Modular C

Jens Gustedt

Modular C

Jens Gustedt

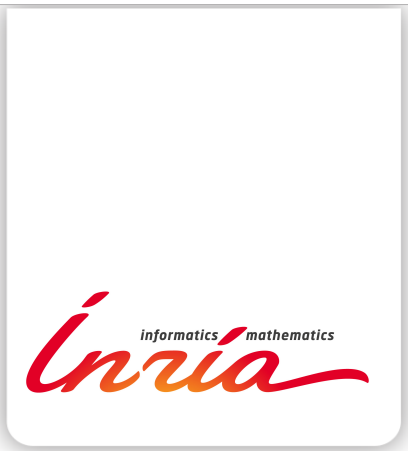
RESEARCH REPORT

N° 8751

June 2015

Project-Team Camus

ISSN 0249-6399 ISRN INRIA/RR--8751--FR+ENG



Modular C

Jens Gustedt

Project-Team Camus

Research Report n° 8751 — version 4 — initial version June 2015 —
revised version June 2018 — 48 pages

RESEARCH CENTRE
NANCY – GRAND EST

615 rue du Jardin Botanique

CS20101

54603 Villers-lès-Nancy Cedex

Abstract: We propose an extension to the C standard called *Modular C*. It consists in the addition of a handful of directives and a naming scheme transforming traditional translation units into *modules*. The change to the C language is minimal since we only add one feature, composed identifiers, to the core language.
Our modules can import other modules as long as the import graph remains acyclic and a module can refer to its own identifiers and those of the imported modules through freely chosen abbreviations. Other than traditional C's `#include`, our `import` directive ensures complete encapsulation between modules.
The abbreviation scheme allows to seamlessly replace an imported module by another one with equivalent interface. In addition to the export of symbols, we provide parameterized code injection through the import of "*snippets*". This implements a mechanism that allows for code reuse, similar to X macros or templates.
Additional features of our proposal are a simple dynamic module initialization scheme, a structured approach to the C library and a migration path for existing software projects.
The whole approach is validated by a formal description of a translation procedure from *Modular C* to common C and a proof of the correctness of that procedure. Thereby we are able to show that the class of *stable* programs can effectively be expressed in Modular C and that the gain of modularity is not thwarted by a loss of expressiveness. Here stable programs have a restriction on the C macro facilities that can be used. Most importantly, they may not use the **##** operator or similar tricks to create new identifiers, and they have to be careful when using the same identifier for a macro and a function.
Our approach is implemented and used successfully and efficiently in several projects. Interfaces can easily be provided both ways, to interface existing projects for Modular C or to interface Modular C libraries with other programming languages

Key-words: C, modularity, encapsulation

C modulaire

Résumé : Nous proposons une extension au langage de programmation C, nommé *C modulaire*. Elle consiste en ajoutant une poignée de directives et d'un schéma de nommage à transformer des unités de traduction traditionnelles en *module*. La modification au language-même est minimale, car nous y ajoutons une seul nouvelle caractéristique, les identifiants composés. Nos modules peuvent importer autre modules tant que la relation d'import reste acyclique et un module peut référer à ses propres identifiants et ceux des modules importés à l'aide d'abréviations librement choisis. Autre que l'`include` traditionel, notre directive d'`import` assure l'encapsulation complète entre modules. Le schema d'abréviation permet de facilement remplacer un module importé par un autre qui réalise le même interface. En plus à l'export de symboles nous fournissons l'injection de code paramétrer par l'importation de *snippets*. Ceci implante un mécanisme de réutilisation de code, similaire au *X-macro* ou `template`. Des outils supplémentaires que propose notre approche sont un schéma d'initialisation, un approche structuré à la bibliothèque standard de C et un chemin de migration pour des projets de logiciel existants.

Mots-clés : C, modularité, encapsulation

1 Introduction

Since decades, C is one of most widely used programming languages, see [2], and is used successfully for large software projects that are ubiquitous in modern computing devices of all scales. For many programmers, software projects and commercial enterprises C has advantages (relative simplicity, faithfulness to modern architectures, backward and forward compatibility) that largely outweigh its shortcomings.

Among these shortcomings, is a lack of some closely related features: encapsulation, reusability and composability. Most importantly, C misses to encapsulate different translation units (TU) properly: all symbols that are part of the interface of a software unit such as functions are shared between all TU that are linked together into an executable.

The common practice to cope with that difficulty is the introduction of naming conventions. Usually software units (referred as *modules* in the following) are attributed a name prefix that is used for all data types and functions that constitute the application programming interface (API). Often such naming conventions (and more generally coding styles) are perceived as a burden. They require a lot of self-discipline and experience, and C is missing features that would support or ease their application.

A lot of examples for "encapsulation by convention" are already present in the C standard itself. To apprehend the difficulty let us try to make a complete list of the conventions that the C standard enforces in view of the encapsulation of its own library.

It reserves the prefixes `str`, `mem` and **`wcs`** (for string and byte functions), **`mtx_`**, **`cnd_`**, **`tss_`** and **`thrd_`** (for the thread interfaces), **`is`** and **`to`** (for character classification and conversion), `E` (for error codes), **`FE_`** (for the floating point environment), **`PRI`** and **`SCN`** (for IO formats), **`LC_`** (for locales), **`atomic_`** and **`memory_`** (for atomic types and functions), **`TIME_`** (for time zones), **`SIG`** (for signals). In addition it also reserves prefix-suffix combinations `[u]`**`int`**`.*`**`_t`** (for integer types) and `[U]INT_.*{`**`MAX`**`|MIN|C}` (for integer macros), and the long list of names of C standard functions (for use as external symbols).

We propose a new tool, *Modular C*, that generalizes this approach by systematically adding name prefixes to the symbols that are exported by a translation unit. A number of simple rules and compiler *directives* are provided to facilitate the usage of this modular naming structure and helps to assemble software projects seamlessly and efficiently.

Overview

This paper is organized as follows. In the continuation of this introduction we will attempt to make our point about the shortcomings that we try to tackle, namely lack of encapsulation, reusability and composability. Then we will argue why C needs a specific approach to overcome these deficiencies and summarize our contributions. Section 2 will then introduce our main concept, a modestly improved naming scheme for identifiers that are exported by a module. Import of features from other modules, Section 3, is then organized simpler and stricter than in common C. As a result, the modular organization of projects is simplified and consistent module initialization can

be guaranteed. Reuse of small software components ("snippets") is then introduced in Section 4. All this leads to a formal description of a translation procedure from *Modular C* to common C, that can be proven to be correct, Section 5. The presentation of our new approach is completed by providing a structured view of the C library, Section 6, and by discussing the transition to *Modular C* in terms of a reference implementation and migration paths for existing projects, Section 7.

Note: An online reference manual can be found here: http://cmod.gforge.inria.fr/doxygen/index.html. The terminology used in this paper is based on the ISO C standard [8]. In particular, we refer to the concept of a *conforming program* as it is defined there.

1.1 Lack of encapsulation

The example of C's handling of its own library API already shows all weaknesses of the current approach:

It is intrusive: Other software projects that include C library headers just have to cope with the naming choices that have been made. If my module declares functions top or strip it might be in conflict with functions in ctype.h or string.h. Such problem might only become visible much later when the C standard evolves, or when my program is linked against a new implementation of the C library.

It is inconsistent: Reserved names may include underscores or not. They may only be reserved if their prefix is followed by uppercase, or if followed by a suffix such as **_t** ... Some of the reserved names are **struct**, **union** or **enum** tags, some are just plain identifiers.

It is illegible: The rules are difficult to apprehend and to memorize. They form an superfluous hurdle for beginners and occasional programmers with the C programming language.

It is ever growing: The C standard evolves from version to version (as most software interfaces) and adds constraints to the above list.

It is incomplete: The C standard headers define naming components, namely **struct** fields, that are not protected by the above rules. User provided macros can thereby interact badly with the internals of the **struct** of an included header. *E.g* the identifier **tv_sec** could be used as a macro, although the time.h header uses it as a **struct** field.

For the latter, no "official" rule forbids an application module to define a macro **tv_sec** which could badly interact with **struct timespec** from time.h. A conflict here may only appear when a third party project attempts to combine that module with some timing code. Other potential conflicts concern identifiers that are used as parameter names for function prototypes or for **inline** functions. A parameter named string could interact with another macro definition. Such a definition could come from another completely independent module or from a future version and implementation of the C standard library that introduces it in strings.h.

