions could be implemented in Cogent, but they are also defined as abstract and implemented in antiquoted C, for efficiency reasons. The implementation of `modifyNullDflt` uses the Genops template `"DefaultVal"` (see Section 3.9.2) to generate the default value for type `out`.

3.9.8 Implementing Record Types

3.9.9 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 `#(CArr<size> El)` to specify the amount of space.

Initializing and Accessing Arrays

All abstract functions for initializing and accessing arrays defined in Section 2.7.12 are implemented in antiquoted C using the following two Genops templates:

```
genopsTemplate("ArrayPointer",(typ) expr,"genopsEnd")
genopsTemplate("ArraySize",(typ) expr,"genopsEnd")
```

In both templates the `typ` must be the translation of a Gencot array type `CArr<size> El`. In the first template `expr` must be an expression of type `typ`, which denotes a wrapper record according to Section 2.6.5. The template is expanded to an expression for the corresponding C array (pointer to the first element), which is constructed by accessing the array in the wrapper record. In the second template `expr` is arbitrary and is ignored by Genops, by convention it should also be an expression of type `typ` as in the first case. The template is expanded to the size of all arrays of type `typ`, specified as a numerical literal.

3.9.10 Implementing Explicitly Sized Array Types

3.10 Generating Isabelle Code

As described in Section 2.12 Gencot extends the shallow embedding and the refinement proof generated by Cogent. This is implemented by

- providing predefined Isabelle code as theory files. These files are located in the Gencot distribution in the subfolder `isa`.
- generating additional Isabelle code in separate theory files, if it depends on the translated C program.
- modifying Isabelle code which has been generated by Cogent. This is only done in cases where there is no other solution possible.

The Cogent refinement proof proceeds in several steps starting at the shallow embedding, through several intermediate representations, ending at the representation of the C code abstracted by Autocorres. Gencot extends the proof by generating similar steps for the additional Gencot and Cogent operations provided in the extended shallow embedding.

3.10.1 Generating and Processing Isabelle Code

3.10.2 Extending the Shallow Embedding

Theory File Structure

The shallow embedding generated by Cogent for a program `X` consists of two Isabelle theory files. The file `X_ShallowShared_Tuples.thy` contains definitions for all types occurring in the Cogent program and declarations for all abstract functions in the Cogent program. The file `X_Shallow_Desugar_Tuples.thy` contains definitions for all non-abstract functions defined in the Cogent program.

There is a third file `X_ShallowConsts_Desugar_Tuples.thy` which contains Isabelle definitions for the constants defined in the Cogent program. It is only generated when the refinement proof to the first intermediate level is generated using command `shallow-tuples-proof` for the Cogent compiler. However, this file is not used, all constants are substituted by their value in the shallow embedding. Since the constants may be useful when reasoning about the shallow embedding, Gencot also includes this file.

This shallow embedding is transformed by Gencot to the following set of files:

`GencotTypes.thy`: this file is predefined and contains definitions used for implementing the Gencot types in Isabelle.

`X_ShallowShared_Tuples.thy`: this is a modified form of the file generated by Cogent, where some type declarations and type synonym definitions have been replaced. The file additionally imports `GencotTypes.thy` since its content may be used in the replaced type specifications.

`ShallowShared_Tuples.thy`: this is a program specific file generated by Gencot. It only contains an import of `X_ShallowShared_Tuples.thy`. It serves as an interface so that the content of `X_ShallowShared_Tuples.thy` can be imported in predefined files without the need to know the program specific prefix `X`.

`Gencot_TTT_Tuples.thy`: these are predefined files for every Gencot include file `TTT.cogent` which defines abstract Cogent functions. They import `GencotTypes.thy` and `ShallowShared_Tuples.thy` and define Isabelle specifications for the abstract Cogent functions defined in `TTT.cogent`.

`CogentCommon_ttt_Tuples.thy`: these are predefined files for some Cogent include files `ttt.cogent` in the Cogent standard library `gum/common` which define abstract Cogent functions. They import `ShallowShared_Tuples.thy` and define Isabelle specifications for the abstract Cogent functions defined in `ttt.cogent`

`X_Shallow_Gencot_Tuples.thy`: this is a program specific file generated by Gencot. It imports `X_ShallowShared_Tuples.thy` and all `Gencot_TTT_Tuples.thy` and `CogentCommon_ttt_Tuples.thy` for which `TTT.cogent` or `ttt.cogent` is used in the Cogent program, respectively. It may contain additional program specific parts of the shallow embedding.

`X_Shallow_Desugar_Tuples.thy`: this is the original file as generated by Cogent, with the only modification that it imports `X_Shallow_Gencot_Tuples.thy` instead of `X_ShallowShared_Tuples.thy`.

Together, the Gencot shallow embedding for a program `X` is made available in an Isabelle theory file by importing `X_Shallow_Desugar_Tuples.thy`. This file structure is designed with the goal to reduce modifications and program specific parts, which must be generated, to an absolute minimum.

The predefined files `GencotTypes.thy`, `Gencot_TTT_Tuples.thy` and `CogentCommon_ttt_Tuples.thy` are provided in directory `isa` in the Gencot distribution. However, they cannot be loaded directly from there by Isabelle, since they depend on the program specific file `ShallowShared_Tuples.thy` and no complete session can be defined for them in the Gencot distribution. For a complete shallow embedding which can be loaded by Isabelle the predefined files must be copied from the Gencot distribution to the directory where the other files have been generated. Then a session can be defined for them in a `ROOT` file there.

It would be possible to determine from the imports in `X_Shallow_Gencot_Tuples.thy` exactly those predefined files which are needed for a specific shallow embedding. However, transitive imports among the predefined files must be taken into account. Therefore a simpler solution is to always copy all predefined theory files to a specific shallow embedding, independent of which are actually needed.

3.10.3 Reasoning Support for the Shallow Embedding

Optionally, Gencot extends the shallow embedding by the additional rules as described in Sections 2.12.4 and 2.12.5, which results in the following additional files:

`Gencot_TTT_Lemmas.thy:` these are predefined files for every Gencot include file `TTT.cogent` which defines abstract Cogent functions. They import the corresponding `Gencot_TTT_Tuples.thy` and define additional rules for reasoning about the abstract functions.

`CogentCommon_ttt_Lemmas.thy:` these are predefined files for some Cogent include files `ttt.cogent` in the Cogent standard library `gum/common` which define abstract Cogent functions. They import the corresponding `CogentCommon_ttt_Tuples.thy` and define additional rules for reasoning about the abstract functions.

`X_Shallow_Gencot_Lemmas.thy:` this is a program specific file generated by Gencot. It imports `X_Shallow_Gencot_Tuples.thy` and all `Gencot_TTT_Lemmas.thy` and `CogentCommon_ttt_Lemmas.thy` for which `TTT.cogent` or `ttt.cogent` is used in the Cogent program, respectively. It may contain additional program specific parts of the reasoning rules.

To use these additional reasoning rules it suffices to import the file `X_Shallow_Gencot_Lemmas.thy` in addition to the file `X_Shallow_Desugar_Tuples.thy` for the shallow embedding.

3.10.4 Extending the Shallow Tuples Proof

The first intermediate representation used by Cogent is directly equivalent to the Cogent core language. The shallow embeddding is actually generated from this representation by reversing the replacement of Cogent tuple types by record types in the core language. As a consequence, both representations have a similar structure and the theory files are named by omitting the `_Tuples` part from the shallow embedding file names.

The extensions provided by Gencot for the intermediate representation follow the same approach and the same naming scheme. It consists of the generated files `ShallowShared.thy`, `Gencot_TTT.thy`, `CogentCommon_ttt.thy`, `X_Shallow_Gencot.thy`, the modified files `X_ShallowShared.thy` and `X_Shallow_Desugar_Tuples.thy`, and the file `GencotTypes.thy` which is common for both representations. Additional reasoning rules as described in Section 3.10.3 are not generated for this representation.

Cogent generates the refinement proof between both representations in a single theory file `X_ShallowTuplesProof.thy`. This file imports program-independent parts of the proof from theory file `ShallowTuples.thy`. Gencot extends the proof by modifying `X_ShallowTuplesProof.thy` and providing additional program-independent parts in the theory file `ShallowTuples_Gencot.thy`.

3.11 C Processing Components

As described in Section 2.1.2 there are several different Gencot components which process C code and generate target code.

3.11.1 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 3.6.3.

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-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-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 3.3).

`items-gen` processes a single file `x.c` and generates a list of all items with internal linkage for which properties may be declared.

`auxcog-ctypstruct` processes a single file `x.h` generated by Cogent and outputs a description of all `struct` types.

`auxcog-ctypfunc` processes a single file `x.h` generated by Cogent and outputs a description of all function types.

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 3.6.4). 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 derived types.

`gencot-dvdtypesh` generates the content to be put in the file `<package>-dvdtypes.h`. It processes 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.

`items-used` determines a list of all declared external items which are actually used by the Cogent compilation unit.

`items-externs` generates a list of all items with external or no linkage for which properties may be declared.

3.11.2 Used External Items

External items are those C items (functions, objects, types) which are known in the sources of the Cogent compilation unit but not defined there.

Functions and objects are external, if they are declared in the Cogent compilation unit but are not defined there. They may be defined in files of the package which do not belong to the Cogent compilation unit (then they are declared in `.h` files of the package), or they may be defined in standard runtime libraries (then they are declared in system include files). In both cases they must have external linkage.

Gencot always assumes that all package include files used by the Cogent compilation unit are translated to Cogent and are thus a part of the Cogent compilation unit. Thus a type can only be external if it is defined in a system include file.

If an external item is actually used (by invoking a function, accessing an object, or referencing a type) in the Cogent compilation unit, it must also be translated to Cogent so that it is known in the Cogent program. Functions are translated in the form of exit wrappers, objects and types are translated as usual.

Usually, C programs know many external items without actually using them. To keep the resulting Cogent code as small as possible, Gencot performs an automatic analysis to determine the external toplevel items which are actually used by the Cogent compilation unit. The result of this analysis can be manually overridden by explicitly specifying additional external items which also should be processed.

The processor `items-used`

The processor `items-used` generates a list of the external toplevel items used by the Cogent compilation unit. It is intended as a filtering list for the processors `gencot-externs`, `gencot-exttypes`, `gencot-dvdtypes`, `items-externs` and `parmod-gen` which process external items, to reduce them to the relevant ones.

The processor is invoked as

```
items-used <additional items file>
```

where the argument is a list of additional external toplevel items specified manually.

A toplevel item is a struct, union, or enum, a typedef name, a function or a global variable.

The processor writes a textual list of the used external items to its standard output. Each item is output in a separate line identified by its item identifer as described in Section 3.2.1. The argument file content must have the same form, it is added to the output. These additional items cannot be added afterwards, since they may use other items which can only be detected by `items-used` if the additional items are known to it.

As conveniency for the developer, the argument file may contain empty lines and may contain comment lines which start with a hash sign `#`.

Determining Used Functions

A function is used by the Cogent compilation unit if it is invoked by a function which is part of the Cogent compilation unit. Gencot determines the invoked external functions with the help of the call graph as described in Section 3.6.15.

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 3.6.4. Then it determines for every table the invoked functions using the call graph. 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 3.6.4 and the invocations are reduced to those for which a declarations is present in the combined table. These are the invocations of functions (including function pointers) external to all read C sources.

Although `gencot-externs` only processes complete function declarations, `items-used` 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. Additionally, these functions may be needed by `gencot-exttypes` and `gencot-dvdtypes` to process their result types.

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. Therefore `items-used` will detect the macro as a used external item.

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 the function item id must be provided manually in the argument file.

Determining Used Variables

Gencot determines usage of an external variable if it or a sub-item of it is invoked as a function, using the call graph. Invocations of sub-items which are the result of another function are currently not detected. Variables accessed in other ways (e.g., read access, assignment) must be added manually in the argument file. This provides more control about the processed external variables for the developer.

Collecting Used Types

As described in Section 2.6.1, type items are all typedef names, tag names, generated names for tagless struct/unions, and some derived pointer types. Typedef names and tag names can be external, if they are declared in a system include file. Tagless struct/unions and derived types cannot be extern in this sense, because for them the declaration is identical with their single use. However, the following situation can lead to external used items of this kind: a typedef name is defined for them in a system include file. This typedef name is used indirectly by the Cogent compilation unit and is thus resolved. Then the result of resolving the name becomes the used external type item.

Properties declared for a derived pointer type item are always applied to the individual items having that type. Thus it is always possible to declare the properties directly for the individual items and Gencot does not support external pointer type items for this reason. For a tagless struct/union the situation is different, it is translated to a single defined type and declared properties must be used in this type definition, they cannot be replaced by properties declared for the individual typed items. Therefore Gencot also supports tagless struct/unions as used external items, identified by their item identifier using a generated name as described in Section 3.2.1.

A type item can be used by code in a translated C soure file or by a declaration of an external function or variable. Therefore, `items-used` first determines the external invoked functions and variables as described above, including the additional items in the argument file.

Type items are reduced by an analysis of type usage by reference. Every item can be associated with the type items referenced in the types of all sub-items. This usage relation is transitive, because in a definition of a composite type or typedef name, other type items can be referenced. To determine the relevant type items defined in system include files, `items-used` starts with all initial items belonging to the Cogent compilation unit, together with all external invoked functions and the processed external variables. It then determines all transitively used type items (which all must be defined in system include files).

To further reduce external items, external type name which are not directly used in the Cogent compilation unit are resolved, i.e. always replaced by their definition (which may again reference external type items).

For the resulting set of type items the item identifiers are added to the output list.

The initial toplevel items are collected with the help of the callback handler during analysis, as described in Section 3.6.3. All toplevel items can be wrapped as a `DeclEvent`. The callback handler is automatically invoked for all toplevel items 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 toplevel items to the list in the user state of the `Trav` monad. The handler is used for every `.c` file separately, the collected items must be combined afterwards to get all items in the Cogent compilation unit. This is done by first combining the symbol tables as described in Section 3.6.4, and then using the union of all items which are local, have internal linkage, or are still contained in the resulting combined table. This will avoid duplicate occurrences of items.