Other commonly used interfaces such as the POSIX operating system API add to that Babylonian disorder by reserving other prefixes (such as pthread_ for threads)

or suffixes (**_t** for **typedef**) or by using the same name as a **struct** tag and a function (stat). All of this makes the usage of the C library quite tedious, but occasionally is also at the origin of errors in implementations of the C library itself: avoiding the pollution of the application name space requires a continuous struggle with feature macros and include guards.

Besides these problems of name space pollution for shared identifiers, C offers some tools for encapsulation on which we will build for our approach. First, a whole set of features (types, macros, named constants) are per default not visible outside a given TU: they have what the C standard coins no "*linkage*", see below. A second mechanism to hide global data and function objects from other TU is to declare them with "*internal linkage*" by using the **static** storage class.

1.2 Lack of reusability

In addition to these issues about mutual identifier space pollution, C fails to provide the necessary infrastructure for painless code reuse. Common strategies to ensure reusability are:

Type-agnostic functions: The C standard itself defines generic interfaces for searching and sorting (**bsearch** and **qsort**) that take data as pointers to **void** and comparison functions as **int** (*)(**void const***, **void const***). This does not lead to real source code sharing but basically only switches off the type system to apply the same type agnostic binary to different application types.

Macros: Complicated software systems such as the Linux kernel[1] or the Boost preprocessor library[2] define parameterized data structures such as lists through a set of macros. The major drawback of this approach is that it quickly leads to code that is difficult to read, to apprehend, to maintain and to interface, and that may lead to replication of side effects.

X macros: This consists of writing small parameterized code snippets in .c files. These provide *e.g.* function definitions that can be repetitively included into other source. Parameterization of such code is effected by defining some specific macros before the include and undefining them right after, see [12]. Boost[3] partly exploits that technique. This approach is probably the most readable of the three (the file contains normal C code) and also is the safest of the three techniques listed here. Nevertheless it has not found much use in the field. It is syntactical odd and disruptive on the user side. Preprocessor **#define**, **#include** and **#undef** are leaked in the implementation part of the user code. The need to provide separate declarations and definitions doubles the maintenance overhead.

[1] http://kernelnewbies.org/FAQ/LinkedLists
[2] http://www.boost.org/doc/libs/1_57_0/libs/preprocessor/doc/index.html
[3] http://www.boost.org/doc/ ... /topics/file_iteration.html

1.3 Limited composability

C's strategy to compose different TU into functional programs is twofold. First, *linkage* ensures that several compiled object files can be stitched together along commonly known features, so-called *external symbols*. These are data or function objects that are provided by exactly one TU and that can be used by all others. They are only identified by their *name*, no other property such as type or size is in general enforced. A second level then consists in the use of **#include** directives that ensure that necessary definitions of types, macros and named constants are shared and that all the commonly used symbols have the same interpretation throughout the different TU.

This mechanism has several weaknesses, in particular almost all consistency checks are left to the user or to external tools. **struct** and **union** types and members, macros and named constants have "no linkage". The equivalence between a description of a feature in a header file and its instantiation is not guaranteed and is difficult to enforce from within the language. Linking TU with contradicting specifications for the same symbol result in "undefined behavior", that is, there is no guarantee to obtain a diagnostic or that the link fails, and bugged executable can be generated.

1.4 C needs a specific approach

In contrast to most other programming languages that are commonly in use, modern C has the advantage of a rigid, static type system and an unambiguous declaration syntax that, in principle, can still be parsed linearly in one pass. To our opinion, this relative simplicity of the language should not be wasted by approaches that address complex features of other languages, such as dynamic types, inheritance or implicit typing. Therefore we try to stay as closely as possible to the existing lexical and semantic properties of C.

Many approaches have been proposed over the years to improve the composability of C projects, but seemingly few of them have resulted in tangible improvements for programming in C on the language level. In particular, none of them seem to improve modularity, that is to provide better encapsulation and reusability in addition to composability. Starting with Unix' `make` [5], many build systems, tools for component composition (*e.g.* `Knit` [11]) or regrouping [1], header file generation [7], and package management (*e.g.* `dpgk` [4]) have been proposed. Integrated development environments (*e.g.* eclipse [9]) and sophisticated editors (*e.g.* `emacs` [14]) help programmers to reference declarations across several TU. Most of these tools need some additional specification (makefiles, project specification, dependency or group description, linker files, ...) to declare the connection between different TU. These usually have to be provided in addition to the source code itself. Complementary to that, in the present paper we try to develop a *language* feature that helps for the modularization of C code. In fact many of the tools from the above categories can be used to implement the system that we present, and not surprisingly our reference implementation uses some of them.

Existing approaches that address modularity on the language level can be split into two main categories. The first are *rule sets* that are provided by coding style and

enforcement strategies. They are used to restrain possible use of language constructs for *information hiding*, and thus to control the mutual impact that different modules may have. Most prominent among the coding styles is probably the Linux coding style [15]. Such coding rules then can be made sound and effectively enforceable as has been shown by [13]: they provide a set of four rules and operational semantics for a sufficiently large subset of C that allows them to check consistency and to identify bugs in large C projects.

To our opinion these rule based approaches are only going half the way. They don't give the necessary tools for real encapsulation or code reuse and they leave the burden of naming conventions to the programmer. Our own approach here extends and improves such *rule sets* and should be compatible with most common practice. It is in particular compatible with the rules of [13], so all proven properties of that approach also apply to ours.

As another approach many new "C-like" compiled programming languages have been defined over the years. Most prominent among these are certainly C++, Objective-C and Java that in the advent of the concept of object oriented programming provided better encapsulation and code reuse. Whereas these languages have found a large distribution, they were only able to take a partial share in the software market. All three have the disadvantage that they are much more complex than C. They add multiple language constructs to a core that is similar to C, but provide no (or tedious) migration paths for existing software, infrastructure or developers.

Recently there have been efforts to provide a common modular framework for all three language siblings C, C++ and Objective-C, see [6]. Whereas that proposal tackles and solves problems with the include hierarchy it seems to add an extra layer of semantic complexity. Most importantly it does not address nor solve one of the major problems that we have identified above: the lack of mutual encapsulation of TU of any of the target languages.

Google's *Go* (also sometimes referred as *Golang*) has gone a different path by introducing a new programming language, see [10], that addresses most if not all of the issues that we mention above. Here, we are particularly interested in its `import` feature that has been one of the seeds of this work.

But unfortunately also, by design *Go* is a rupture. C on the other hand owes much of its success to the fact that it cautiously watches to be backward and forward compatible: standard conforming code that worked 10 or 20 years ago still works today and will most likely still work as many years from now. For many, switching to a new programming language is not an option and this proposal here is intended to help to improve the C programming language with respect to the issues that we raised. The strong emphasis on compatibility still has allowed C to evolve constantly. As major new features C11 [8] integrated threads to the C library and atomics and type generic functions to the core language. Our proposal inscribes itself in that steady and constructive line of improvements that have been added to C.

The presented proposal for "Modular C" shares a lot of ideas with the ones mentioned above, but is meant to be much simpler. It is designed to be compatible with C's core language and with the common coding practice in the C community. It only targets that one language, C, and its support library, alone. The other two languages targeted

by [6] have their own proper features and problems concerning modularization and a mixed discussion easily misses language specific features and opportunities. On the other hand, compared to the other two, C has a specific default, its lack of language infrastructure for consistent code reuse that must be improved.

1.5 Our contributions

Modular C adds one main language feature (*composed identifiers*) and a handful of directives that provide an extension of the C language with the following properties:

Encapsulation: We ensure encapsulation by introducing a unique composed module name for each TU that is used to prefix identifiers for export. This creates unique global identifiers that will never enter in conflict with any identifier of another module, or with local identifiers of its own, such as function or macro parameters or block scope variables. Thereby by default the import of a module will not pollute the name space of the importer, nor interfere with parameter lists.

Declaration by definition: Generally, any identifier in a module will be defined (and thereby declared) exactly once. No additional declarations in header files or forward declarations for **struct** types are necessary.

Brevity: An abbreviation feature for module names systematically avoids the use of long prefixed identifiers that are necessary for naming conventions.

Completeness: The naming scheme using composed identifiers applies to *all* file scope identifiers, that are objects, functions, enumeration constants, types and macros.

Separation: Implementation and interface of a module are mostly specified through standard language features. The separation between the two is oriented along the C notion of external versus internal linkage. For an **inline** function, the module that defines it also provides its "instantiation". Type declarations and macro definitions that are to be exported have to be placed in code sections that are identified with a **declaration** directive.