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 and has the parameters as sub-items. Function and global variable declarations are also ignored, since they represent external functions and variables which are determined separately. 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 items defined in system include files are omitted, since they will be determined using the type usage relation. To determine whether an item 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 above and added to the combined items

determined by the callback handler. No duplicate items will result here since the combined symbol table contains every external declaration only once. External invoked composite type members are not included, because they are sub-items of the composite type. To invoke a composite type member an object must exist which is an item with the composite type, so the composite is always a used type item.

The resulting set of items is transitively completed by adding all type items used by referencing them, as described above. This will add all required type items defined in system include files. Duplicates could occur if a type item is referenced by several different types, they must be detected and avoided. Since all toplevel items correspond to syntactic entities in a source file, type items can be identified and compared by their position and source file.

Finally, the type items are reduced to those defined in a system include file using the same heuristics as above. Together with the item identifiers of the external invoked functions and variables their items identifiers are written to the output.

3.11.3 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
- object definitions
- type definitions

The remaining global items are removed by the predicate passed to `Gencot.Input.getDeclEvents`: all declarations, 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 3.6.3.

Translating Toplevel Definitions

The translation of the `DeclEvent` sequence is implemented by the monadic action

```
transGlobals :: [DeclEvent] -> FTrav [GenToplv]
```

in module `Gencot.Cogent.Translate`. It performs the monadic traversal as described in Section 3.6.14 and returns the list of toplevel Cogent definitions of type `GenToplv`. It is implemented by mapping the function `transGlobal` to the `DeclEvent` sequence.

A struct/union/enum definition corresponds to a full specifier, as described in Section 2.8.1. The language-c analysis already implements moving all nested full specifiers to separate global definitions and the sorting step done by `Gencot.Input.getDeclEvents` creates the desired ordering. Therefore, basically the `DeclEvent`s in the list could be processed by `transGlobal` independently from each other.

However, this approach would create inappropriate origin markers which could result in duplicated conditional preprocessor directives and comments. If a definition in lines $b1 < e1$ contains a nested struct definition in lines $b2 < e2$ independent translation would wrap the translated nested struct always in origin markers for $b2$ and $e2$. If $e1 = e2$ the corresponding origin marker would occur twice. A better approach is to use origin markers for $b1, e1$ to wrap the whole sequence consisting of the translated definition together with the translations of all nested structs. Then, the translated nested struct would be followed by the two origin markers for $e2$ and $e1$ which can be reduced to a single marker if $e2 = e1$.

Therefore function `transGlobal` is defined as

```
transGlobal :: DeclEvent -> FTrav [GenToplv]
```

It translates every definition together with all nested struct/union/enum definitions and returns the sequence of the corresponding Cogent toplevel definitions, wrapped by the origin markers as described.

Since the nested definitions still occur seperately in the list of `DeclEvent`s processed by `transGlobals`, their processing must be suppressed. This is implemented by sorting the list according to the position of the first character in the source file (which will put all nested definitions after their surrounding definitions), marking processed nested definitions in the monadic user state, and testing every struct/union/enum definition found by `transGlobal` in the `DeclEvent` list whether it is marked as already processed.

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.

Translating Type References

A type reference is translated by the function

```
transType :: Bool -> ItemAssocType -> FTrav GenType
```

It must be paired with the associated item identifier as an `ItemAssocType` as described in Section 3.2.2. The identifier is used to retrieve the declared item properties required for the translation.

The first parameter specifies whether typedef names should be resolved when translating the type. The reason why typedef names are resolved as part of the type translation function is that item properties may be associated with typedef names or their sub-items. If typedef names would be resolved in a separate function the item identifier could not be associated anymore afterwards. This would cause the declared item properties for typedef names to be ignored when translating the resolved type.

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 2.6.8. An exception are externally defined typedef names not used directly in the Cogent compilation unit, they are resolved. 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 `()`. For struct and union types the unbox operator must be applied to the Cogent type name. 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 also be modified by applying the unbox operator.

Primitive types or typedef names referencing a primitive type are semantically unboxed. However, if marked as unboxed, the Cogent prettyprint function will add an explicit unbox operator. For better readability Gencot specifies these types as boxed, to suppress the redundant unbox operator.

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, union, or array type or a typedef name referencing such a type, the pointer type is translated to the translation of the base type in its boxed form. If it is a function type (possibly after resolving type names), the function type is encoded as described in Section 2.6.7 using the function `encodeType`, then the function pointer type is constructed from the encoding as described in Section 2.6.7. The monadic operation

```
encodeType :: Bool -> ItemAssocType -> FTrav String
```

works similar to `transType` but returns the encoding as a string instead of the translated Cogent type. In particular, it optionally resolves typedef names and respects item properties. If the base type is `void` the special type name `CVoidPtr` is used.

In all other cases, as described in Section 2.6, the pointer type is translated to the parametric type `CPtr` with the translated base type as type argument.

For an array type, the translation is the parameterized type for an array according to the number of elements. The translated base type is used as type argument. In case that the base type is again an array type (multidimensional

array), an explicit unbox operator is applied to the type argument to differentiate the multidimensional array from an array of array pointers, as described in Section 2.6.5.

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. Declared Read-Only and Add-Result properties are used to determine whether a parameter type should be translated as readonly, or whether it should be returned as part of a result tuple.

The translation functions do not respect the `const` qualifiers in C type specifications, they only use the Read-Only property to determine whether a type should be translated as readonly. The `const` qualifiers are used by `items-gen` to generate the default properties as described in Section 3.11.9.

Configuring Function Body Translations

C function bodies can be translated in different ways. This is mainly intended for purposes of testing Gencot. The way how C function bodies are translated is determined by a translation configuration string. It consists of single letters in arbitrary order where every letter suppresses a step in the translation.

The letter `A` completely suppresses the automatic translation of C function bodies. Instead, a simple expression for a default result is generated for every function according to the function result type. It has the form of a tuple of values for the result where the components are either the unit value `()`, if the type of the corresponding result component is the unit type, the value `0`, if the type of the corresponding result component is one of the arithmetic types `U8`, `U16`, `U32`, `U64`, and the dummy expression `gencotDummy ""` for all other types. Additionally the original C code of the function body is appended in a comment, after processing it as described in Section 3.6.9. Together, this is always valid Cogent code which is accepted by the Cogent compiler, it can be used as a starting point to construct manual translations of all function bodies.

If `A` is specified as translation configuration string all other letters with the exception of `G` are ignored. Otherwise, the basic translation described in Sections 3.6.10 and 3.6.12 is performed and every letter suppresses a single post-processing step. See Section 3.7.16 for the list of postprocessing steps and the configuration letters used for suppressing them. Note that suppressing post-processing steps will usually result in invalid Cogent code, because Gencot uses intermediate forms which only become valid code after further postprocessing.

If additionally the letter `G` is specified in the translation configuration string the additional type information is output which is maintained by Gencot in the Cogent AST as described in Section 3.6.11. This always results in invalid Cogent code because the type information is added to every expression using a specific syntax which is not supported by Cogent.

Filter `gencot-translate`

The filter `gencot-translate` is invoked as

```
gencot-translate <source name> <aux-namap> <aux-props> <aux-items> <tconf>
```

where the arguments are the name of the original C source file, a file `<aux-namap>` with the name prefix map (see Section 3.6.7), a file `<aux-props>` with the item property declarations (see Section 3.2), a file `<aux-items>` with the identifiers of the used external items as created by `items-used` (see Section 3.11.2), and the translation configuration string `<tconf>`, as described above.

3.11.4 Entry Wrappers

The entry wrappers are generated by the filter `gencot-entries`. It processes all `DeclEvent`s which are function definitions with external linkage as described in Section 2.9.3. Additionally it processes all `DeclEvent`s which are object definitions as described in Section 2.9.4.

The filter must be able to translate all C types which may occur as parameter or result type or as type of a global variable to Cogent. It uses the same function `transType`, like `gencot-translate` (see Section 3.11.3). For this `gencot-entries` requires the same additional information as `gencot-translate`: the name of the original C source file, the item properties and the used external items.

The generation of the entry wrappers and variable definitions is implemented by the monadic action `genEntries` in the module `Gencot.C.Generate`:

```
genEntries :: DeclEvent -> FTrav [Definition]
```

The result is a sequence of C function and object definitions represented in the reimplemented language-c-quote AST in module `Gencot.C.Ast` (see Section 3.6.9). It is output using the reimplemented pretty printer from `Gencot.C.Output`, as described in Section 3.6.13.

Origin markers are added by wrapping every entry wrapper and variabe definition according to the start and end position of the original C function definitions. This makes it possible to insert the conditional directives from the C source, so that for a conditionally defined C function also the entry wrapper or variable is defined conditionally. Otherwise, the invocation of the translated function in the wrapper would lead to an error when the generated C code is compiled.

Filter `gencot-entries`

The filter `gencot-entries` is invoked as

```
gencot-entries <source name> <aux-namap> <aux-props> <aux-items>
```

where the arguments are the name of the original C source file, a file `<aux-namap>` with the name prefix map (see Section 3.6.7), a file `<aux-props>` with the item property declarations (see Section 3.2) and a file `<aux-items>` with the identifiers of the used external items as created by `items-used` (see Section 3.11.2).

3.11.5 External Functions and Variables

Used external functions and variables ("objects") are translated so that they are known and usable in the Cogent program.

Translating External Functions and Variables

Normally, a used external function is translated as an exit wrapper. For the exit wrapper functions Gencot generates the Cogent abstract function definitions and the implementations in antiquoted C. The information required for each of these functions is contained in the C function declaration.

A C 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 Gencot does not generate exit wrappers for incompletely declared functions.

As described in Section 3.11.2 a used external function seen by Gencot may also be a macro call which shall be translated to Cogent. In this case a dummy function declaration must have been manually provided for it. By using an incomplete declaration the generation of an exit wrapper can be suppressed for macros which shall be translated to a macro in Cogent.

If an external function is accessed through a function pointer no exit wrapper is generated for it. Function pointers are invoked as described in Section 2.7.9. Since in C all functions which are sub-items of other items must be function pointers, exit wrappers are only generated for toplevel items, sub-items are not considered here.

For a used external object (including invoked function pointers) with ConstVal property Gencot generates a Cogent abstract function definition for an access function and provides an implementation in antiquoted C together with the exit wrappers for external functions. Again, this is only done for toplevel items, sub-items are not considered.

Processor `gencot-externs`

The processor `gencot-externs` generates the Cogent abstract function definitions. It is invoked as

```
gencot-externs <aux-namap> <aux-props> <aux-items>
```

where the arguments are a file `<aux-namap>` with the name prefix map (see Section 3.6.7), a file `<aux-props>` with the item property declarations (see Section 3.2) and a file `<aux-items>` with the identifiers of the used external items as created by `items-used` (see Section 3.11.2).

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 3.6.4. The tables are combined and from the resulting table all declarations of toplevel items are retrieved which are listed in the file of used items.

The processor translates these declarations using the monadic action `transGlobals` of module `Gencot.Cogent.Translate`. It translates C function and variable declarations to abstract function definitions in Cogent. It uses the item properties to fine tune the translation. The result is output using `prettyTopLevels` from module `Gencot.Cogent.Output`.

Processor `gencot-externsac`

The processor `gencot-externsac` generates the exit wrapper implementations in antiquoted C code. Additionally it generates before the exit wrapper definitions:

- declarations of all used external functions, so that they can be invoked by the exit wrappers.
- declarations of all used external global variables. For those having a ConstVal property declared their declaration is followed by the implementation of the access function (see Section 2.6.1).
- declarations of all non-external global variables with a declared GlobalState property.

The latter two are generated here for two reasons. First, the global variables may be accessed by arbitrary entry wrapper implementations. The file generated by `gencot-externsac` is included before all entry wrapper source files and thus can declare variables for all entry wrappers, also for the non-external variables which are defined somewhere in the entry wrapper files. Second, the definitions or declarations of the global variables must be known to generate the declarations. Since they may not be visible in every single C source file, they must be generated by a processor of all C sources in the package using the combined symbol table (see Section 3.6.4).

Non-external global variables may have internal linkage and thus be no more present in the combined symbol table. Therefore they must be processed before the tables are combined. Since the variable names are mapped using the source file name, the names of the original source files must be passed to `gencot-externsac` as an additional argument. After reading all source files, for each a monadic traversal of its toplevel variables with internal linkage is performed using `runFTrav` to generate their declarations. Then the tables are combined and another monadic traversal of the combined table is performed to generate the remaining declarations and the access function and exit wrapper implementations.

3.11.6 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 <aux-namap> <aux-props> <aux-items>
```

where the arguments are a file `<aux-namap>` with the name prefix map (see Section 3.6.7), a file `<aux-props>` with the item property declarations (see Section 3.2) and a file `<aux-items>` with the identifiers of the used external items as created by `items-used` (see Section 3.11.2).

The task of `gencot-exttypes` is to translate all type item definitions listed in the file of used external items. It parses and analyses all C source files specified in the input file name list and combines the resulting symbol tables as described in Section 3.6.4. The type item (struct/union, tagged enum, and typedef name) definitions in the resulting table listed in the argument file are then translated.

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 typedef names not used directly are resolved in these definitions, as described in Section 3.11.2. The same monadic action `transGlobals` is used as for `gencot-translate`.

3.11.7 Derived Types

The processor `gencot-dvdtypes` generates all Cogent type definitions required for the derived types used in the C program. It is invoked as

```
gencot-dvdtypes <aux-namap> <aux-props> <aux-items>
```

where the arguments are a file `<aux-namap>` with the name prefix map (see Section 3.6.7), a file `<aux-props>` with the item property declarations (see Section 3.2) and a file `<aux-items>` with the identifiers of the used external items as created by `items-used` (see Section 3.11.2).

A C derived type, is a pointer type, array type or function type. As described in Section 2.6, Gencot usually maps these types to parameterized Cogent types. The generic types `CPtr`, `CFunPtr` and `CFunInc` used for pointer and function types are predefined by Gencot. However, function pointer types use abstract types for encoding the base function type and the generic types of the form `CArr<size>` used for array types syntactically contain the array size, both depend on the translated C program. For these types Cogent definitions must be provided.

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 item associated types as described in Section 3.2.2. These include all types used in translated C sources, in declarations of external used functions and variables and those defined in system include files.

Together, `gencot-dvdtypes` processes *all* item associated types for all subitem types which are a derived array or function pointer type.

Generating Type Definitions

For all derived array types a generic type definition must be generated (see Section 2.6.5). However, only one such definition may be generated for every array `<size>`, although array types with a specific size may be associated with several different items and element types.

For all derived function pointer types an abstract type definition or a type synonym for the fully resolved type must be generated for encoding the function type used as its base type (see Section 2.6.7). For fully resolved function types an additional type synonym definition must be generated for the translated function type.

Gencot first generates the translated Cogent type names for all derived array and function pointer types associated with an item. Then it generates the corresponding definitions for every type name used in all these derived types. If

a translated array type uses the generic name `CArrXX` no definition is generated since this type name is predefined by Gencot.

Generating the type definitions for derived array and function pointer types is implemented by the monadic action

```
genTypeDefs :: [DeclEvent] -> FTrav [GenToplv]
```

in module `Gencot.Cogent.Translate`, where the list of `DeclEvent`s corresponds to the items to be processed.

In the type definitions for derived array types the array size must be specified as a Cogent expression. However, only restricted forms of expressions which can be evaluated at compile time are allowed here (e.g., no function application). These expressions have the same syntax in C and in Cogent, therefore the current Gencot version does not translate them to Cogent. Instead, it only translates them from the language-c AST to language-c-quote AST, as described for function bodies in Section 3.6.9.

The only identifiers which may occur in an array size expression are preprocessor macros, usually representing constant values in C. It would be possible to map these names to Cogent, as it is done in function bodies, since Gencot introduces a Cogent constant for every preprocessor constant. However, these constants may have types other than `U32`, such as `U8` or `U16`, which is not allowed in array size expressions (not even when using the upcast operator). Since the original preprocessor constants are still defined in the generated Cogent code, the names are not mapped in array size expressions. This implies, however, that the generated type definitions must be placed in the Cogent source after all preprocessor constant definitions. This is achieved by including the output of `gencot-dvdtypes` after that of `gencot-translate` in the generated main Cogent source file (see Section 3.12.8).

Processor `gencot-dvdtypesh`

3.11.8 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 3.4.4. This information is used to move declaration comments to the corresponding definitions, as described in Section 2.2.4. The filter is invoked as

```
gencot-deccomments <aux-namap>
```

where the argument is a file `<aux-namap>` with the name prefix map (see Section 3.6.7). This file is required since the written information includes the mapped names of the declarations.

Only global declarations are processed by this filter. Therefore the filter processes the list of `DeclEvent`s returned by `getDeclEvents` (see Section 3.6.3). 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 need not be moved from the declaration to the definition.

Processing the declarations is implemented by the monadic action

```
transDecls :: [DeclEvent] -> FTrav [String]
```

in module `Gencot.Text.Decls`. A monadic action is required here for accessing the name prefix map. The resulting string list is output one string per line.

3.11.9 Item Properties

To support an easy manual declaration of item properties Gencot generates default declarations from the C sources for all items with an identifier. The declarations are either empty or declare the properties Read-Only and/or Add-Result, if they can be derived from the item's type.

The properties Heap-Use and Input-Output are not set in default declarations. Heap-Use could be set if a C memory management function such as `malloc` is directly invoked by a function, however, the most relevant way to set both properties is by transitive dependency among functions. This is implemented by the parameter modification mechanism (see Section 2.11.5) and therefore omitted here.

The filter `items-gen`

The filter `items-gen` is used to generate default item property declarations from a single C source file (`.h` or `.c`). It is invoked as:

```
items-gen <source file name>
```

The filter processes as individual items all `DeclEvent`s in the source file which are function definitions or definitions for objects. The filter processes every definition of a composite type (with or without a tag) and every type name definition in the source file as collective item. (The filter does not process any derived pointer types as collective items.) The processing is done by the monadic action

```
transGlobals :: [DeclEvent] -> FTrav ItemProperties
```

defined in module `Gencot.Items.Translate`.

For every item, if its declared type (in the case of a typedef name item its defining type) is a derived function, array, or pointer type, its sub-items according to the type are processed recursively. For a composite type item its members are processed recursively as sub-items.

An item is processed as follows. If it is a global individual item (global variable or function) an empty property declaration is generated, since a global variable may be affected by a Global-State property and a function by a Heap-Use or Input-Output property. Otherwise (local variable, type, or subitem), if its declared or defined type is primitive or an enum or a function pointer no output is generated since properties are not supported or always ignored for it. Otherwise, if it is a parameter item, or its declared type is not composite, a property declaration is generated for it (parameters of composite type may be affected by the Add-Result property, items of pointer or array type may

be affected by the Read-Only property). For non-parameter and non-global-variable items of composite type no output is generated since properties are not supported or always ignored for them.

If the non-function-pointer type of an item is intrinsically readonly, the Read-Only property is declared for it. If an item has an intrinsically readonly array type and is not an element of another array or a member of a composite type (in both cases it is translated with an unboxed type and the Read-Only property would be ignored), the Read-Only property is declared for it. Otherwise, if it is a parameter and its type is linear (is a pointer or array or contains any), the Add-Result property is declared for it. Otherwise the declaration is empty.

All generated declarations are written in textual form to the standard output.

Whether a type is intrinsically readonly depends on its `const` qualifiers. 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. All other array types and all pointer types are translated to linear types which can be mutable in Cogent. To be intrinsically immutable, the base types of these types and all transitively contained array and pointer types must be const qualified. The immutability is determined by function`isReadOnlyType`. It is implemented by a monadic action since it must access the symbol table to resolve references to tagged C structs.

Default Add-Result properties are generated whenever the parameter type is linear. In cases where the parameter shall be discarded because it is deallocated in the function or it is already returned as the function's result the default property must be removed manually or with the help of the parameter modification descriptions before using the properties for translating the function.

The processor `items-externs`

The processor `items-externs` is used to generate the default property declarations for items intended to be used by `gencot-externs` (see Section 3.11.5) and `gencot-exttypes` (see Section 3.11.6). It is invoked as

```
items-externs <used external items file>
```

where the argument is the list of external toplevel items used by the Cogent compilation unit, as it is created by `items-used`.

The processor reads the source files of the Cogent compilation unit as described in Section 3.6.4 and processes all external toplevel items specified in the list of used items in the same way as `items-gen` to generate default property declarations for them on standard output. The same monadic action `transGlobals` is used as by `items-gen`.

The type of a processed item may involve an externally defined type name which is not in the list of used external items. This means that the type name is not used directly in the Cogent code and is resolved in the translation. In this case it is also resolved by `items-externs` so that collective sub-items of the type name become individual sub-items of the processed item, as described in Section 2.6.1.

If a processed external function or object belongs to the <package> but is outside the Cogent compilation unit, it may have type subitems which are defined in an include file which is also used in the Cogent compilation unit. In this case the subitems are not extern, but still output by `items-externs`. They

must be removed in a postprocessing step which recognizes them because they have no prefix in the list of used external items.

The processor `items-extfuns`

The processor `items-extfuns` is used to generate the list of identifiers of all used external items which have a derived function type as declared type. It is invoked as

```
items-extfuns <used external items file>
```

where the argument is the list of external toplevel items used by the Cogent compilation unit, as it is created by `items-used`.

The processor reads the source files of the Cogent compilation unit as described in Section 3.6.4 and processes all external toplevel items specified in the list of used items in the same way as `items-externs`. Instead of generating default property declarations for them, it lists the identifiers of all function items on standard output. This list is intended for selecting parameter modification descriptions for these functions. It is produced by the monadic action

```
functionsInGlobals :: [DeclEvent] -> FTrav [String]
```

defined in module `Gencot.Items.Translate`.

It is not possible to directly use the list of identifiers of the used external toplevel items for this purpose, since functions can be sub-items of toplevel items and the resolved external typedef names have to be taken into account.

The result must include all functions processed by `gencot-externs` and all function types and composite type members processed by `gencot-exttypes`.

Although `gencot-externs` only processes complete function declarations of non-variadic external functions, `items-extfuns` also lists identifiers for incompletely declared functions and for variadic functions. This is intended as an information for the developer who has to translate invocations of such functions manually.

For declared functions and for all kinds of function pointers the function type can be specified in C using a typedef name. Such function items are not listed, instead, the typedef name is listed as the collective item for which the parameter modification description is required. If, however, the external typedef name is not used diectly by the Cogent compilation unit, it is resolved, and the item for which the typedef name has been declared is listed, as described in Section 2.11.4.

Manually specified additional external functions are also listed, since they have been added by `items-used` to the `<used external items file>`.

3.11.10 Parameter Modification

The goal of working with the parameter modification descriptions (see Section 2.11) is to analyse transitive function invocation chains which only stop at functions external to the C <package>, for which no sources are available. Therefore the approach used for item property declarations in Section 3.11.9 cannot be used here, since it treats all items as external which do not belong to the Cogent compilation unit.

Instead, we use a single filter `parmod-gen` which is always applied to a single C source file in the <package>. It may be run in two modes: in normal mode it processes all functions *defined* in the source file and outputs descriptions for them, in "closing mode" it processes all functions only *declared* in the source file or in a file included by it and outputs a description template. This covers all functions which may be invoked by the functions defined in the source, including functions external to the <package>.

The invocation chains can then be followed using the descriptions generated in normal mode, until a function is invoked which is external to the <package>. Only for these functions the templates generated in closing mode are used to "close" the invocation chains, all other templates (in particular, for functions defined in the <package>) are ignored.

The filter still needs the information about the used items outside the Cogent compilation unit to correctly handle typedef names which are used only indirectly and are resolved during the translation to Cogent. If the type of an invoked function is specified by a type name, the invocation either refers to the type name or to individual function items, depending on whether the type name is resolved, as described in Section 2.11.

`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> <used external items file>
parmod-gen <source file name> <used external items file> 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 a type syntactically including a function type (called a "SIFT" type in the following). 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 items with a SIFT type and all parameters with a SIFT type.

The filter processes every definition of a composite type in the source file and translates all its members with a SIFT type to the corresponding description template. The filter processes every type name definition in the source file where the defining type is a SIFT type, and translates it to the corresponding description template.

The descriptions of functions, their function parameters, composite type members, and typedef names are intended to be converted to item property declarations for being used by `gencot-translate` when it translates the function and type definitions. The descriptions of invoked local items are intended for evaluating dependencies as described in Section 3.3.

In closing mode the filter processes all `DeclEvent`s in the source file and in all included files which are declarations for functions or items with a SIFT type. Every declaration is translated to an entry in the parameter modification template. The filter also processes all composite type definitions and type name definitions which are specified in system include files in the same way as

described above. These templates are only intended for "closing" the dependencies for the evaluation.

Here, resolved external type names are taken into account, when the type of a declaration is tested for being a SIFT type: all such type names are resolved before testing whether a derived function type directly occurs in the type.

After analysing the C code as described in Section 3.6 the call graph is generated as described in Section 3.6.15. 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 3.3.4.

3.11.11 Retrieving Type Information from Cogent-Generated C

Retrieving the type information as described in Section 3.8 is implemented by two filters `auxcog-ctypstruct` and `auxcog-ctypfunc`. They are implemented as Haskell programs. They take the preprocessed `.h` file generated by Cogent as input and write the textual type information as described in Section 3.8.1 to their output.

The filter `auxcog-ctypstruct` generates the information about struct types, as described in Section3.8.3, the filter `auxcog-ctypfunc` generates the information about function types, as described in Section3.8.4.

3.12 Other Components

The following auxiliary Gencot components exist which do not parse C source code:

`parmod-proc` processes parameter modification descriptions in JSON format (see Section 3.3).

`items-proc` processes item property declarations in text format (see Section 3.2).

`auxcog-remcomments` processes Cogent source code to remove all comments.

`auxcog-macros` (deprecated) processes Cogent source code to select macro definitions.

`auxcog-numexpr` (deprecated) processes Cogent source code to translate numerical expressions to C.

`auxcog-mapback` (deprecated) processes Cogent source code to map Cogent constants back to preprocessor constants.

`gencot-prclist` processes list files to remove comments.

`auxcog-shalshared` processes Isabelle theory code to modify array types.

`auxcog-shalgencot` processes Isabelle theory code to generate interpretations for array types.

`auxcog-lemmgencot` processes Isabelle theory code to generate interpretations and lemmas for array types.

`auxcog-genops` processes a text file for expanding all Genops templates (see Section 3.9).

`gencot-mainfile` processes a list of C files to generate the main file of the Cogent source.

`auxcog-mainfile` processes a list of C files to generate the main file of the generated C source.

3.12.1 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 3.3.4. 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 3.3.6):

`check` Verify the structure of the parameter modification description according to Section 3.3.1 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.

`out` Using the function `convertParmods` the parameter modification description is converted to an item property map. The map is written to the standard output in the format described in Section 3.2.4.

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 3.3.4.

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 or heap use specifications 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.

3.12.2 Item Property Declarations

Processing item property declarations is implemented by the filter `items-proc`. It reads item property declarations from standard input as a sequence of text lines in the format described in Section 3.2.4. The first command line argument acts as a command how to process the item property declarations. The filter implements the following commands:

`merge` Takes an additional file name as command line argument. The file contains additional item property declarations. Both declarations are merged. If both contain properties for the same item the union of properties is declared for it. If the union contains negative properties they are removed together with all positive occurrences of the same property.

`idlist` Outputs the list of item identifiers for all toplevel items occuring in the declaration.

`filter` Takes an additional file name as command line argument. The file contains a list of item identifiers. The declarations are filtered, only declarations for those items remain where a prefix of the identifier is in the item identifier list. Usually, this command is applied to find the property declarations for all subitems of a given list of toplevel item identifiers.

3.12.3 Processing Cogent Code

There are some Gencot components which process a Cogent source to generate auxiliary files in antiquoted C or normal C code or other output. All these components are invoked with the name of a single Cogent source file as argument. Additionally, directories for searching files included by cpphs can be specified with the help of `-I` options. The result is always written to standard output.

`auxcog-remcomments`

This component is a filter which removes comments from Cogent sources. Note that the Cogent preprocessor `cpphs` does not remove comments, so it cannot be used for this task.

The filter is implemented as an awk script.

3.12.4 Using Macros in Generated C Code

The components described here support the transferral of some preprocessor macro definitions from the Cogent source to the generated C source. This feature has been used when array types have been mapped to abstract Cogent types. Since array types are now mapped to Cogent builtin array types, it is no more required and has been deprecated. For information reasons the concept is still documented here.

Note that in a definition