Code Reuse: We export functionality to other modules in two different ways:

- by interface definition and declaration as described above, similar to what is usually provided through a C header file,
- by sharing of code snippets, similar to X macros or C++'s templates.

The latter allows to create parameterized data structures or functions, easily.

Code repetition: The **foreach** and **do** directives allow simple code repetition at compile time. This is *e.g.* useful to instantiate a number of functions for a list of values.

Acyclic dependency: Import of modules is is not from source but uses a compiled object file. This enforces that the import relation between modules defines a directed acyclic graph. It can be updated automatically by tools such as POSIX' `make` and we are able to provide infrastructure for orderly startup and shutdown of modules according to the import dependency.

Interchangeability: The abbreviation feature allows easy interchange of software components that fulfill the same interface contract.

Optimization: Our approach allows for the full set of optimizations that the C compiler provides. It eases the use of **inline** functions and can be made compatible with link time optimization.

C library structure: We provide a more comprehensive and structured approach to the C library.

Extendability: Our approach does not interfere with other extensions of C, such as OpenMP or OpenCL.

Migration path: We propose a migration path for existing software to the module model. Legacy interfaces through traditional header files can still be used for external users of modules.

2 All is about naming

In its simplest form, our proposal just formalizes a common approach that is used for C projects. It uses a *name prefix* (the **module** name) to specify the membership to a module and to access other software components, see Listing 1 for an example. A C module chooses itself a composed name, and may then be imported by other modules through that name. This new import feature combines linking and **#include** and therefore guarantees consistent use of all features according to their definition throughout a whole software project.

The module name may be deduced implicitly from the source file name or, alternatively, it can also be specified explicitly in a **module** directive. Names of imported modules may be deduced implicitly just by using their features, or, alternatively, they can also be specified explicitly in an **import** directive. *E.g* we could add the following to Listing 1 to make these names explicit:

```
#pragma CMOD separator ::
#pragma CMOD module proj::structs::list
#pragma CMOD import ..::element
#pragma CMOD import C::io
```

This sets the module prefix to be composed of three identifier elements, namely proj, structs and list. Syntactically, the components of identifiers are separated by a token, the *separator*, that may (with some restrictions) be chosen for each module. Here the **separator** directive uses the Unicode character ∶∶, but this could *e.g.* also be C++'s traditional namespace separator :: or some other locale imposed character. The significance of the composition of such module names will be explained later in Section 3. For brevity we will omit all **separator** directives from other code samples.

The only additions of *Modular C* to the code are *directives* and composed identifiers, no keywords or constructions are added to the core language itself. We have chosen to use the prefix **#pragma CMOD** for lines that contain directives, but this choice by itself is not essential for the feasibility of our approach.

In the above, all file scope identifiers of the module are visible to the outside with the prefix. *E.g.*, our module implements a function init that is externally visible as proj∶∶structs∶∶list∶∶init. If the imported module proj∶∶structs∶∶element in turn

Listing 1 A module that exports two symbols and a type. The module name, proj::structs::list say, is supposed to be deduced from the file name. A type proj::structs::element is implicitly imported and the usage of it supposes that it has a member var.

```
/* The following declarations are exported */
#pragma CMOD declaration

/* Exports proj::structs::list::head */
struct head {
  /* Implicitly imports proj::structs::element. */
  proj::structs::element* first;
  proj::structs::element* last;
};

/* From here only exported if external linkage. */
#pragma CMOD definition

void say_hello(void){
  C::io::puts("Hello world");
}
static unsigned count;
static void say_goodbye(void){
  C::io::printf("on exit we see %u", count);
}
head* init(head* h) {
  if (h) *h = (head){ 0 };
  return h;
}
/* Exports proj::structs::list::top, no conflict with ctype.h */
double top(head* h) {
  /* Expects proj::structs::element to have a member ``val''. */
  return h->first.val;
}
```

provides a symbol init, our module can access that one through the universal name proj::structs::element::init, and no naming conflict occurs between the two modules.

Similar to C++'s strategy, all composed identifiers are *mangled* during compilation such that no naming conflict with other modules or standard headers can occur. Such mangling needs the notion of *reserved identifiers*. These are identifiers that no application code is allowed to use. For our purpose it is sufficient to target a particular subset of C's reserved identifiers:

Definition 2.1 *An identifier that starts with an underscore and is followed either by a second underscore or by a capital letter is* strictly reserved.

C reserves these for internal use of the compiler implementation and for future language extensions, such as had been done by introducing **_Bool** in C99 and **_Static_assert** in C11. Identifier starting with an underscore, followed either by a second underscore or by a capital letter are *strictly reserved*.

Rule 2.2 *A module may not define any strictly reserved identifier or use it as a component of a composed name.*

As stated above, the C library then reserves a lot more identifiers for features that are interfaced via the different standard header files.

Definition 2.3 *An identifier is* reserved *if it is strictly reserved, a keyword, or used to name a feature of the C library.*

Observe that we allow the use of "normal" identifiers that might also be used by the C library. In fact, there are two reasons for that. First, we want to ensure that *Modular C* modules may seamlessly compile with any future version of the C library. Then, we want to be able to propose different versions of C library functions in modules with equivalent interfaces, as e.g. a strict C library version of **printf**, C::io::**printf**, and an augmented version POSIX::io::**printf**.

Rule 2.4 *A module may freely use any identifier that is not strictly reserved as a component of a composed name.*

This rule also applies to keywords (with some caution) and library function names. Also, the identifier **main** loses its special meaning when used in a module. Instead, optionally one or several entry points are declared via directives:

```
#pragma CMOD entry unit_test
```

This declares a semantic similar to traditional **main**. In "backwards compatibility mode" the identifier **main** is used as if such an **entry** directive for **main** would be present. The function unit_test must have a prototype that is equivalent to one of the following:

```
RT unit_test(void);
RT unit_test(int argc, char*argv[argc+1]);
RT unit_test(int argc, char const*argv[argc+1]);
RT unit_test(int (*argc), char* (*argv)[argc+1]);
```

and where RT is either **int** or **void**. Any valid identifier with a suitable prototype can be chosen as entry point. This also includes the module-local identifier **main**, or function names from other modules.

2.1 Exported features

A traditional C header file can provide access to a number of different *features*: constants, types, macros, objects and functions. To simplify the life of the user of C modules and to ease encapsulation we have the simple rule:

Property 2.5 *All the exported features of a module* M *correspond to a composed identifier* M :: L *where* L *is not strictly reserved.*

In particular this means that all types are named with composed identifiers and that no exported feature uses a name that does not start with the module's name.

In the following paragraph we go a bit into detail how Property 2.5 is assured. C already has a convention that regulates the export of symbols, namely the *linkage* of identifiers: only identifiers of functions and global variables have linkage. Other identifiers, *e.g.* tags,[4] **typedef** identifiers or macro names don't interact with those from other TU. Further, C distinguishes *internal* (declared with the keyword **static**) and *external* linkage. If not specified otherwise, *Modular C* just follows and enforces this concept.

Property 2.6 *A module exports all its identifiers that have external linkage.*

E.g., the module of Listing 1 exports proj :: structs :: list :: say_hello, but the identifiers proj :: structs :: list :: say_goodbye and proj :: structs :: list :: count are not exported since they are declared **static**. In addition it uses the imported identifiers C :: io :: **puts** and C :: io :: **printf** from the module C :: io.

On the proper language level (after preprocessign) C has a rigid static type system: identifiers can only be used after they have been declared, that is if sufficient type information has been associated to an identifier to formalize if it is itself a **typedef**, a tag, a named constant, a variable or a function. Within the same scope, this type association may not be changed. Using an identifier not according to its definition is a "*constraint violation*" and usually leads to an abortion of the compilation process.

For data objects, functions and tag types C also distinguishes *declarations* and *definitions*. A declaration only provides information about an identifier such as type and size, expected function arguments or the fact that a type with that tag exists. A definition additionally "instantiates" a feature. For data it reserves storage of the appropriate size, initializes it and identifies the object with its name. For a function it implements the function and provides a symbol in the symbol table of the binary file. For a tag type it fixes the kind (**struct**, **union** or **enum**) and the internal structure of the type.

In traditional C, header files (.h) usually contain declarations, whereas TU (.c) usually contain definitions.