```
type CArrX<id>X el = {arrX<id>X: el#[<id>]}
```

using a builtin array type the macro name `<id>` is still present in the type name on the left side, however, its second occurrence on the right side will be expanded, so the numerical value of the array size is known and used by the Cogent compiler when it generates the C source.

`auxcog-macros`

Gencot array types may have the form `CArrX<id>X` where `<id>` is a preprocessor constant (parameterless macro) identifier specifying the array size (see Section 2.6.5). If these types are implemented using abstract types in Cogent, they must be implemented in antiquoted C. The implementation has the form

```
typedef struct $id:(UArrX<id>X el) {
  $ty:el arrX<id>X[<id>];
} $id:(UArrX<id>X el);
```

where the identifier `<id>` occurs as size specification in the C array type. Cogent translates this antiquoted C definition to normal C definitions for all instances of the generic type used in the Cogent program. These C definitions are included in the C program generated by Cogent. Thus the definition of `<id>` must be available when the C program is processed by the C compiler or by Isabelle.

The definition of `<id>` may depend on other preprocessor macros, it may even depend on macros with parameters. Therefore all macro definitions must be extracted from the Cogent source and made available as auxiliary C source. This is implemented by `auxcog-macros` with the help of `cpp` and an `awk` script.

The macro definitions are extracted from the Cogent source `file.cogent` by executing the C preprocessor as

```
cpp -P -dD -I<dir1> -I<dir2> ... -imacros file.cogent /dev/null 2> /dev/null
```

The option `-imacros` has the effect that `cpp` only reads the macro definitions from `file.cogent`. The actual input file is `/dev/null` so that no other input is read. The option `-dD` has the effect to generate as output macro definitions for all known macros. This will also generate line directives, these are suppressed by

the option `-P`. The `-I` options must specify all directories where Cogent source files are included by the preprocessor. All uses of the unbox operator `#` in the Cogent code are signaled as an error by `cpp`, but the output is not affected by these errors. The error messages are suppressed by redirecting the error output to `/dev/null`.

Note that `cpp` interpretes all include directives and conditional directives in the Cogent source. If for a macro different definitions occur in different branches of a conditional directive, depending on a configuration flag, `cpp` will select the definition according to the actual flag value.

The C preprocessor also adds definitions of internal macros. Some of them (those starting with `__STDC`) are signaled as redefined when the output is processed again by `cpp`. This is handled by postprocessing the output with an `awk` script. It removes all definitions of macros starting with `__STDC`.

Since `cpp` does not recognize the comments in Cogent format, it will not eliminate them from the macro definitions in the output. Therefore it should be postprocessed by `auxcog-remcomments`. For a better result it should also be postprocessed by `auxcog-numexpr`.

`auxcog-numexpr`

The macro definitions selected by `auxcog-macros` are normally written as part of the Cogent program to expand to Cogent code. However, to be included in the C code they must expand to valid C code. This must basically be done by the developer of the Cogent program.

The filter `auxcog-numexpr` supports this by performing a simple conversion of numerical expressions from Cogent to C. Numerical expressions are mostly already valid C code. Currently, the filter only removes all occurrences of the `upcast` operator.

The filter is implemented as a simple `sed` script.

`auxcog-mapback`

The `auxcog-mapback` component supports this by converting specific kinds of Cogent code back to C. It handles the use of Cogent constants in integer value expressions. This occurs in code generated by Gencot: every reference to another preprocessor constant in the definition of a preprocessor constant is translated to a reference of the corresponding Cogent constant.

This is handled by including another C source file before the file generated by `auxcog-macros`. This file is generated by `auxcog-mapback`. It contains a preprocessor macro definition for every Cogent constant where the constant name is a mapped preprocessor constant name, which maps the Cogent constant name back to the preprocessor constant name. For example, if the preprocessor constant name `VAL1` has been mapped by Gencot to the Cogent constant name `cogent_VAL1`, the generated macro definition is

```
#define cogent_VAL1 VAL1
```

The component `auxcog-mapback` is implemented in Haskell and uses the Cogent parser to read the Cogent code. It then processes every Cogent constant definition.

3.12.5 Processing List Files

Gencot uses some files containing simple lists of text strings, one in each line: the item property declaration list (see Section 3.11.9), and the list of used external items to be processed by `gencot-externs`, `gencot-exttypes`, and `gencot-dvdtypes`.

To be able to specify these lists in a more flexible way, Gencot allows comments of the form

```
# ...
```

where the hash sign must be at the beginning of the line.

The filter `gencot-prclist` is used to remove these comment lines (and also all empty lines). It is implemented as an awk script.

3.12.6 Processing Isabelle Code

Gencot processes Isabelle code as described in Section 3.10. The extension of the shallow embedding (see Section 3.10.2) is implemented with the help of the filters `auxcog-shalgencot` and `auxcog-shalshared`. Both filters take as input an Isabelle theory generated by Cogent in file `X_ShallowShared_Tuples.thy` or `X_ShallowShared.thy` for the proof name `X`. In the first case the filters generate output for the shallow embedding, in the second case for the intermediate representation described in Section 3.10.4. The filters determine what to do from the theory name.

The reasoning support described in Section 3.10.3 is implemented with the help of the filter `auxcog-lemmgencot` which takes as input an Isabelle theory generated by Cogent in file `X_ShallowShared_Tuples.thy`.

auxcog-shalshared

This filter takes as additional argument the name of a file containing the textual struct type information which has been generated by filter `auxcog-ctypstruct` (see Section 3.11.11). It uses this information to determine the size of array types which have been defined using a constant name as size specification.

This filter replaces some type declarations and type synonyms in its input and writes the modified Isabelle theory to standard output. The folllowing type mappings are performed:

- types of the form `'a MayNull` are mapped to `'a option`,
- types of the form `'el CArrPtr`$_T$ are mapped to `'el list`,
- types of the form `'el CArr<size>`$_T$ are mapped to `(<nr>, 'el) FixedList CArr<size>` where `<nr>` is the explicit array size according to `<size>`.

The filter is implemented as an awk script.

auxcog-shalgencot

This filter takes as additional argument the name of a file containing the Gencot standard components list (see Section 3.12.8). It takes as input a theory generated by `auxcog-shalshared`. It generates the Isabelle theory to be stored

in file `X_Shallow_Gencot_Tuples.thy` or `X_Shallow_Gencot.thy`, respectively, and writes it to standard output.

The filter determines the theories to be imported from the entries `"gencot: <name>"` in the Gencot standard components list. These entries name the standard include files which are part of the Gencot distribution. Some of them have no corresponding theory, since they do not define abstract functions. Therefore `auxcog-shalgencot` has an internal list of the existing theories and generates imports only for those entries where a corresponding theory exists.

From its input the filter determines the array sizes for which locale interpretations must be created.

The filter is implemented as an awk script.

auxcog-lemmgencot

This filter works similar as `auxcog-shalgencot`. It takes as input a theory generated by `auxcog-shalshared`. It generates the Isabelle theory to be stored in file `X_Shallow_Gencot_Lemmas.thy` and writes it to standard output. It cannot be used for the intermediate representation in `X_Shallow_Gencot.thy`.

The filter is implemented as an awk script.

3.12.7 Processing Genops Templates

The component for implementing the Genops mechanism described in Section 3.9.1 processes C code but is not implemented using a C parser, therefore it is described here. Genops is implemented by the filter `auxcog-genops`. It takes as input a single C source file which is the result of processing an antiquoted C file by the Cogent compiler using `-infer-c-funcs`. However, the input is not parsed as C code, it can be arbitrary text. The filter processes all embedded Genops templates as described in Section 3.9 and replaces them by the result. It also processes the `genopsInitClearDefinitions` marker and replaces it by the definitions of all required initialization and clearing functions, as described in Section 3.9.4. The modified C code is written to the output.

The filter takes as additional arguments the names of two files containing the struct type information and the function type information as described in Section 3.8.

The filter is implemented as a Haskell program, the Genops specific functions are implemented in module `Gencot.Text.Genops`. To extend Genops by additional templates, implement their expansion there.

3.12.8 Generating Main Files

The main files as described in Section 2.1.3 are `<unit>.cogent` and `<unit>.c`. They contain include directives for all parts of the Cogent unit sources, the former for the Cogent sources, the latter for the C sources resulting from translating the Cogent sources with the Cogent compiler.

The content of these main files depends on the set of `.c` files which comprise the Cogent compilation unit. A list of these files, as described in Section 3.6.4, is used as input for the components generating the main files. In contrast to the components reading the package C code, these components only process the file names and not their content. The file name `additional_includes.c` is ignored

in the list, it can be used to specify additional `.h` files for Gencot components reading the C code, which are not used in the Cogent code or after compiling Cogent back to C. For an application see Section 4.7.2.

The content of the main files also depends on the additional standard library components used from the Gencot distribution or the Cogent distribution. These components must be explicitly specified by the "Gencot standard components list" which is a sequence of lines of the form

```
<kind>: <name>
```

where `<kind>` may be one of `gencot`, `common`, or `anti`. In the first case the file `include/gencot/<name>.cogent` from the Gencot distribution is included in the Cogent program, in the second case the file `lib/gum/common/<name>.cogent` from the Cogent distribution is included in the Cogent program, and in the third case the file `lib/gum/anti/<name>.ac` from the Cogent distribution is processed by Cogent so that it becomes a part of the Cogent compilation unit.

Generating the Cogent Main File

The component `gencot-mainfile` generates the content of the Cogent main file `<unit>.cogent`. It is implemented as a shell script. It reads the list of `.c` files comprising the Cogent compilation unit as its input and it takes the unit name `<unit>` and the name of a file containing the Gencot standard components list as additional arguments. The generated content is written to the standard output.

The component adds include directives for the following files:

- The files to be included according to the Gencot standard components list.
- The file `x.cogent` for every source file `x.c` specified as input.
- The files `<unit>-externs.cogent`, `<unit>-exttypes.cogent`, `<unit>-dvdtypes.cogent`.
- The file `<unit>-manabstr.cogent`, if it exists in the current directory. This file can be used to manually define additional abstract types and functions used in the translated program.

The files `x.cogent` must be included first, because they may contain definitions of preprocessor macros which may also be used in the remaining files.

Generating the C Main File

The component `auxcog-mainfile` generates the content of the C main file `<unit>.c`. It is implemented as a shell script. It reads the list of `.c` files comprising the Cogent compilation unit as its input and it takes the unit name `<unit>` and the name of a file containing the Gencot standard components list as additional arguments. The generated content is written to the standard output.

The component adds include directives for the following files:

- The file `<unit>-gencot.h` for non-generic abstract standard types used by Gencot.

- The files `<unit>-dvdtypes.h` and `<unit>-exttypes.h`, if they exist in the current directory. They contain definitions for the non-generic abstract types defined in `<unit>-dvdtypes.cogent` and `<unit>-dvdtypes.cogent`, respectively.
- The file `<unit>-manabstr.h`, if it exists in the current directory. This file can be used to manually provide definitions for the non-generic abstract types defined in `<unit>-manabstr.cogent`.
- The file `<unit>-gen.c` which must be the result of the translation of `<unit>.cogent` with the Cogent compiler.
- The file `<unit>-externs.c` with the implementations of all exit wrappers.
- The file `<unit>-gencot.c`, if it exists in the current directory. It contains the implementations of all generic abstract functions the program uses from the Gencot standard library.
- The file `<unit>-manabstr.c`, if it exists in the current directory. This file can be used to manually provide implementations of the abstract functions defined in `<unit>-manabstr.cogent`.
- The file `<unit>-cogent-common.c` with implementations of functions defined in `gum/common/common.cogent` but not implemented by Cogent.
- The file `x-entry.c` for every source file `x.c` specified as input. These files contain the implementations of the entry wrappers.
- The file `std-<name>.c` for every line `"anti:  <name>"` specified in the Gencot standard components list.

The files `<unit>-*.c` and `*-entry.c` result from postprocessing the corresponding `_pp_inferred.c` files by `auxcog`. The files `std-<name>.c` result from renaming the files `<name>_pp_inferred.c` which have been generated by Cogent by processing `<name>.ac` as antiquoted C code.

3.13 Putting the Parts Together

The single filters and processors of Gencot are combined for the typical use cases in the shell scripts `gencot`, `items`, `parmod`, and `auxcog`. The script `items` is intended for handling all aspects of working with the item property declarations (see Section 3.2), the script `parmod` is intended for handling all aspects of working with the parameter modification descriptions (see Section 3.3), the script `gencot` is intended for handling all other aspects of translating from C to Cogent. The script `auxcog` is intended for handling all aspects of additional processing after or in addition to processing the Cogent program by the Cogent compiler.

3.13.1 The `gencot` Script

Usage

The overall synopsis of the `gencot` command is