C has a "one definition rule", that stipulates that any data or function object and any tag type must have exactly one definition. For *Modular C* we have an extension of this rule:

Rule 2.7 *The definition of an identifier also serves as its unique declaration.*

The first advantage that we obtain from this is that we don't have to separate a header (".h" file) from the code of the translation unit.

[4] For C, tagnames of **struct**, **union** and **enum** form their own "name space". These constructs are the only ones that result in a new type that is created. Other than the name suggest, a **typedef** does *not* define a new type but only introduces an alias for an existing type.

Property 2.8 *Identifiers are imported with the type of their definition.*

So far we can easily decide for global variables and functions if they are visible by exporters or not. If they are declared **static** they are local to the module, otherwise they are exported. Other identifiers (types, macros and named constants) are not exported unless we say so, explicitly. By that we maintain the encapsulation that C provides for these features. Nevertheless, we have to add a mechanism for those features that are part of the module's interface:

Rule 2.9 *Identifiers without external linkage are exported iff they are in a declaration section.*

This can be done by placing a **declaration** directive in the source as shown. All code after a such a directive, before the next **definition** directive, should only be declarations or definitions similar to the ones found in a traditional header file. So typically, such declaration sections only consist of type declarations (**typedef**, **struct**, **union** or **enum**) and definitions of macros. Similar to C++, Modular C overcomes the distinction between tag name space from an identifier name space.

Property 2.10 *All **struct**, **union** or **enum** declarations give rise to **typedef** that are implicitly inserted.*

E.g., for Listing 1 we can assume that the **struct** declaration itself is preceded by

```
typedef struct head head;
```

to declare the tag name and identifier at the same time. This means in particular, that the identifier head cannot be reused in file-scope to refer to a different entity than the **struct** head type.[5]

Definition 2.11 *A **typedef** of an identifier ID is* canonical *if is of the form*

```
typedef {struct|union|enum} ID ID;
```

Property 2.12 *All tagged **struct**, **union** or **enum** declarations are accompanied by an appropriate canonical **typedef**.*

2.2 Avoiding identifier clashes

The function top that is defined at the end of Listing 1 shows another feature. If is only visible as proj :: structs :: list :: top to the outside. So no naming conflict can occur with C library header ctype.h which reserves the prefix **to**. Generally, this feature allows us to reuse all reserved names of the C library, locally, and in general to decide freely on names inside a project. They will never conflict with names of other projects, as long as the chosen module names differ.

[5] Duplicating a **typedef** is allowed in C since C11.

This makes it easier to provide extensions to the C library, *e.g.*, the POSIX standard extends C by adding features to C library functions such as **printf**, much as it reads in traditional C. Within *Modular C* any extension can do that freely, if it uses its own module name space. Importers may use the two different versions of such functions simultaneously as long as they use prefixed names, such as C::io::**printf** and POSIX::io::**printf**.

For the C library itself, this approach is more flexible, too. Currently, the C library already has several groups of functions that provide the same functionality for different base types, such as the **fabs**/**cabs** or **strlen**/**wcslen** functions. In the case of **fabs**, there are even two "functions" with the same name, one that is provided by math.h and the other that is included by tgmath.h

With our approach, the C library can provide different modules to implement interfaces to real and complex floating point functions. We introduce six modules C::real::**float**, C::real::**double**, ..., C::**complex**::ldouble for the type specific functions from math.h. Any of these modules has a function that locally is named **abs** and that replaces **fabsf**, **fabs**, ... **cabsl**. The type generic function **fabs** from tgmath.h can be superseded by an identifier **abs** in C::math that interfaces all the above and analogous functions for integer types with a type generic macro.

```
1  #pragma CMOD module          C::math
2
3  #pragma CMOD declaration
4  /* Exports a type generic abs for
5     arithmetic base types. */
6  #define abs(X) _Generic((X)+0,  \
7     float:  C::real::float::abs,    \
8     double: C::real::double::abs,   \
9     /* more cases */            \
10    )(X)
```

To finish this part we summarize what it means for a type or an identifier to be exported by a module.

Definition 2.13 *A type definition or forward declaration is* exported *by a module M iff it is found in a* ***declaration*** *section. If the type is a* ***struct****,* ***enum*** *or* ***union*** *type with tag name X, it is exported with tag name M::X.*

Definition 2.14 *A file scope identifier X is* exported *by a module M as composed identifier M::X iff one of the following conditions holds:*

- *X is an object or function that is defined with external linkage. M::X then refers to the same object and has the same type as X.*
- *X is declared or defined in a* ***declaration*** *section.*
  - *X is an object or function that is defined with internal linkage. For each importing module N, M::X then refers to a specific object or function in N with internal linkage and has the same type, storage class, definition and initialization (if any) as X.*
  - *X is a* ***typedef*** *name. Then, M::X is an alias to same type as X.*

- *It is an enumeration constant. Then,* M::X *is an enumeration constant of the same value as* X*.*
- X *is a macro. Then,* M::X *is a macro with an expansion identical to the one of* X*.*
- X *is the tag name of a* ***struct****,* ***union*** *or* ***enum*** *type.* M::X *then is an alias to the corresponding type with tag name* M::X*.*

Property 2.5 also ensures that imported macros, as they have composed names, do not conflict with local identifiers:

Property 2.15 *No imported macro will replace internal identifiers such as variable or function names, tags,* ***struct*** *or* ***union*** *members, named constants or function parameters.*

2.3 Abbreviations

So far the gain we have achieved with our approach is that we avoid code duplication for identifiers with external linkage, reflected in Rule 2.7, and that we are better able to encapsulate functions that represent similar features. Otherwise the grammatical complexity of the code that this allows to write is similar to the traditional prefix convention for C identifiers. Listing 2 introduces another functionality, *abbreviations*. These identifiers (on the left side of the =, above) are *local* names that are only valid inside the particular module. Here, they may replace the composed name on the right side. By themselves, these abbreviations are not exported to other modules.

The name of every imported module and of the module itself can be abbreviated. This feature is inspired by a similar feature of the *Go* programming language. With

```
#pragma CMOD import elem=..::element
```

we introduce a shortcut for refering to the identifiers of the imported module. The symbol proj::structs::element::init can now be referred as elem::init.

Property 2.16 *Features can be accessed with two component names as* abbrev::name *where* abbrev *is a short prefix that is chosen in a* ***module*** *or* ***import*** *directive and* name *is the local name given by the module that defines the feature.*

In the example above, the **import** directive

```
#pragma CMOD import printf = io::printf
```

allows to refer to the C library IO function as **printf**, without prefix. This gains modularity. To partially or fully replace C's IO module by POSIX', we just have to replace the corresponding import directives. For a partial replacement of **printf** we could use:

```
#pragma CMOD import printf = POSIX::io::printf
```

For a full replacement of the whole io module we could use:

```
#pragma CMOD import io = POSIX::io
```

Provided that the function prototypes are compatible, no other code change is necessary.

Listing 2 The module of Listing 1 with abbreviations.

```
#pragma CMOD module head   = proj::structs::list
#pragma CMOD import elem   = ..::element
#pragma CMOD import io     = C::io
#pragma CMOD import printf = io::printf
#pragma CMOD import puts   = io::puts

/* The following declarations are exported */
#pragma CMOD declaration

/* Exports proj::structs::list */
struct head {
  elem* first;
  elem* last;
};

/* From here on, only exported if
   external linkage. */
#pragma CMOD definition

void say_hello(void){
  puts("Hello world");
}
static unsigned count;
static void say_goodbye(void){
  printf("on exit we see %u", count);
}
head* init(head* h) {
  if (h) *h = (head){ 0 };
  return h;
}
/* Exports proj::structs::list::top,
   no conflict with ctype.h */
double top(head* h) {
  return h->first.val;
}
```

Rule 2.17 *An abbreviation must not conflict with other local identifiers.*

In particular, no two **import** directives may use the same abbreviation.

2.4 The module name

Inside the module, all names exported by it are usually referenced through their short name, *e.g.*, init as we introduced it above. But the name proj::structs::list::init

could equally be used. There is one particular composed identifier of special interest, the module name itself. This identifier, can freely be used to put emphasis on one particular feature that is central for the module. For a first example, let us suppose that our "element" module exports a **struct** type:

```
#pragma CMOD module proj::structs::element
#pragma CMOD declaration
/* Exports proj::structs::element */
struct proj::structs::element {
  double data;
  proj::structs::element* next;
};
```

With an abbreviation elem the code simplifies:

```
#pragma CMOD module elem=proj::structs::element
#pragma CMOD declaration
/* Exports proj::structs::element */
struct elem {
  double data;
  elem* next;
};
```

As Listing 2 has shown, using the module name for the **struct** declaration then eases the use of that **struct** by the importer, here through the same abbreviation elem.

Property 2.18 *The composite name of a module M can be referred to by the abbreviation given by an **import** or **module** directive.*

Similar holds for the name of the module in Listing 2 itself. The structure that had been exported as proj :: structs :: list :: head in Listing 1 is now accessible as proj :: structs :: list, thereby reflecting the idea that this is the main data type this module is about. All users of the list module can easily use an "object centered" syntax to access the data type and its functions:

```
#pragma CMOD module another_project::doit
#pragma CMOD import list = proj::structs::list

list* pop_or_push(list* l, double a) {
  if (l && l->data == a) {
    l = list::pop(l);
  } else {
    l = list::insert(l, a);
  }
  return l;
}
```

In the example above the module name proj :: structs :: list is now abbreviated as head. Thus the definition that is internally visible as **struct** head is exported as

struct proj :: structs :: list. Our module can export identifiers in an "object oriented" scheme: it exports proj :: structs :: list for the type name and proj :: structs :: list :: init and proj :: structs :: list :: top as functions that act on this type.

This use of the module name is not limited to types. *E.g.*, a function can be in the center of interest of a module: a module C :: lib :: **rand** that interfaces the C library function **rand** as C :: lib :: **rand** may export the function **srand** as C :: lib :: **rand** :: set and the macro **RAND_MAX** as C :: lib :: **rand** :: **MAX**.

```
#pragma CMOD module    rand = C::lib::rand
#pragma CMOD declaration
int rand(void);
int set(unsigned int seed);
#define MAX C::lib::RAND_MAX
```

Another example could be to add integral and derivatives to a numerical function:

```
#pragma CMOD module    func = proj::func
double func(double x){ ... }
double func::deriv(double x){ ... }
double func::integ(double low, double high){ }
```

2.5 Composing names on a finer granularity

Modular C uses a second character that separates identifier components.

```
#pragma CMOD composer □
#define macro□0()  macro□1("empty")
#define macro□1(S) macro□2(5, S)
#define macro□2(N, S) /* do something with N and S */
```

This provides a simple tool to parameterize local names that share some properties. This avoids to force a particular choice of naming convention to the importer. If an importer chooses *e.g.* a "long dash" – as composer, the above can be accessed as macro–0, macro–1 and macro–2.

More importantly, this scheme will also be used to compose certain identifiers when they are imported from so-called "snippets", see below.

2.6 Mangling

To be able to set up some formal proofs for the validity of our approach, see Section 5, we have to specify how we will ensure that all file scope identifiers are encapsulated.

Definition 2.19 *Let S the source character set of a C platform, $::\notin S$, and $\widehat{S} = S \cup \{::\}$.*

1. *An identifier $N \in S^*$ is* valid *for a platform if it complies to a length restriction that the platform may impose.*

2. *A string* $\widehat{N} = N_0 :: N_2 \cdots :: N_{n-1} \in \widehat{S}^*$ *is a* valid composed identifier *if* $N_0, \ldots, N_{n-1} \in S^*$ *are valid identifiers. Such a valid composed identifier is* reserved *if either* $n = 1$ *and* N_0 *is reserved or if* $n > 1$ *and any of the* $N_0, \ldots, N_{n-1}$ *is strictly reserved.*
3. *A function* $f : (S \cup \{::\})^* \mapsto S^*$ *is a valid* mangling *if it is injective and if the image under* f *of any non-reserved valid composed identifier is not a reserved identifier.*

Observe that this definition allows that composed names with two or more parts may contain identifiers such as **int**, **printf** etc. that would otherwise be reserved by the core language or the library. It also disallows that a mangling accidentally maps a composed identifier to one that would be used by the C library.

For languages such as C++, many different functions have traditionally been used for mangling. In contrast to the requirement in 3 they usually map to reserved identifiers. This is because they have to ensure that non-reserved names that are defined elsewhere with **extern** `"C"` binding cannot clash with mangled names. *Modular C* implements complete encapsulation and has no mechanism to introduce "unmangled" names, so we don't need such a property, here.

On the other hand, we only require a mangling to map to non-reserved names because it makes our proofs simpler. Even in our reference implementation we use a derivation of such a function that is often used for C++ name mangling. It maps the identifiers N_i as of the definition to a concatenation of strings $M_i = \ell_i N_i$ where ℓ_i is the length of N_i, prefixed and postfixed by `_ZN2_C` and `E`, respectively. E.g. `C::real::double::minv` is mapped to `_ZN2_C1C4real6double4minvE`. By construction this function is injective. The prefix `_ZN` ensures it can effectively identified by the tools of our platform as a composed identifier. Identifiers with such a prefix are in fact strictly reserved and could theoretically clash with other internal identifiers of the platform. Any implementation of *Modular C* that uses reserved identifiers for mangling should ensure that this is not the case, and in particular that the mangling does not produce identifiers that are used by its C library. Equally, any mangling should ensure that none of the identifier patters that are listed in Section 7.31 of the C standard, "*Future library directions*", are in the image of f.

Definition 2.20 *For a conforming C source* P *and a mangling* f*, the* mangled source $f(P)$ *is the source file that results from a replacement of all non-reserved identifiers* N *in file scope that are not keywords by* $f(N)$*.*

Definition 2.21 *For a conforming module* M *and a mangling* f*, the* mangled module $f(M)$ *is the C source file that results form the application of the following:*

1. *remove the snippet, if any,*
2. *replace all abbreviations by the corresponding composed identifiers,*
3. *prefix all non-composed file scope identifiers by the module name,*
4. *replace all non-reserved composed identifiers* N *that are not keywords by* $f(N)$*.*

3 The import graph

The composed name of a module also determines some of the other modules that it imports, its parent modules.

Property 3.1 *A module implicitly imports all modules that have a name that is a prefix of its name.*

E.g., implicitly by its naming, module proj::structs::list imports proj and proj::structs. Then, with the **import** directive it explicitly imports proj::structs::element.

Consider another_project::doit from above as an example. It gives rise to the directed import graph that is depicted in Figure 1. Explicit or implicit imports define a

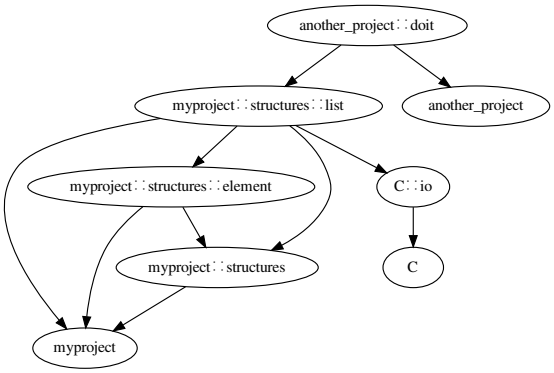


Figure 1 The import dependency graph of the example module

directed import relationship between the modules of a given project, its *import graph*. The import graph of a specific module A is the subgraph of the import graph of the project that contains all modules that A imports.

Property 3.2 *The import graph is transitive, that is if module A imports B it imports also all modules that B imports.*

Claiming full transitivity of imports differs from *Go*'s strategy. There, an import of a module B only provides the necessary information to use all interfaces of B and not more. This would be a possibility for C, too, that information could be directly stored in the compiled object of B, much similar as *Go* does this.

We have not adopted such a restricted approach for several reasons. First, it is difficult to implement. It would necessitate to write or integrate a full parser for the language into our front end, whereas the approach presented here is mostly text replacement of composed identifiers. Modular C can mostly be implemented with scripting as provided by the POSIX utility sed, see Section 7.1, below. Keeping track of the contents of the import graph would add a lot of complexity to the implementation of Modular C and probably worsen compile times substantially.

Second, forcing transitivity ensures that our approach is consistent with the rules of [13] for modularity and information hiding: their "Rule 3" that forbids "*vertical dependency*" for imported features is equivalent to claim that no module can be imported before any of its dependencies.

Then, *Go*'s main motivation for its restrictive strategy comes from observing the large compile time overhead that C++ include hierarchies impose on software projects: there, include files tend to be monolithic, they include a large number of other files themselves, and the language is difficult to parse. In particular, declarations of recursive templates can incur an exponential computational overhead at compile time, and this overhead occurs for *every* TU in a project uses such a type.