```
gencot <options> <subcommand> [<file>]
```

The `gencot` command supports the following subcommands:

`check` Test the specified C source file or include file for parsability by Gencot (see Section 4.1 for how to make a C source parsable, if necessary).

`cfile` Translate the specified C source file to Cogent and generate the entry wrappers. If the source file name is `x.c` the results are written to files `x.cogent` and `x-entry.ac`.

`hfile` Translate the specified C include file to Cogent. If the source file name is `x.h` the result is written to file `x-incl.cogent`

`config` Translate the specified C include file to Cogent with specific support for configuration files as described in Section 3.5.5. If the source file name is `x.h` the result is written to file `x-incl.cogent`

`unit` Generate additional Cogent files for the Cogent compilation unit.

`cgraph` Print the call graph for the Cogent compilation unit.

The input file `<file>` must be specified for the first four subcommands and it must be omitted for the latter two. The subcommands `unit` and `cgraph` use the "unit name" to determine the files to be processed. If the unit name is `x` the subcommand `unit` generates the following additional files (see Section 2.1.3):

- `x.cogent`
- `x-externs.cogent`
- `x-exttypes.cogent`
- `x-dvdtypes.cogent`

The `gencot` command supports the following 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` Directory for searching Gencot auxiliary files. Only one can be given, default is `"."`.

`-C` Directory for retrieving stored declaration comments. Default is `"./comments"`.

`-u` The unit name. Default is `"all"`.

`-k` Keep directory with intermediate files. This is only intended for debugging.

`-t` Translation configuration string. Default is `"-"`. This is mainly intended for debugging.

As described in Section 3.4.4, Gencot moves comments from C function declarations to translated function definitions. Since a declaration may be specified in a different C source file (often an include file) than the definition, the subcommands `cfile`, `hfile`, and `config` write all declaration comments in the processed file to the directory specified by the `-C` option and use all comments found there when translating function definitions.

The translation of C function bodies can be configured using the `-t` option by specifying the translation configuration string described in Section 3.11.3. The empty string corresponds to `"-"` which is the default when the option is omitted.

Auxiliary Files

Gencot uses an approach, where all manual annotations added to the C program for configuring the translation to Cogent are stored in auxiliary files separate from the original C program sources. This way it is possible to update the C sources to a newer version without the need to re-insert the annotations.

There are different auxiliary files for different purposes. The `gencot` script determines the auxiliary files using a predefined naming scheme. These files are searched in the directory specified by the `-G` option. All auxiliary files are optional, if a file is not found it is assumed that no corresponding annotations are needed.

When processing a file `x.<ext>` where `<ext>` is `c` or `h`, let `y` be `x-incl` when `<ext>` is `h` and `x` otherwise. The `gencot` command uses the following auxiliary files, if they exist:

`y.gencot-addincl` The content of this file is prepended to the processed file before processing.

`y.gencot-noincl` List of ignored include files, as described in Section 3.6.1.

`y.gencot-omitincl` The Gencot include omission list (see Section 2.4.4).

`y.gencot-ppretain` The Gencot directive retainment list (see Section 2.4.1).

`y.gencot-chsystem` The content of this file is inserted immediately after the last system include directive.

`y.gencot-manmacros` The Gencot manual macro list (see Section 2.4.3).

`y.gencot-macroconv` The Gencot macro call conversion (see Section 2.4.3).

`y.gencot-macrodefs` The Gencot macro translation for the processed file (see Section 2.4.3).

For all cases with the exception of the last, additionally a file with name `common` instead of `y` ist used, when it is present. If both files are present the concatenation of them is used. In this way it is possible to specify common annotations for all C sources and add specific annotations in the source file specific files.

The subcommands `unit` and `cgraph` use corresponding auxiliary files for every file `x.c` processed by them.

The subcommand `unit` additionally uses the auxiliary file `common.gencot-std` which must contain the Gencot standard components list, as described in Section 3.12.8.

The subcommands `cfile`, `hfile`, `config`, and `unit` additionally use the auxiliary file named `<file>-itemprops` which must contain all relevant item property declarations (see Section 3.2.4).

These subcommands also use the auxiliary file `common.gencot-namap` which must contain the name prefix map, as described in Section 3.6.7. No source file specific versions of the name prefix map are supported, since the map is used globally for all names translated to Cogent.

Another kind of auxiliary files describe the Cogent compilation unit. These files are required to be present in the directory specified by the `-G` option. If they are not found, an error or a warning is signaled.

If the unit name is `<uname>` the auxiliary files for the Compilation unit are

`<uname>.unit` The "unit file". It contains the list of C source file names (one per line) which together constitute the Cogent compilation unit. Only the original source files `x.c` must be listed, include files `x.h` must not be listed.

`<uname>-external.items` The list of used external items, as generated by the processor `items-used` (see Section 3.11.2).

The first file is only used by the subcommands `unit` and `cgraph`. They process all files listed in the unit file together with all included files. The second file is used by all `gencot` subcommands with the exception of `check`.

The subcommand `unit` also processes the following optional auxiliary file by generating an include directive for it in the main Cogent source `<uname>.cogent`, if the file exists:

`<uname>-manabstr.cogent` This file is intended for manually specifying additional abstract functions used in the Cogent program.

Implementation

todo

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.

3.13.2 The `items` Script

Usage

The overall synopsis of the `items` command is

```
items <options> <subcommand> [<file>] [<file2>]
```

The `items` command supports the following subcommands:

`file` Create default property declarations for all items defined in the specified C source file or include file `<file>`.

`unit` Create default property declarations for all external items used in the Cogent compilation unit.

`used` List external toplevel items used in the Cogent compilation unit.

`merge` Merge the declarations in `<file>` and `<file2>`. Two declarations for the same item are combined by uniting the properties and removing negative properties.

`mergeto` Add properties from `<file2>` to items in `<file>`. Properties are only added/removed if their toplevel item is already present in `<file>`.

If the unit name is `<uname>`, subcommand `used` writes its output to the file `<uname>-external.items` The presence of this file is expected by most other Gencot command scripts. All other subcommands write their result to standard output.

For the subcommands `unit` and `used` no `<file>` must be specified, instead, they use the unit name in the same way as the subcommand `unit` of the `gencot` command.

The `items` command supports the options `-I`, `-G`, `-u`, `-k` with the same meaning as for the `gencot` command.

Auxiliary Files

The subcommand `file` uses the same auxiliary files as the subcommands `cfile` and `hfile` of `gencot` with the exception of `common.gencot-namap` and `<file>-itemprops`. The subcommand `unit` uses the same auxiliary files as the subcommand `unit` of `gencot` with the exception of `common.gencot-namap` and `<file>-itemprops`. The subcommand `used` uses the unit file `<uname>.unit` like subcommand `unit` and additionally the optional auxiliary file

`<uname>.unit-manitems` Manually specfied external items, as described in Section 3.11.2 as input to the processor `items-used`.

Implementation

** todo **

The `unit` command is implemented by the processor `items-externs` (see Section 3.11.9). The postprocessing step for removing declarations of non-external subitems is implemented by the filter `items-proc` with command `filter` (see Section 3.12.2).

3.13.3 The `parmod` Script

Usage

The overall synopsis of the `parmod` command is

```
parmod <options> <subcommand> <file> [<file2>]
```

The `parmod` command supports the following subcommands:

`file` Create parameter modification descriptions for all functions defined in C source file or include file `<file>`. The result is written to standard output.

`close` Create parameter modification descriptions for all functions declared for the C source file `<file>` in the file itself or in a file included by it. The result is written to standard output.

`unit` Select entries from `<file>` (a file in JSON format containing parameter modification descriptions) for all external functions used in the Cogent compilation unit. The result is written to standard output.

`show` Display on standard output information about the parameter modification description in `<file>`.

`idlist` List on standard output the item identifiers of all functions described in the paramer modification description in `<file>`.

`diff` Compare the parameter modification descriptions in `<file>` and `<file2>`. The output has the same form as the Unix `diff` command, however, entries of functions occurring in both files are directly compared.

`iddiff` Compare the item identifiers of all functions described in `<file>` and `<file2>`. The output has the same form as the Unix `diff` command.

`addto` Add to `<file>` all entries for required dependencies found in `<file2>`. Both files must contain parameter modification descriptions in JSON format. The result is written to `<file>`.

`mergin` Merge the parameter modification description entries in `<file>` and `<file2>` by building the union of the described functions. If a function is described in both files the entry with more confirmed parameter descriptions is used. The result is written to `<file>`.

`replin` Replace in `<file>` all function entries by an entry for the same function in `<file2>` if it is present and has not less confirmed parameters. Both files must contain parameter modification descriptions in JSON format. The result is written to `<file>`.

`eval` Evaluate the parameter modification description in `<file>` as described in Section 3.3.5. The resulting parameter modification description is written to standard output.

`out` Convert the evaluated parameter modification description in `<file>` to an property declarations. The result is written to the file `<file>-itemprops`.

The subcommand `unit` expects no argument `<file>`, instead, it uses the unit name in the same way as the subcommand `unit` of the `gencot` command.

The `parmod` command supports the options `-I`, `-G`, `-u`, `-k` with the same meaning as for the `gencot` command. They are only used for the subcommands `file`, `close`, and `unit`.

Auxiliary Files

The subcommands `file` and `close` use the same auxiliary files as the subcommands `cfile` and `hfile` of `gencot` with the exception of `common.gencot-namap` and `<file>-itemprops`. The subcommand `unit` uses the same auxiliary files as the subcommand `unit` of `gencot` with the exception of `common.gencot-namap` and `<file>-itemprops`.

Implementation

The subcommand `unit` First, all C source files are prepared for parsing, as in command `gencot unit`. Then the list of used external toplevel items `<uname>-external.items` is passed to processor `items-extfuns` to determine the list of used external functions. This list is then used to filter the descriptions in `<file2>` with `parmod-proc filter`.

Additionally the result is tested, whether all external functions determined by `items-extfuns` have been found in `<file2>`. This is done by calculating the function ids of all descriptions in the result with `parmod-proc funids`, sorting both lists and comparing them with `diff`. If the result is not empty, a warning is displayed on standard error.

3.13.4 The `auxcog` Script

Usage

The overall synopsis of the `auxcog` command is

```
auxcog <options> <subcommand> [<file>]
```

The `auxcog` command supports the following subcommands:

`unit` Generate additional C files for the Cogent compilation unit.

`shallow` Extend the shallow embedding generated by Cogent.

`refine` Extend the refinement proof generated by Cogent.

`comments` Remove comments from the Cogent source `<file>` and write the result to standard output.

The subcommand `unit` expects no argument `<file>`, instead, it uses the unit name in the same way as the subcommand `unit` of the `gencot` command. It expects all C files generated by Cogent in the current directory. In particular, if the unit name is `x`, it expects files `<f>_pp_inferred.c` for the following cases of `<f>`:

- `x-externs`

- `y-entry` for all `y.c` listed in the unit file `x.unit`
- `std-<name>` for all entries `"anti:  <name>"` in file `common.gencot-std`
- optionally: `x-gencot`
- optionally: `x-manabstr`

If the unit name is `x` the subcommand `unit` generates the following additional files in the output directory (see Section 2.1.3):

- `x.c`
- `x-externs.c`
- `x-gencot.h` and `x-cogent-common.c` (copied and renamed from the Gencot distribution).
- `x-gencot.c` (only if `x-gencot_pp_inferred.c` is present)
- `x-manabstr.c` (only if `x-manabstr_pp_inferred.c` is present)

and for all files `y.c` listed in the unit file `x.unit`:

- `y-entry.c`

and for all entries `"anti:  <name>"` in file `common.gencot-std` the file

- `std-<name>.c`

The subcommands `shallow` and `refine` expect as `<file>` a file named `X_ShallowShared_Tuples.thy` which has been generated by Cogent for a "proof name" `X`. Subcommand `shallow` expects the file `X_Shallow_Desugar_Tuples.thy` in the same directory as specified for `<file>`. In the output directory the following files are provided (see Section 3.10.2):

- `X_ShallowShared_Tuples.thy` (modified)
- `X_Shallow_Desugar_Tuples.thy` (copied from the directory of `<file>` and modified)
- `ShallowShared_Tuples.thy`
- `X_Shallow_Gencot_Tuples.thy`
- file `GencotTypes.thy` and all files `Y_Tuples.thy` in directory `isa` in the Gencot distribution
- `X_Shallow_Gencot_Lemmas.thy`
- all files `Y_Lemmas.thy` in directory `isa` in the Gencot distribution

The subcommand `refine` additionally expects the file `X_Shallow_Desugar.thy` in the same directory as specified for `<file>`. In the output directory the following files are provided (see Section 3.10.4):

- all files provided by `shallow` but the `...Lemmas.thy` files

- `X_ShallowShared.thy` (modified)
- `X_Shallow_Desugar.thy` (copied from the directory of `<file>` and modified)
- `ShallowShared.thy`
- `X_Shallow_Gencot.thy`
- file `GencotTypes.thy` and all files `Gencot_TTT.thy` and `CogentCommon_ttt.thy` in directory `isa` in the Gencot distribution
- file `X_ShallowTuplesProof.thy` (copied from the directory of `<file>` and modified)
- ... more to come

The subcommand `comments` is mainly intended for testing the comment structure in a Cogent source generated by `gencot`. In rare cases it may happen that a closing comment delimiter and a subsequent opening comment delimiter are omitted, so that the Cogent compiler will not detect the wrong comment structure.

The `auxcog` command supports the options `-I`, `-G`, `-u` with the same meaning as the `gencot` command. Additionally it supports the option `-O` for specifying the output directory (default `"."`) for the subcommands `unit`, `shallow` and `refine`.

Auxiliary Files

The subcommands `unit`, `shallow`, and `refine` use the auxiliary file `common.gencot-std` which must contain the Gencot standard components list, as described in Section 3.12.8. The subcommand `unit` additionally uses the unit file `<uname>.unit` as auxiliary file. Both files are expected in the directory specified by the `-G` option.

Additionally the subcommands `unit`, `shallow`, and `refine` also read the file `<uname>-gen.h` which must have been created by Cogent when compiling file `<uname>.cogent` with option `"-o <uname>-gen"`. This file is always expected in the current directory. Files included by `<uname>-gen.h` are searched in the directories specified by the `-I` options.

Implementation

todo

The subcommand `shallow` is implemented by applying the filter `auxcog-shalshared` to the content of file `X_ShallowShared_Tuples.thy` to generate the modified file `X_ShallowShared_Tuples.thy`. Then the filters `auxcog-shalgencot` and `auxcog-lemmgencot` are applied to this modified file to generate the files `X_Shallow_Gencot_Tuples.thy` and `X_Shallow_Gencot_Lemmas.thy`. The file `X_Shallow_Desugar_Tuples.thy` is directly modified using `sed` (only replacing the theory import). The remaining files are simply copied from the Gencot distribution.

The subcommand `refine` is implemented by applying the filter `auxcog-shalshared` to the content of file `X_ShallowShared.thy` to generate the modified file `X_ShallowShared.thy`.

Then the filter `auxcog-shalgencot` is applied to this modified file to generate the file `X_Shallow_Gencot.thy`. The file `X_Shallow_Desugar.thy` is directly modified using `sed` (only replacing the theory import). The remaining files are simply copied from the Gencot distribution.

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. However, Gencot supports working with alternative translation bases at the same time, which result in different Cogent compilation units.

A translation base (and its resulting Cogent compilation unit) may be named by a "unit name" and is defined by a "unit file" `<uname>.unit` where `<uname>` is the unit name. 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 Determining Used External Items

Most other commands expect the presence of the auxiliary file `<uname>-external.items` (see Sections 3.13.1 and 3.11.2).

Therefore, after all sources in the translation set can be read by Gencot, the command