This situation is not at all comparable for C because C has no syntax constructs for recursive declarations. C compilers nowadays are able to parse C headers extremely fast. Their bottleneck usually is code generation and optimization, and so the problem that has been observed for C++ does not apply to C.

We will see below, how our **snippet** feature that provides functionalities that are similar to C++ `template` can avoid the pitfall of exponential expansion for every TU.

3.1 Import is binary

As already mentioned, the import feature is not expected to work source to source.

Property 3.3 *The import of a module uses a compiled object.*

For our reference implementation we chose the POSIX archive file format to store some textual information along with the object file.

With that property a software project only has to provide the compiled objects of its modules to its clients. This then enables them to link and execute executables that use that module. But since all interface information is present in the compiled object, they also may compile their own code that would use such a module. In that sense, any binary distribution of modules is, with our model, also a development distribution.

As a direct consequence we have the following rule:

Rule 3.4 *The import graph for any module must be acyclic.*

This property is easily checked by build tools such as POSIX' `make`. It also has the advantage that on a source level we have no need for include guards or similar mechanism that deal with cyclic dependencies.

Lemma 3.5 *There is a topological sort of the import graph.*

Later, in several places we will assume that one such topological sort among all the possible ones is given and fixed.

Lemma 3.6 *All exported symbols of all modules of the import graph are accessible.*

The reason behind this rule is simple: a module `a::b::c` that appends a new naming component `c` to an existing module `a::b` is supposed to extend that module. To extend it,

Listing 3 A module with an initialization function.

```
#pragma CMOD module          proj::something
#pragma CMOD import  thrd  = C::thrd;
#pragma CMOD import  mtx   = thrd::mtx;

#pragma CMOD startup start

#ifndef thrd::NO_THREADS
mtx bkl;
void start(void) {
  mtx::init(&bkl, mtx::recursive);
}
#else
void start(void) {
  // empty
}
#endif
```

- it potentially needs all identifiers that are exported by `a::b`, and
- it must not provide conflicting definitions for objects that are already defined there.

In particular the identifier `a::b::c` could already have a definition or declaration in the parent module `a::b`.

The syntax for importing other modules makes no reference to a source file name for a module, in particular it is not fixed what the separator character for the external file name representation would be. Such a detail is left to the implementation. Our reference implementation separates file name components with a hyphen, *e.g.* the source file name for the `C::io` module is `C-io.X`.

3.2 Module initialization and cleanup

The strict dependency relation between modules given by the import graph also lets us provide secondary features, module initialization and cleanup. Such features may be helpful for file scope objects that need to be initialized dynamically; C does not allow to call functions to initialize statically allocated objects. This feature is achieved by a **startup** directive:

```
#pragma CMOD startup start
void start(void) { ... }
```

If such a directive is given, the function is guaranteed to be executed *before* accessing any symbol (variable or function) of the module, and *after* all such **startup** functions are executed for all imported modules. Such a function must have a prototype as indicated in the example.

Property 3.7 *The functions listed in **startup** directives of all modules of a program are executed before the **entry** function, and this in an order that is consistent with the import graph.*

Consider Listing 3 as an example. Here a module that needs a global mutex variable bkl to regulate access to some of its functions initializes that mutex with a non-default property mtx :: recursive. If *e.g.*, the module C :: thrd :: mtx would need by itself a similar form of initialization, such an initialization is done before. By that the call to mtx :: init is always guaranteed to operate in a well defined context.[6]

Our approach has two main advantages over the techniques that are currently available for C or C++. The C standard already offers the type **once_flag** that can be used to launch a function dynamically, when the first use of an object occurs. That dynamic check is repeated at each execution of that same code. The presence of a conditional function call can inhibit some optimization opportunities.

Our approach has only an overhead at startup of the application, where all the **startup** functions are called once and for all. Thereafter, during the execution of the program itself there is no overhead at all and an application has the guaranty to start all its operations in a well defined state.[7]

Our approach is also different from the approach in C++. There, statically allocated objects can be initialized dynamically with a function, but the execution order of these initializations is not prescribed by the standard. Thus C++ makes it quite difficult to provide a consistent initialization of static objects, as soon as there are dependencies between them.

To complement this approach for initialization we also propose two other interfaces, named after the C library functions **atexit** and **at_quick_exit** that they use under the hood.

```
#pragma CMOD atexit module_atexit
#pragma CMOD at_quick_exit module_at_quick_exit
void module_atexit(void);
void module_at_quick_exit(void);
```

These are executed when the program terminates.

Property 3.8 *The functions named in **atexit** and **at_quick_exit** directives of all modules of a program are are inserted as **exit** and **quick_exit** handlers before the **entry** function (as if inserted with the **atexit** and **at_quick_exit** standard library functions). The insertion is such, that the functions are executed in inverse order of the corresponding **startup** functions.*

[6] All of this is only done conditionally, C :: thrd :: NO_THREADS will be explained in Section 6.3.

[7] If an application has to perform an expensive initialization that would be too costly to impose to all its users, it may still use the **once_flag** mechanism, instead.

4 Code sharing

C's **#include** directive is not only a tool to share interfaces between TU, but can also be used to share almost any form of C code. To provide the same expressiveness in Modular C we have to provide mechanisms for that, too.

A first mechanism that C natively supports are inline functions, that is functions for which the function body is made visible to any importer. The next section briefly describes how Modular C uses and facilitates this feature. Then we introduce and develop a feature coined *code snippets* that is meant to replace inclusion of arbitrary C code.

4.1 Inline functions

In traditional C, classifying a function with **inline** ensures that it can be placed in a header file without creating symbol definition conflicts between different TU. This feature is intended to ease optimization, because the code of such a function can then be integrated in place (*inlined*) by its users. Thus optimization techniques such as special case analysis and constant propagation can be applied to the body of the inlined function.

The engineering of inline functions is a bit tedious in traditional C. They have to be moved to a header file (to be visible by others) and a special "instantiation" has to be provided in one TU (to emit the external symbol). Modular C avoids such code movements and additions:

Property 4.1 *A function definition with **inline** specifier is made visible to all importers.*

4.2 Snippets

We propose an additional formalism to share code that will give rise to *different* functions (macros, types, constants ...) in the context of the *importer*. The main property of a **snippet** directive is that the code that follows after it is not compiled within the containing module but it is only injected into a module that imports it explicitly, much as the traditional X macro strategy would do.

Property 4.2 *Other than for the replacement of some identifiers specified below, all code that follows a **snippet** directive is injected* as is *at the point of an* explicit ***import*** *directive.*

Compared to C++ templates, the replaced identifiers play the role of template parameters. But in contrast to that, a snippet only provides a textual injection of code into the importer where the identifiers in question are textually rewritten as specified.

Let us start with a simple module `C::snippet::init` whose only purpose is to share code for a uniform `init` function with the importer.

```
1 #pragma CMOD module init = C::snippet::init
2
3 /* All code hereafter is integrated by the importer. */
4 #pragma CMOD snippet  T = complete
```

```
5 T* init(T* x) {
6   if (x) *x = (T){ 0 };
7   return x;
8 }
```

The **snippet** directive introduces a local name, here T, that stands for the name of the importing module. It is the first case of an identifier inside the snippet that is rewritten before the snippet code is injected into its target.

As we have seen in Section 2.4 above, such a module name may represent a type, but may also represent a function, macro, etc or no particular feature at all. On the right hand side of the = the coder of the snippet may specify what (and if) some particular property is expected from the importing module.

In the example, T should clearly be a type. Here, the indication **complete** on the right requests that this module name corresponds to a complete type.[8] The function that is defined tests if its argument x is a valid pointer and then initializes the object with the zero-initialized *compound literal*.

Property 4.3 *The identifier (if any) that is specified in the **snippet** directive is replaced by the name of the importing module before insertion.*

Since all the code after a **snippet** directive forms the snippet, there can only be one such directive:

Rule 4.4 *There may be at most one **snippet** directive in the source of a module.*

Listing 4 shows an example for three different ways to import snippets. For the first, module C::**snippet**::init is imported without an abbreviation. All the symbols from the snippet are injected into the importer. Here, this is just one function, init. It is now visible as a member of this module, namely as proj::structs::toto::init. The identifier T stands for the name of the importer and so the function has the prototype:

```
proj::structs::toto* proj::structs::toto::init(proj::structs::toto*);
```

The mechanism for the two other **import** directives that actually have abbreviations are described below.

4.3 Conflicting names in snippets

Generally, importing snippets as described so far may lead to naming conflicts if two different imported snippets use the same local identifier to name two different features. This can be mitigated by using the abbreviation feature of Section 2.3. Identifiers that originate from a named import receive that prefix as an additional name component, separated by the **composer**. For the second import in Listing 4, the module C::**snippet**::alloc has the abbreviation bare. All symbols that are injected from the its snippet will be prefixed by that.

[8] For C, a type is complete, if it permits to define a variable. Incomplete types are *e.g.* **void**, arrays types without size of the form **double**[], or forward declared **struct** tags.

Listing 4 A module that imports code snippets for two functions and a specialized type.

```
#pragma CMOD module toto = proj::structs::toto

#pragma CMOD declaration
struct toto {
  C::size counter;
  /* your favorite data */
};
#define bare□init init

#pragma CMOD definition
/* Import three snippets. */
#pragma CMOD import             C::snippet::init
/* A snippet imported with an abbrevation. */
#pragma CMOD import bare      = C::snippet::alloc
/* A snippet with 3 slots that are filled. */
#pragma CMOD defill vec::size  = 23
#pragma CMOD fill   vec::vType = vec23
#pragma CMOD fill   vec::bType = toto
#pragma CMOD import vec       = C::tmpl::vector

static C::size _Atomic running = 0;
toto* alloc(void){
  toto* x = bare□alloc();
  if (x) x↦counter = running++;
  return x;
}
vec23 myState = { 4711, };
```

It is implemented in a similar way as C::**snippet**::init, see Listing 5, but provides two snippet symbols init and alloc. The first is only declared, the second is also defined. Note that the module C::**snippet**::alloc does not know much about the context in which the snippet function alloc will be placed by the import mechanism. It formulates an interface contract for init and enforces that the name of the importing module must be a complete type and that it must implement a function init with the specified prototype. Also note that this second part of the interface contract does not need special syntax. A standard C declaration of a function prototype is sufficient for its specification.

Because we have given an abbreviation when importing this module, the importer changes the imported identifiers to bare□init and bare□alloc, respectively. If it weren't for the **#define**, there would be four functions. But with the define, proj::structs::toto::bare□init is replaced by proj::structs::toto::init and effectively the alloc snippet will use the latter for its implementation of bare□alloc.

Listing 5 A snippet module for a simple allocation function.

```
#pragma CMOD module alloc = C::snippet::alloc
#pragma CMOD import lib   = C::lib

/* Integrate as-is to the importer. */
#pragma CMOD snippet  T = complete

#pragma CMOD declaration
extern T* init(T*);

#pragma CMOD definition
T* alloc(void) {
  return init(lib::malloc(sizeof(T)));
}
```

Up to now the two imports that we discussed have had the effect of the following four declarations.

```
#define proj::structs::toto::bare□init proj::structs::toto::init
proj::structs::toto* proj::structs::toto::bare□alloc(void);
proj::structs::toto* proj::structs::toto::init(proj::structs::toto*);
proj::structs::toto* proj::structs::toto::alloc(void);
```

Observe that the function alloc that is defined by the module itself may even use bare□alloc, internally. No naming conflict occurs.

Property 4.5 *Snippets can be imported several times into the same module M if each* ***import*** *uses a different abbreviation.*

4.4 Slots

As we have seen in the last example, naming **import**s and redefining some prefixed names can be used to parameterize snippets.[9] By that snippets can even be imported several times with different prefixes and different definitions for some of their identifiers. But this mechanism is cumbersome, does not scale very well and lacks automatic verification.

To ease the parameterization of snippets there is a third mechanism with **slot**, **fill** and **defill** directives, see Listing 6 for an example. Here, the snippet is parameterized with three identifiers, its *slots*. A **slot** directive names a snippet-local identifier, that will be replaced as specified when the snippet is inserted. E.g Listing 6 has slot bType. On the importing side, Listing 4, this slot is *filled* with the identifier toto. It is important to note that this mechanism replaces one identifier (the slot) by another (the fill).

[9] Here it the function init served as a parameter to bare□alloc.

Listing 6 The first lines of a module with a snippet with three slots.

```
#pragma CMOD module C::tmpl::vector

/* A default value for the size slot, below.        */
#define size 1

/* No requirement concerning the importer.          */
#pragma CMOD snippet  none
/* Three slots parameterize the code.               */
#pragma CMOD slot vType  = complete
#pragma CMOD slot bType  = complete
#pragma CMOD slot size   = ice       /* has default */

#pragma CMOD declaration
typedef bType vType[size];
...
```

Rule 4.6 *A **fill** directive replaces the slot with the identifier after the = sign.*

A **defill** directive, "**define fill**", is a specialized form of a fill. Instead of an identifier, its right specifies a token sequence that serves as a macro replacement. *E.g.* the **defill** for vec::size from above is equivalent to

```
#define A_UNIQ_ID = 23
#pragma CMOD fill vec::size = A_UNIQ_ID
```

We will not further go into details of this mechanism.

Much as for the snippet name, a *fill* identifier could be any identifier with a variety of properties. As before, the coder of the snippet may request some specific properties. Here, two of the slots are expected to be rewritten to types (**complete**) and the other to an *integer constant expression* (**ice**). Listing 4 shows how the three slots in the specification of C::tmpl::vector are filled:

C::tmpl::vector	proj::structs::toto
vType	vec23
bType	toto
size	value 23

or with their full external names

C::tmpl::vector	proj::structs::toto
C::tmpl::vector::vType	proj::structs::toto::vec23
C::tmpl::vector::bType	proj::structs::toto
C::tmpl::vector::size	value 23

Generally, all slots of all directly imported modules must be filled. If for example bType would not be filled by an importer, a syntax error for the use of the identifier

C::tmpl::vector::bType would abort the compilation. But this mechanism can also be used to provide a default value for a slot. The module C::tmpl::vector globally defines the macro C::tmpl::vector::size. If an importer omits the slot size, the value 1 from that global definition is used.

This approach with snippets has several advantages over the *X macro* technique:

- The possibility of specifying certain requirements for the replacement.
- No consistent header declarations have to be maintained.
- There is no need to use conditional (**#ifdef**) compilation and **#undef**.
- Slots can have default values.
- A subsequent use of a snippet will not be polluted by symbols of a previous use.

Definition 4.7 *A snippet is called a* template *if all the features that it specifies for the importer are named through* ***slot*** *directives.*

Or, stated otherwise, it is a template if it does not pollute the name space of the importer.

Besides **complete** and **ice**, the declaration of a **slot** allows many other specifications, for example **global** (may not be overwritten), **intern** (for a per-import unique identifier), **startup** and **atexit** for startup and shutdown functions.

4.5 Reach of snippets

Even though we have transitivity of the import graph, name spaces of modules are completely separated and don't pollute each other. This property should be preserved if we use snippets. Therefore we ensure that any name that is imported from a snippet is indistinguishable from an identifier that was directly defined in the module itself. The snippet is fully "absorbed" by the module that imports it:

Rule 4.8 *Any identifier N in B whose definition originates from a snippet-import for A in B is seen by any importer of B as if it were genuinely defined by B.*

If we would integrate snippets by transitivity, coding with them would become quite complicated. E.g. for Listing 4 this would have the consequence that a module D that imports proj::structs::toto would also define its own functions init and alloc. Consequently, the name spaces of D would be polluted with identifiers of which it has no control. In fact, the name space of a module M that indirectly imports a snippet from A should not be affected by any future changes of A, possibly without the author of M even being aware of the import.

Rule 4.9 *The snippet of a module A is only integrated by an explicit import of A.*

By this, our strategy also avoids the overhead of repeated compilations of transitively imported snippets, as they occur for C++'s template. As already mentioned above, for C++ intermediate instantiations must be compiled for any TU that transitively includes a recursive template. In contrast to that an instantiation of a *Modular C* **snippet** must be explicit (by an explicit **import**) and is only compiled *once* for a whole project.

5 A formal description

One of the advantages of our approach is that much of it can be described by text rewriting that allows for a formal proof of its validity.