```
items used
```

must be used to create the list of used external items (see Section 3.13.2).

If there are external items which must be processed, but are not detected automatically by Gencot, the auxiliary file `<uname>.unit-manitems` must be provided in advance. If present, this file is automatically read by `items used`

and the items listed in it (and all items transitively used from them) are added to `<uname>-external.items`.

If additionally required external items become known after C code has already been translated to Cogent, they must be added to this file and the command `items used` and all following steps must be re-executed.

4.3 Building Parmod Descriptions

The goal of this step is to create the parameter modification descriptions (see Section 2.11) for the C sources. This step is optional, if it is omitted Gencot will translate all functions using the assumption that all parameters of linear type are discarded by their function. This will usually lead to invalid Cogent code if the parameter value is still used after a function invocation.

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 3.13.3).

4.3.1 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.

For every file in the translation base and for every other file in the translation set which contains function definitions, the command

```
parmod file
```

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 file`. 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 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 file` 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 yield the final parmod description file. It contains at least the descriptions for all functions defined in the translation set.

4.3.2 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. This is done by the command

```
parmod unit
```

applied to 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 file` 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 unit` 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 unit`.

Instead of starting with an empty description, all files created by `parmod file` in the first substep (see Section 4.3.1) can be merged and used as a starting point.

When the descriptions resulting from `parmod file` 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 file` 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 unit` has successfully been executed, the result must be iteratively completed in the same way as in Section 4.3.1.

Initially, however, the file `parmod-system.json` should be added using command

```
parmod replin <json-file> parmod-system.json
```

This file is included in the Gencot distribution and contains completed descriptions for a number of standard C system functions. In particular, it covers the five memory management functions, which are the only functions for which the heap is always used and from which heap usage is then derived for other functions. The command `parmod replin` replaces all templates in `<json-file>` by the description for the same function in `parmod-system.json`, if present there.

Afterwards, 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`.

4.4 Building ItemProperty Declarations

The goal of this step is to create the item property declarations (see Section 3.2) to be used in the translation from C to Cogent. This step is optional, Gencot can translate a C source without them. However, the assumptions it uses for the translation will in many cases lead to invalid Cogent code (which is syntactically correct but does not pass the semantics checks).

Instead of specifying the additional information used for the translation as annotations in the C source files, Gencot uses the item property declarations which reside in separate files and reference the affected C items by name (the "item identifiers" described in Section 3.2.1. This makes it easier to replace the C sources by a newer version and still reuse (most of) the item properties.

Item properties could be specified fully manually by the developer, if she knows the item naming scheme and searches the C source files for items which require properties to be declared. However, Gencot provides support by generating default item property declarations. This has the following advantages:

- The developer gets a list of all items with their correct names.
- Gencot can use simple heuristics to populate the declarations with useful default properties.

The default item property declarations are intended to be manually modified by the developer. Additionally, the results of a parameter modification analysis described in Section 4.3 are intended to be transferred to the item property declarations.

Property declarations are used by Gencot, if present, for all items processed in the translation. These are all internal items which are defined in the translation set and all external items which are declared and used in the translation set.

The automatic parts of this step are executed with the help of the script command `items` (see Section 3.13.2).

4.4.1 Grouping Item Property Declarations in Files

C items for which Gencot supports property declarations are types (struct/unions and named types), functions and objects ("variables"). Items may have subitems (function parameters and results, array elements, pointer targets).

For type items there is always a single definition (which also defines subitems). For functions and objects there can be declarations in addition to the single definition. The definition or declaration introduces a name for the item and it specifies its type. Afterwards, an item can be used any number of times (e.g. type items can be used in other definitions and declarations, functions can be invoked and objects can be read and written).

In most cases Gencot uses the properties of such items only when translating the item's definition or declaration. However, there are also cases where item properties are required when translating the item's use. For example, as described in Section 2.6.8, the generic type `MayNull` is applied to all *references* of a translated type name, depending on the Not-Null property of the type name. Therefore, when Gencot translates a C source file it needs the property declarations for all items defined or declared in the source file and all included files.

Gencot expects the item properties to be used when translating file `x.<e>` (where `<e>` is `c` or `h`) in an auxiliary file named `x.<e>-itemprops` (see Section 3.13.1).

Item properties are intended to be declared manually. For this it is useful to have a single place for every item where to expect its property declaration. For a small <package> a single common file would be feasible. To support also larger <package>s Gencot allows grouping property declarations according to the source files where the items are defined. The command `items file` (see Section 3.13.2) must be executed for every single source file and it generates the item identifier list with the default property declarations only for the items defined in the file itself.

To group item property declarations it is recommended to store the result of `items file` applied to `<file>` in a file `<file>-dfltprops`. Then manually added properties should be specified in a seperate file `<file>-manprops` so that they are not overwritten when re-creating the default properties.

When Gencot translates a single C source file it ignores all declarations and only processes item definitions and uses. However, when Gencot processes all C source files of the translation set together to generate the additional Cogent sources for the Cogent compilation unit (command `gencot unit`, see Section 3.13.1), it processes items defined in the translation set and also external items declared and used in the translation set. Therefore it needs property declarations for all these items, it expects them in the file `<ufile>-itemprops` where `<ufile>` is the unit file. These property declarations can be retrieved by uniting all item property declaration files for the single C sources and adding property declarations for the external used items.

The default item properties for the external used items are generated by command `items unit` (see Section 3.13.2) applied to a unit file `<ufile>`. To group item property declarations it is recommended to store the result in a file `<ufile>-dfltprops` and specify manually added properties in a file `<ufile>-manprops`.

Since the properties in the `-itemprops` files may originate from several sources (default properties, manual specification, parmod descriptions, merging other property declarations) it is recommended to keep these sources separate and recreate the `-itemprops` files from these sources whenever they are used for a translation.

Then the file `<file>-itemprops` for a single C source file is generated by first modifying `<file>-dfltprops` and all `<f>-dfltprops` for included files `<f>` and the unit file according to the parameter modification descriptions and then adding the properties from `<file>-manprops` and all `<f>-manprops` for included files `<f>` and the unit file. All resulting property declarations must be merged. The file `<ufile>-itemprops` can then be generated by merging the `x.c-itemprops` files for all files `x.c` listed in the unit file `ufile`.

Gencot also supports item property declarations for a restricted form of type expressions (derived pointer types). These items are not explicitly defined, they are used by simply denoting the type expression. For this kind of type items the properties are needed by Gencot for translating the item uses.

Gencot does not provide default item property declarations for these items, they must always be specified manually. Since they are not related to a specific C source file or a specific translation set, they should be put in a single separate file. The recommended name for this file is `common.types-manprops`. The properties in this file must be used for the translation of all single C source files and for the generation of the additional Cogent sources for the compilation

unit. This is achieved by merging the properties in this file to all generated `-itemprops` files.

4.4.2 Generating Default Item Property Declarations

The first step for using item property declarations is to generate the default property declarations. For a single C source file or include file `<file>` the command

```
items file <file>
```

is used. The result should be stored in file `<file>-dfltprops`.

To generate the default property declarations for the external items used by the Cogent compilation unit the command

```
items unit
```

is used. The result should be stored in file `<uname>.unit-dfltprops` where `<uname>` is the unit name.

4.4.3 Transferring Parameter Modification Descriptions

If parameter modification descriptions are used as described in Section 4.3, the result must be converted to item property declarations to be used by Gencot. The result consists of a single file `<jfile>` in JSON format containing parameter modification descriptions. All entries have been confirmed and the file has been evaluated as described in Section 4.3.2. The file contains entries for all functions defined in the translation set, for all external functions used in the translation set, and maybe additional functions which have been used to follow dependencies.

Every function is an item, and every parameter is a sub-item of its function. Thus the information in the parameter modification description can be converted to property declarations for function and parameter items. This is done using the command

```
parmod out <jfile>
```

It will create the file `<jfile>-itemprops`.

The file `<jfile>-itemprops` contains properties to be added to / removed from the default properties. For every file `<file>-dfltprops` the command

```
items mergeto <file>-dfltprops <jfile>-itemprops
```

is used. Its output should be stored in an intermediate file `<file>-ip1`. The command only selects from `<jfile>-itemprops` declarations for items where items with the same toplevel item are already present in `<file>-dfltprops`. In this way the collective information in `<jfile>-itemprops` is distributed to the different specific property declaration files for the single C sources and the file `<uname>.unit-dfltprops`. Information about functions only used for following dependencies is not transferred, it is not used by the translation to Cogent.

4.4.4 Adding Manually Specified Properties

It is recommended to put manually specified item properties in separate files, grouped in the same way as the default properties. For every file `<file>-dfltprops` a file `<file>-manprops` should be used. To create this file, you can make a copy of `<file>-dfltprops`, manually remove all declared properties and then populate the empty declarations with manually specified properties. Remaining empty declarations can be removed.

The manual properties are then merged to the default properties. This makes it possible to re-apply the manual properties whenever the default properties change (e.g., because the C source is modified). To avoid removing manually specified properties by transmitting parameter modification description results, it is recommended to merge the manual properties afterwards into the intermediate file `<file>-ip1` using the command

```
items mergeto <file>-ip1 <file>-manprops
```

and storing the result in another intermediate file `<file>-ip2`.

If properties shall be specified for type expression items, they should be collected in the file `common.types-manprops`. No default property declarations exist for these items. Since the type expressions may be used in all C source files, the file should be merged into *all* item property declaration files additionally to the file specific manual properties. This must be done using the command

```
items merge <file>-ip2 common.types-manprops
```

because here the added items shall not be reduced to those already present in `<file>-ip2`. The result should be written to an intermediate file `<file>-ip3`. Note that this file does not contain negative properties anymore, since they are processed and removed by `items mergeto` and `items merge`.

For convenience for the developer Gencot supports empty lines and comment lines starting with a hash sign `#` in the `-manprops` files. These lines are removed by the commands `items merge` and `items mergeto`.

4.4.5 Property Declarations for C Source Translations

The resulting intermediate files `<f>-ip3` must be merged to create the files `<file>-itemprops` used by Gencot when translating C sources. For a file `x.h` translated by `gencot hfile` or `gencot config` the file `x.h-ip3`, all `<f>-ip3` for (transitively) included non-system files `<f>`, and `<uname>.unit-ip3` must be merged. For a file `x.c` translated by `gencot cfile` the file `x.c-ip3`, all `<f>-ip3` for (transitively) included non-system files `<f>`, and `<uname>.unit-ip3` must be merged. For a unit file `<uname>.unit` processed by `gencot unit` all the property declarations required for all files `x.c` listed in `<uname>.unit` must be merged, which can be done by merging all corresponding files `x.c-itemprops`.

Merging item property files is done by iteratively using the command

```
items merge
```

This command removes all duplicate item declarations which may originate from earlier merging operations.

4.5 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 4.7.

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 and the used external items must be determined as described in Section 4.2. Then the parameter modification descriptions for all functions and function types defined in the source file should be created and evaluated as described in Section 4.3.1. Afterwards, the item property declarations must be provided as described in Section 4.4.

Finally, additional Cogent source files must be generated for the Cogent compilation unit.

4.5.1 Translating Normal C Sources

A `.c` file is translated using the command

```
gencot [options] cfile file.c
```

where `file.c` is the file to be translated. 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
```

where `file.h` is the file to be translated. The result of the translation is stored as `file-incl.cogent` in the current directory.

Note that the unit name must be specified with the `-u` option if it is not the default unit name `all`.

4.5.2 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
```

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.

4.5.3 Generating Additional Cogent Sources

The additional Cogent sources `<uname>-dvdtypes.cogent`, `<uname>-exttypes.cogent`, `<uname>-externs.cogent`, and `<uname>.cogent`, are generated using the command

```
gencot [options] unit
```

and stored in the current directory.

The unit name `<uname>` must be specified with the `-u` option if it is not the default unit name `all`. The unit file `<uname>.unit` must be present in the current directory (or the directory specified by the `-G` option) and it must contain the names of all `.c` files comprising the unit's translation base.

4.6 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.

4.6.1 NULL Pointers

A value of linear type in Cogent corresponds in C to a pointer which is not `NULL`. If a value of linear type 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. The natural choice for a linear type `T` is the type `Option T` using the Cogent standard library. However, this type is not binary compatible to `T`. If the value is also used outside the Cogent compilation unit in the remaining C program, the type `(MayNull T)` (see Section 2.7.10) must be used instead.

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 also be represented in a binary compatible way 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, a type name must be introduced for the `MayNull` type instance and then used to consistently rename the type in the function type encoding used for the function pointer 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
```

4.6.2 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 R = {f1: A1, f2: A2, f3: A3, f4: A4, f5:A5}
```

the fields `f2` and `f3` can be grouped by introducing the new record type `E`:

```
type E = {f2: A2, f3: A3}
type R = {f1: A1, embedded: #E, f4: A4, f5:A5}
```

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 accessing the fields in 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 polymorphic functions

```
getrefFldEmbedded: all(rec,pfld). rec! -> pfld!
modrefFldEmbedded: all(rec,pfld,arg,out). ModPartFun rec pfld arg out
```

and use their instances `getrefFldEmbedded[R,E]` and `modrefFldEmbedded[R,E,A,O]` for the required types `A` and `O`. The functions are implemented in antiquoted 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.

4.6.3 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: CPtr U32, ...}
```

and does not generate instances of the array access functions defined in Section 2.7.12.

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. If this way is translated to Cogent, an explicitly sized array (see Section 2.7.13) and the corresponding access functions provided by Gencot can be used, by constructing values of type `CArrES U32` on the fly as pairs `(p,<size>)` or using the macro `MKCAES`. The only additional requirement is to mark the pointer type as an array by replacing the generic type `CPtr` by the generic type `CArrPtr`:

```
type Struct_Cogent_ip = {... p: CArrPtr U32, ...}
```

This approach can be improved by reorganizing the type of the surrounding record `Struct_Cogent_ip`. We assume that it shall be binary compatible to the original C struct `ip` and 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 a 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 `CArr_tt_El` as type name for the array, thus the type of `p` must be manually changed as follows:

```
type Struct_Cogent_ip = {... p: CArr_tt_El, ...}
```

The type `CArr_tt_El` is defined as a synonym for the array pointer type

```
type CArr_tt_El = CArrPtr El
```

Since the array size is determined from the array content this will usually be specific for the type of elements, so a non-generic type is defined here.

For the type `CArr_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. Instead, the function `createCAES` can be used (see Section 2.7.13) by specifying the array size explicitly. The resulting explicitly sized array has type `EVT_CAES(,El)` which is equivalent to `EVT(CArrPtr El),U64)`, and the first component is equivalent to `EVT(CArr_tt_El)`. As usual, it denotes the type of the uninitialized array, realized in the same way as described for `CPtr`. Corresponding abstract functions for initializing and freeing the array elements must be provided manually.

For disposing values of the type `CArr_tt_El` the usual polymorphic function `dispose` can be used, Gencot is able to automatically provide a correct C implementation for all types for which it is used.

Element access functions for single elements can now 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. The type `UNN` used for the index values must be chosen by the developer so that it can represent all sizes which are used for the array.

```
getCArr_tt_El : (CArr_tt_El!,UNN) -> El!
getCArrChk_tt_El : (CArr_tt_El!,UNN) -> Result El! ()
setCArr_tt_El : ModFun CArr_tt_El (UNN,El) ()
exchngCArr_tt_El : ModFun CArr_tt_El (UNN,El) (UNN,El)
modifyCArr_tt_El : all(arg,out).
  ModFun CArr_tt_El (UNN, ChgFun El arg out, arg) out
getrefCArr_tt_El : (CArr_tt_El!,UNN) -> CPTR(<knd>,El)!
getrefCArrChk_tt_El : (CArr_tt_El!,UNN) -> Result CPTR(<knd>,El)! ()
modrefCArr_tt_El : all(arg,out).
  ModFun CArr_tt_El (UNN, ModFun CPTR(<knd>,El) arg out, arg) out
```

where `setCArray_tt_El` can only be defined if the element type `El` is discardable. All these functions must be defined as abstract and implemented in C, since they access elements outside the Cogent record for `CArrPtr El)`. They cannot use the corresponding access function for the explicitly sized array, because they already need to access array elements to determine the size.

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 4.6.2. 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 `SArr_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 SArr_tt_El = { m1: T1, ... Mn: Tn }
```

where the element pointer type `Ti` uses the generic type `CArrPtr` instead of `CPtr` or, providing additional shielding as a wrapped abstract type

```
type SArr_tt_El = { cont: #USArr_tt_El }
```

with a C definition:

```
typedef struct { t1 m1;...tn mn; } USArr_tt_El;
```

Note that in both cases `EVT(SArr_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: #SArr_tt_El, ...}
```

To access and modify the group in the struct in-place the abstract polymorphic functions

```
getrefFldA: all(rec,sarr). rec! -> sarr!
modrefFldA: all(rec,sarr,arg,out).
  ModPartFun rec sarr arg out
```

must be defined and used in the same way as described in Section 4.6.2.

Initialization and clearing functions for `SArr_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 `SArr_tt_El`. Since they do not need to access the array for determining its size they can always be implemented in Cogent by calculating the size using the fields of `SArr_tt_El`, then constructing an explicitly sized array and then using the access function provided by Gencot.

Usually, additional functions will be required for working with values of type `SArr_tt_El`, such as for moving an internal pointer to a "current" element. If such a function modifies some of the fields of `SArr_tt_El` it must be defined as a modification function which can be applied with the help of `modrefFldA`.

4.7 Manual Translation of C Function Body Parts

Manual actions in a function body translation are required if Gencot cannot translate a C code fragment. In this case it inserts a dummy expression, as described in Section 2.10.4.

In the following sections we provide some rules and patterns how to translate such cases manually. These rules and patterns are not exhaustive but try to cover most of the common cases.

4.7.1 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. Gencot always assumes that named C objects never share parts with the help of pointers. If this is not the case in the original C program, it must be manually modified. The following approach can be used 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`.

Alternatively, in Cogent a `take` operation can be used to bind the value of `p` to a variable `v`. This prevents accessing the value through `p1`. Before the end of the scope of `v` the value must be put back into `p1` using a `put` operation:

```
let p1 { p = v }
and ...
in p1 { p = v }
```

In other cases individual solutions must be found. Note that parameters and variables of nonlinear type never cause such problems.

4.7.2 Function Pointer Invocation

When a function pointer `fptr` shall be invoked in Cogent, it must be converted to the corresponding Cogent function using `fromFunPtr[...](fptr)` or the macro `FROMFUNPTR(<enc>)(fptr)` where `<enc>` is the encoding of the corresponding function type (see Section 2.7.9).

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 );
```

When applied to `NULL` the conversion function `fromFunPtr` returns a result which can be invoked in Cogent. To treat the `NULL` case separately it must be tested using the function `nullFunPtr`:

```
if nullFunPtr(fptr) then ...
else fromFunPtr(fptr)(params)
```

In Cogent the `then` case only covers the case where the function pointer is `NULL`, it does not cover the cases where the function referenced by the pointer is not known by Cogent. A function is only known by Cogent if there is a Cogent definition for it, either as a Cogent function or as an abstract function.

Function pointers are not needed for Cogent functions which are only invoked from Cogent, then the Cogent function can be assigned and stored directly. Function pointers are only relevant if they are also used in C code external to the Cogent compilation unit. So there are two application cases:

- A function which is defined in C and is passed as a function pointer to Cogent for invocation there. As described in Section 2.1.2, this invocation requires an exit wrapper. The function `fromFunPtr` converts the function pointer to the exit wrapper function. To be consistent, function `toFunPtr` converts every exit wrapper to a pointer to the wrapped external function.

- A function which is defined in Cogent and is passed as a function pointer to C for invocation there. As described in Section 2.1.2, this invocation requires an entry wrapper. Entry wrappers are automatically generated for all translated functions which had external linkage in C. For all these functions `toFunPtr` converts them to a pointer to the entry wrapper. To be consistent, function `fromFunPtr` converts pointers to entry wrappers to the corresponding Cogent functions. For translations of functions with internal linkage `toFunPtr` converts to a pointer to the Cogent function. If such a pointer is invoked from C the invocation will normally fail, because of different parameter and result types. In the current version of Gencot this must be handled manually.

Exit wrappers are automatically generated by Gencot for functions which are invoked from the Cogent compilation unit (see Section 3.11.5), determined with the help of the call graph (see Section 3.6.15). However, functions which are *only* invoked through function pointers are not detected by the call graph (the invoked function pointer is detected but not which actual functions may have been assigned to the function pointer), so no exit wrapper will be generated and `fromFunPtr` will return an invalid result. Moreover, if there are no other Cogent

functions of the same function type, the Cogent program will be translated to inconsistent C code: the invocation of the converted function pointer will be translated using a dispatcher function which does not exist.

Gencot supports forcing exit wrapper generation for functions not detected by the call graph, by specifying them explicitly in the file `<package>.gencot-externs` when generating the unit files by executing `gencot unit` (see Section 3.13.1). The following additional steps are required when forcing an exit wrapper for an external C function `f` which is not detected by the call graph:

- To generate the correct type for the exit wrapper a parameter modification description is required for `f`. Since the required descriptions are also determined using the call graph, `f` must additionally be specified explicitly when generating descriptions for external functions using `parmod externs`. The `parmod` script reads the same file `<package>.gencot-externs` (see Section 3.13.3), so this happens automatically. However, `parmod externs` does not generate the descriptions, it only selects them from a file given as input. Therefore the file containing the definition of `f` must be processed by `parmod init` in addition to the files processed as described in Section 4.3.2 and the resulting descriptions must be added to the file input to `parmod externs`.

- Even if explicitly specified, `gencot unit` generates exit wrappers only for functions which are declared in one of the source files belonging to the Cogent compilation unit (usually the declaration will be in a `.h` file included by a source file). If `f` is only invoked through a function pointer it may be the case that the declaration of `f` is not present in either Cogent compilation unit source file. In this case the "pseudo source" `additional_externs.c` must be added to the Cogent compilation unit which only includes the `.h` file where `f` is declared. More generally, it should include all `.h` files with declarations of functions for which an exit wrapper shall be forced and which are not included by regular sources in the Cogent compilation unit. As described in Section 3.12.8, the name `additional_externs.c` is treated in a special way by Gencot, it is ignored when the main Cogent and C source files are generated.

4.7.3 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 a new variable with the same name `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, (setFld_m, v)))
```

using a function `setFld_m` for modifying the structure.

4.7.4 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 if the variable is local, since it only supports the address operator when the result is a heap address or a global 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[EVT(CPtr U32)] heap
| Success (ptr,heap) ->
  fst INIT(Full,CPtr U32) (ptr,#{cont=5})
  | Success ptr ->
    let  ... use ptr ...
    in dispose (fst CLEAR(Simp,CPtr U32) (ptr,()),heap)
  | Error eptr -> dispose (eptr,heap)
| Error heap -> heap
```

where `INIT(Full,CPtr U32)` is used to initialize the referenced value. Here, the result of the expression is only the heap, in more realistic cases it would be a tuple with additional result values.

The drawback of this solution is that it is less efficient to allocate the variable on the heap, than to use a stack allocated variable. If it is only used for a short time, a better solution should be created manually.

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 allocated on the stack, 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 an abstract polymorphic function `getrefFld_i2` 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 an abstract polymorphic function `modrefFld_i2` 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 4.6.3). 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`, `getrefArrChk`, 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(FXU32X_U32) Cogent_f
in ...
```

In this case `ptr` has the nonlinear type `#CFunPtr_FXU32X_U32` in Cogent and may freely be copied and discarded in its scope.

4.8 Completing the Cogent Compilation Unit

Chapter 5

Verification

Additionally to running the translated Cogent program, the goal is to verify it by formally proving properties about it, such as correctness or safety properties.

The Cogent compiler generates a formal specification and it generates a refinement proof that the specification is formally valid for the C code generated by the Cogent compiler. The specification consists of definitions for all types and functions defined in the Cogent program. The definitions are formulated in the HOL logic language of the Isabelle proof assistant and reflect the definitions in the Cogent program very closely. In particular, they follow a pure functional style and are thus a good basis to prove properties about them.

Cogent does not provide any support for working with the formal specification. This chapter describes methods and support for this, assuming that the formal specification has been generated for a Cogent program which results from translating a C program with Gencot. The support consists of three main levels. The first level supports specifying alternative "semantic" definitions for the Cogent functions. The second level supports replacing the the data structures used in the Cogent program by more abstract data types and lifting the function semantics accordingly. The third level supports using these lifted semantics to prove application specific properties, such as functional or security properties.

5.1 Function Semantics

The formal specification generated by the Cogent compiler (the "shallow embedding") consists of three theory files. For each Cogent program a specific prefix `X` is used in the theory names. As described in Section 3.10.2 the file `X_Shallow_Desugar_Tuples.thy` contains the definitions for all non-abstract functions defined in the Cogent program. The function names are the same as those used in the Cogent program. Additionally it contains definitions for functions corresponding to all lambda terms used in the Cogent program, where the function names are automatically generated.

These function definitions strictly reflect the Cogent function definitions in the following ways

- Functions invoked by a function `f` in Cogent are also invoked by `f` in the shallow embedding. These may either be defined functions or abstract functions. For abstract functions the generated theory `X_ShallowShared_Tuples.thy`

contains declarations as Isabelle "constants" specifying the name and the type.

- Operations on Cogent record and tuple types are represented by equivalent operations on HOL record and tuple types.
- Cogent control structures like `let`, `if`, and `match` expressions are represented by corresponding HOL control functions (`let`, `if`, and `case`).

The main differences are that matching with complex patterns in Cogent is resolved to matching single variables in the shallow embedding and that the shallow embedding may use additional bindings to auxiliary variables.

The generated shallow embedding must be complemented by HOL definitions of all abstract functions used in the Cogent program. Since Cogent generates declarations for them, their definitions must either be provided as axioms or by overloading.

Note that invoked abstract functions also cover operations on data types other than primitive and struct types (see Section 2.7) and loop control structures (see Section **??**). Since we assume that the Cogent function definition has been generated by Gencot from the C source, Gencot also generates HOL definitions for these abstract functions. Together, the functions in the shallow embedding strongly reflect the steps used in C programs to implement tasks.

The goal of the function semantics is to provide an alternative definition (a "semantics") for all (abstract and non-abstract) functions and prove its equivalence with the original definition. The semantics is defined using HOL functions for working with the data structures resulting from translating a Gencot-generated Cogent program. Gencot provides a rich collection of such functions, so that the semantics definitions can achieve the following properties:

- Every semantics does not use any of the defined or abstract functions from the Cogent program and can be regarded as a "pure" standalone HOL description of the function.
- Sequential observations and modifications of data structure parts are aggregated to parallel observations and modifications of the whole data structure. In particular, loops for processing arrays are replaced by bulk operations on arrays.
- "Explicit" modifications of data structure parts, where the old value is accessed, modified, and then inserted at the same place, are replaced by "implicit" modifications, where the modification is specified by a modification function to be mapped to the part.

Gencot provides such semantics for all abstract functions implementing its basic operations, either by defining a semantics and proving equivalence to the provided function definition, or by directly defining the function in the form described above.

5.1.1 HOL Types for Semantics Specifications

The functions used to specify the semantics have to deal with values of the types defined in the Cogent program. The nonprimitive Cogent types are translated by the Cogent compiler to Isabelle as follows.

- Cogent record types are translated to HOL record types where field names get an index f appended, thus `name` becomes $name_f$.
- Cogent tuple types are translated to HOL tuple types.
- Gencot array types are translated to HOL record types which contain a single field for a HOL list. The record type is generic for the element type, but it is specific for the array size, which is achieved by encoding the array size in the field name, as described in Sections 2.6.5 and 2.12.1. Gencot provides support for these types as described in Section 2.12.4.
- Gencot pointer types are translated to HOL record types which contain a single field of the type referenced by the pointer. The field always has the name $cont_f$, as described in Section 2.6.7.
- Gencot array pointer types of the form `CArrPtr el` used for explicitly sized arrays (see Section 2.7.13) are directly translated to the HOL list type *'el list*, generic for the element type.
- Gencot types of the form `(MayNull a)` are translated to option types of the form *'a option*.

Therefore the functions for semantics specification mainly work with HOL records, tuples, and lists. Since in C arrays are naturally viewed as partial mappings from index values to element values, Gencot also provides functions for HOL map types of the form $A \rightharpoonup B$.

5.1.2 Functions for Semantics Specification

According to their type, we classify the functions for semantics specification into *construction functions*, *observation functions*, and *modification functions*.

Construction Functions

A function f is a **construction function for a type** T if it has the type $A \Rightarrow T$ for some type A or is a constant of type T.

Observation Functions

A function f is an **observation function for a type** T (also called a *projection*) if it has the type $T \Rightarrow A$ for some type A. An observation function need neither be surjective nor injective. If it is not injective it is an "abstracting" observation which discards some information about the observed values.

Two observation functions $f :: T \Rightarrow A$ and $g :: T \Rightarrow B$ can be combined to a new observation function by an injective combination function $c :: A \Rightarrow B \Rightarrow C$. The combined observation is $(obscmb\ c\ f\ g) :: T \Rightarrow C$ which is defined by $(obscmb\ c\ f\ g)\ x \equiv c\ (f\ x)\ (g\ x)$. Since c is injective the single observations can always be retrieved from the combined observation. Useful combination functions are the HOL pairing function *Pair* or every HOL record construction function *make* for a record with two components. In the former case the combined observation is a function $(f \times g) :: T \Rightarrow (A \times B)$ defined by $(f \times g)(x) \equiv (f\ x,\ g\ x)$, which returns the pair of observation values. Usually, a combined

observation is "less abstract" than the single observations, because it returns more information about the observed values.

The observations f and g are **independent** if *range* $(f \times g) =$ (*range* f) $\times$ (*range* g), i.e. if for every value of f function g can observe all its possible values and vice versa.

If the result type A of an observation function $f :: T \Rightarrow A$ is again a data structure with an observation function $g :: A \Rightarrow B$, the composition $g \circ f :: T \Rightarrow B$ is an observation function for type T. A typical case where this occurs are nested data structures, where a structure of type A is nested in a structure of type T.

Basic observation functions for HOL records are the field access functions which are automatically provided by HOL for each defined record type. Functions for different fields are independent observations. The basic observation function for HOL lists is function *nth* :: *'el list* ⇒ *nat* ⇒ *'el* with the abbreviation (*nth l i*) = (*l* ! *i*). Gencot provides the function *elm* :: *nat* ⇒ *'el list* ⇒ *'el* with swapped arguments, so that for every index *i* the function (*elm i*) is an observation function on lists. If $i \neq j$ the observations (*elm i*) and (*elm j*) are independent. Independent basic observation functions for HOL pairs are the HOL functions *fst* and *snd* which return the first and second component, respectively. For larger tuples Gencot provides observation functions *fstOf*<*n*>, *sndOf*<*n*>, *thdOf*<*n*>, *frtOf*<*n*>, *fifOf*<*n*>, and *sixOf*<*n*> for <*n*> ranging from *2* to *6*, where *fstOf2* = *fst*, *sndOf2* = *snd*, and so on.

Modification Functions

A function f is a **modification function for a type** T (also called an *update function*) if it has the type $T \Rightarrow T$. A modification function need neither be surjective nor injective.

For an observation function $o :: T \Rightarrow A$ the *part update function for* o is a function $m :: (A \Rightarrow A) \Rightarrow T \Rightarrow T$ which takes a modification function for type A as its argument and satisfies the following laws:

1. $o\ (m\ u\ x) = u\ (o\ x)$
2. $(m\ id) = id$
3. $(m\ u1) \circ (m\ u2) = m\ (u1 \circ u2)$
4. $m\ u\ x = m\ (\lambda\text{-}.\ u\ (o\ x))\ x$

According to (1) it modifies the observation by applying u to it. According to (2) it does not modify any other part. According to (3) it can be chained by chaining the modification functions for the observation. According to (4) it takes no other information into account than the modified observation. For every modification function u for type A the function $(m\ u)$ is a modification function for type T. The laws (2) and (3) imply that type T is a functor for type A and m is a "functorial action" or "mapping function" for it. A part update function need not exist for arbitrary observation functions.

Gencot provides support for part update functions in theory *Gencot-PartAccess*. It defines the type constructor *PartUpdate* with (*T*,*A*) *PartUpdate* = (*A*⇒*A*) ⇒ *T* ⇒ *T* and it defines the locale *PartUpdate* which introduces instances of

laws (2) and (3) for a function m and the locale *PartAccess* which additionally introduces instances of laws (1) and (4) for a pair of functions o and m.

If two observations $o1 :: T \Rightarrow A$ and $o2 :: A \Rightarrow B$ have part update functions $m1 :: (T,A)$ *PartUpdate* and $m2 :: (A,B)$ *PartUpdate* the composition $m1 \circ m2$ is a part update function for the composed observation $o2 \circ o1$ (note the reversed order of composition).

If two independent observations $o1 :: T \Rightarrow A$ and $o2 :: T \Rightarrow B$ have part update functions $m1 :: (T,A)$ *PartUpdate* and $m2 :: (A,B)$ *PartUpdate* the following additional laws are satisfied:

6. $o1 \circ (m2\ u) = o1$
7. $(m1\ u1) \circ (m2\ u2) = (m2\ u2) \circ (m1\ u1)$

According to (5) observation $o1$ absorbs modification $m2$ (by symmetry also $o2$ absorbs $m1$). According to (6) modifications $m1$ and $m2$ commute.

Gencot provides support for these laws by defining the locale *PartComm* which introduces an instance of law (6) for two part update functions $m1$ and $m2$, and the locale *PartAbsrb* which introduces an instance of law (5) for two pairs of observations and part update functions $o1$,$m1$ and $o2$,$m2$.

For every HOL record field access function *fld* the part update function is *fld-update* which is also automatically provided by HOL. For every list element access function (*elm* i) Gencot provides the part update function (*elm-update* i).

5.1.3 Arrays and Wellsizedness

In C most arrays have a known size, either specified as part of its type, or stored somewhere in the context where it can be retrieved when the array is processed. The array size is used for array processing, such as as a limit for loops or as a comparison value when testing whether an index is valid. In the shallow embedding arrays are represented as lists of arbitrary length, therefore the size information is not available.

Gencot supports C arrays generically in theory *Gencot-CArray-Lemmas*. A C array is an arbitrary type *carray* with two observations: a function *arr* :: *carray* $\Rightarrow$ *'el list* for the representing list and the function *siz* :: *carray* $\Rightarrow$ *nat* for the array size.

Array Wellsizedness

For all correct representations of C arrays the length of the list must equal the array size. Additionally, Gencot does not support empty arrays, so the array size must not be zero. Gencot defines the predicate *wlsd* :: *carray* $\Rightarrow$ *bool* ("wellsized") to express these properties. Wellsizedness is a special case of the general notion of wellformedness. This approach covers the C array types as described in Section 2.6.5 as well as the explicitly sized arrays (see Section 2.7.13 and self-descriptive and externally described arrays (see Section 4.6.3).

Gencot provides support for C array types and explicitly sized arrays. Other arrays are program specific, support for them must be provided manually. For C array types with a fixed size Gencot defines the locale *FxdArrSemantics* which

sets up the support for all arrays of a specific size, and interpretes it for every array size occurring in the translated program.

Generalized Wellsizedness

C arrays may be nested in other data structures, a data structure may contain several different arrays and arrays may be nested in elements of other arrays. For every data structure which contains one or more arrays, all of them must be wellsized. To express this property it is recommended to define an according generalized wellsizedness predicate for all data structures which contain atleast one array.

Since the data structures are program specific this must be done manually.

In a correct C program the generalized wellsizedness must hold for every data structure at every moment in time (with the exception of time spans where the array size is changed (e.g. by allocating more memory) and the stored size must be updated or, for self-described arrays, the content must be adapted to the new size). This property also holds for all values in a Cogent program which results from a translation by Gencot, and for every value in the shallow embedding. Thus, wellsizedness can be assumed for every otherwise unknown value in the shallow embedding, in particular, for all arguments of the Cogent functions.

This allows to always derive the length of all lists used for representing arrays. It is an important prerequisite when specifying the semantics for Cogent functions which process arrays, since it is required for proving, e.g., that a loop actually processes the whole array and is equivalent to a bulk operation for the representing list. Therefore, semantics specification theorems usually have wellsizedness premises for all arguments which contain arrays.

Wellsizedness Preservation

In a correct C program all primitive array modification operations preserve wellsizedness. For arrays specified by an array type with a fixed size they do not modify the array size. For other arrays, whenever the size changes the associated size specification is updated.

This implies that all other modification functions which are built from these primitive operations also preserve (generalized) wellsizedness for all their arguments. The same holds for the Cogent functions resulting from a translated C program and for their shallow embedding form and their semantics. If two such functions are composed, the semantics of the second function can only be substituted if the wellsizedness premise for its argument has been proved, which means to prove that the first function must preserve wellsizedness.

Moreover, the property that every function which modifies data structures with arrays preserves wellsizedness is an important correctness property for the translated C program, so its proof is a relevant part of the correctness prove for the program.

Therefore it is recommended to provide wellsizedness preservation proofs for all Cogent functions in the shallow embedding, which modify data structures with arrays. In the simplest case the corresponding proposition for a function f has the form $wlsd\ x \Longrightarrow wlsd\ (f\ x)$. If f takes an argument tuple and returns a result tuple where one component contains arrays the proposition has the form $wlsd\ x \Longrightarrow wlsd\ (sel\ (f\ (x1,...,x,...,xn)))$ where sel is one of the tuple selector

functions like *thdOf*<*n*>. Alternatively, generalized wellsizedness predicates can be defined for the argument and result tuples and the simpler form above can be used. This may be useful if several functions have argument and/or result tuples of the same type.

Wellsizedness Preservation for Part Update Funćtions

If m is a part update function for an observation o which contains arrays, (m u) only preserves wellsizedness if u does for the part. Since u is not applied to an argument, wellsizedness preservation for m cannot be specified as described above.

Therefore Gencot provides the predicate $prsvwlsd :: ('x \Rightarrow 'x) \Rightarrow bool$ for modification functions defined by $prsvwlsd\ f \equiv \forall x.\ wlsd\ x \longrightarrow wlsd\ (f\ x)$. Now the wellsizedness preservation proposition for a part update function can be specified as $prsvwlsd\ u \Longrightarrow prsvwlsd\ (m\ u)$. The predicate can also be used to specify wellsizedness preservation for arbitrary modification functions f as $prsvwlsd\ f$ or, with tuples as argument and result as $prsvwlsd\ (\lambda x.\ sel\ (f$ (*x1*,...,*x*,...,*xn*))).

Automatic Wellsizedness Deduction

Gencot provides support for defining a set of rules so that the simplifier can deduce all relevant wellsizedness propositions automatically.

There are two main kinds of such propositions: wellsizedness propositions for observations and wellsizedness propositions for modification results.

If for an observation function $o :: T \Rightarrow A$ the result type contains arrays it must be possible to deduce $wlsd\ x \Longrightarrow wlsd\ (o\ x)$, in particular, if the result type A is a C array type. The proof is usually done by unfolding the definition of o and the *wlsd* predicate for type T. For every C array type it must be possible to deduce $wlsd\ x \Longrightarrow length\ (arr\ x) = siz\ x$ which follows from the definition of *wlsd*. Together these rules allow to deduce the length of all array representation lists from wellsizedness premises for Cogent function arguments. The corresponding rules must be specified for every observation function o and for every C array type, they should be added directly to the simpset.

If for a modification function $m :: T \Rightarrow T$ the type T contains arrays it must be possible to deduce $wlsd\ x \Longrightarrow wlsd\ (m\ x)$, i.e., wellsizedness preservation. It is required for wellsizedness premises of semantics rules, if the argument is the result of applying a modification function. This deduction is mainly supported by rules of the form $[\![wlsd\ x; prsvwlsd\ m]\!] \Longrightarrow wlsd\ (m\ x)$ and rules for deducing *prsvwlsd m*.

Gencot provides a rule bucket *wlsd* for collecting such rules and it provides locales *Wlsd* and *Prsvwlsd* for generating the rules and inserting them into the bucket. The *Wlsd* locale must be interpreted for every wellsizedness predicate, the *Prsvwlsd* locale must be interpreted for every modification function m for a type with arrays. Additionally, for every modification function m a rule for deducing wellsizedness preservation must be specified and inserted in the bucket. For construction functions a rule must be specified, that the result is always wellsized. Gencot provides locales *PrsvwlsdAlways*, *PrsvwlsdIfpart* for the most common cases. The bucket must be added to the simplifier whenever wellsizedness deduction involves modification functions.

If arrays nested in other arrays are involved, additional rules are required which deduce properties for all elements of a list. Gencot provides the bucket *lstp* for these rules and it provides the locale *PrsvwlsdIfarr* which puts the rules into the bucket and must be interpreted for every C array part update function $m :: (carray,'el\ list)\ PartUpdate$ which modifies the array by modifying the representing list. The bucket must be added together with *wlsd* to the simplifier, whenever wellsizedness deduction involves modification functions and nested arrays.

5.1.4 Semantics Specification

Semantics Theorems

Wellformedness Preservation

Semantics for Abstract Functions

5.1.5 Proving Semantics Specification

Proving Record Operations

Proving Array Operations

Proving Loop Semantics

5.2 Datatype Abstraction

5.3 Proving Properties