5.1 Source reorganization and stability

As a first step we have to define the properties that we expect of a module:

Definition 5.1 *For a module M the* completed source *M^+ is M to which all imported snippets are integrated in the order they appear in* ***import*** *directives.*

Definition 5.2 *A module M is* valid *iff all of the following hold:*

1. *M fills all slots of imported snippets.*
2. *M uses a set of mutually distinct valid identifiers for all abbreviations and slots.*
3. *Besides the fact of using composed identifiers, M^+ is syntactically correct C code.*
4. *No declaration of an exported feature of M^+ depends on a non-exported feature.*
5. *After preprocessing,[10] the feature declarations of M^+ are in dependency order.*
6. *Each identifier used by M^+ is defined exactly once, either by M^+ or by a module that it imports.*
7. *M^+ uses local and imported identifiers, including tag names, according to their declaration.*

With *Modular C* we want to reduce the need to explicitly specify header information and to have such information extracted automatically by a tool. Such an extraction would be very difficult, if we would allow arbitrary dependencies between macro definitions, type declarations and object declarations. Therefore we introduce a separation of a source in three conceptually different parts, for macro definitions, declarations (types, objects and functions) and definitions (objects and functions).

Definition 5.3 *For a conforming C source P, $\mathscr{P}(P)$, the* preprocessor extract*, is the collection of all logical lines of P that are prefixed with* ***#*** *and that are not* ***#pragma*** *or* ***#line*** *directives.*

In the two remaining extracts, **inline** functions are treated specially. The intent of C's **inline** feature is to make the definition of a function visible to all users of the function. Therefore, all **inline** functions are placed in the declaration part and possible additional declarations are removed.[11]

Definition 5.4 *For a conforming C source P, $\mathscr{D}(P)$, the* declaration extract*, is obtained from P by*

- *preprocessing it,*

[10] C's compilation phase 4, see [8].

[11] In fact for those **inline** functions that have external linkage such an additional declaration would force an instantiation of the corresponding linker symbol.

- *adding canonical* ***typedef*** *for all tagged* ***struct*** *and* ***union*** *types to the beginning,*
- *adding canonical* ***typedef*** *for all tagged* ***enum*** *types immediately after their definition,*
- *replacing all data object definitions and all definitions of functions that are not* ***inline*** *by declarations that use appropriate* ***extern*** *or* ***static*** *linkage specification,*
- *removing all* ***extern inline*** *declarations that are not definitions.*

The third part contains the "rest" of the code of a source P, that is mainly object and function definitions. Observe that this definition only removes macro definitions, but leaves conditional preprocessor directives (**#if**/**#else**) in place.

Definition 5.5 *For a conforming C source P, $\mathscr{I}(P)$, the* definition extract*, is obtained from P by*

- *removing all logical lines that form* ***#define*** *of* ***#include*** *directives,*
- *removing all file scope* ***typedef***, ***struct***, ***union*** *or* ***enum*** *definitions,*
- *removing all definitions of* ***static inline*** *functions,*
- *replacing all remaining definitions of* ***inline*** *functions by appropriate* ***extern inline*** *declarations that are not definitions.*

Finally, we need to be able to tell, when two conforming C programs are equivalent. Whereas this is a difficult problem in general, we can restrict ourselves to programs where all features are defined by the same token sequence, but may perhaps appear in a different order and may be complemented by declarations that are not definitions.

Definition 5.6 *Two conforming C sources P and P' are* token equivalent*, if, after preprocessing, for any definition $A \in P$ there is $A' \in P'$ that is defined by the same token sequence, and, vice versa, for any definition $A' \in P'$ there is $A \in P$ that is defined by the same token sequence.* [12]

C's syntax and semantic are quite restrictive:

Lemma 5.7 *Two sources that are conforming and token equivalent are functionally equivalent and give rise to the same set of external symbols.*

This can be seen by using a *feature dependency graph* that links any global identifier to the terms that it uses in its declaration. The fact that the two sources are conforming and token equivalent implies that their feature dependency graphs are isomorphic: if a declaration of an identifier `A` uses another identifier `B`, `B`'s declaration must precede `A`'s in any conforming code.

Definition 5.8 *A conforming C source P is called* stable *if all the following hold:*

1. *P and the concatenation $\mathscr{P}(P) \cdot \mathscr{D}(P) \cdot \mathscr{I}(P)$ are token equivalent.*
2. *It does not contain* ***#include*** *directives.*
3. *It does not contain explicit references to C library functions or objects.*

[12] This correspondence does not necessarily define a bijection between instances of definitions, because **typedef** definitions may appear several times in a conforming C source.

4. *If defined,* ***main*** *has prototype* ***int main(int*** *argc,* ***char****[argc+1]) and no execution can reach the terminating } of the function body.*
5. *It does not use preprocessor operators* ***#*** *and* ***##****.*
6. *It does not use the reserved identifiers* ***__func__****,* ***__LINE__*** *or* ***__FILE__****.*

Stability for a C source may look like a strong restriction, but it isn't much in our view. (2) and (3) stem from the fact that *Modular C* replaces **#include** directives by **import**. To be comfortable with the concept, the reader may just assume for the following discussion that all **#include** directives have already been applied to the code.

By the requirement in (1) and (2), all conditional compilation (with **#if** etc) only depends on predefined constants and on macros that are defined in P itself. This requirement imposes some discipline for the reuse of macros inside **#if**/**#elif** evaluation, and interferes with cases were a macro name is also used for a function. In most cases code with such macros can be sanitized. *E.g.* in the following, the declaration of the function and the macro definition cannot be interchanged.

```
double fabs(double a);
#define fabs(X)           \
   _Generic((X),          \
            default: fabs, \
            float: fabsf)
```

Thus this short code isn't stable. But the code can easily be sanitized:

```
#define fabs(X)           \
   _Generic((X),          \
            default: fabs, \
            float: fabsf)
double (fabs)(double a);
```

Placing the function name inside parenthesis for the declaration (and other places where it shouldn't expand) protects the identifier from macro expansion.

Another requirement in (1) that is perhaps not obvious, is that it forbids the definition of **struct**, **union** and **enum** inside object or function declarations or in macro expansions. Consider the following definition:

```
static struct { unsigned a; } x = { .a = 0 };
```

This would give rise to the following declaration in the declaration extract:

```
static struct { unsigned a; } x;
```

Having two **struct** definitions in $\mathscr{P}(P)\cdot\mathscr{D}(P)\cdot\mathscr{I}(P)$ would be erroneous: if the **struct** is without tag name (as in this example) the two types are considered different and we have a redeclaration of `x` with a different type. If on the other hand we would add a tag name, we would have two definitions for the same **struct** type.

Such situations can be avoided easily by having all type definitions separated:

```
typedef struct { unsigned a; } type_of_x;
static type_of_x x = { .a = 0 };
```

Such code is then easily separated into declaration and definition extract.

Restrictions (5) and (6) have to do deal with "identifier aware" programs that would change their behavior if identifiers are mangled. Whereas the latter restriction is mostly harmless, (5) may constrain the use of sophisticated macro packages. A program that would want to use these features would have to adjust longer output strings (for **__func__** *e.g.*) and prove equivalence of token concatenation and similar features with mangled identifiers.

Observe, that the use of canonical **typedef** in Definition 5.4 implies that a tag name cannot be reused as another normal identifier, such as *e.g.*. POSIX does for **struct** stat and function stat. We view the difference of tag names and identifiers as a historic artifact, that has not much importance in modern code.

Also, remember that a conforming C source can be compiled into an object file, but may perhaps not lead to a valid executable: the source may refer to external identifiers that must be linked from other TU.

5.2 The replacement procedure

Figure 2 shows a formalized replacement procedure to compile a module into an archive file. It involves replacements of abbreviations, cut and paste of text chunks and name mangling. In addition to simple text replacement we need to generate the import graph and a topological sort of it (3), some code generation for some stub functions (9) and, most difficult, declaration extraction (7e).

The stub functions can be generated similar to Listing 7. By that each module exports an initialization function with local name _ModuleInit and eventually another one that contains a conventional **main**. The first calls all generated _ModuleInit functions for all imported modules (in topological order), and then (if any of the three are defined) inserts module_atexit and module_at_quick_exit in the appropriate queues and calls the modules's own module_startup function. Here, as an example, this uses the C standard data type **once_flag** to guarantee race freeness of the overall initialization. Implementations that don't provide the C11 threads interface would have to use a mechanism that makes this at least asynchronous signal safe, *e.g,* by using a flag of type **sig_atomic_t**.

In addition, if the module defines an **entry** function, EntryFunction, say, we create a traditional **main** function as entry point to the program that just calls the module's _ModuleInit and then jumps to the modules EntryFunction.

In a first step, we have to show when mangling does not change the validity of a given C source. For the C compilation itself, before linking to an object file, the only thing that could prevent a valid mangling is *identifier inspection*, that is if a program refers and acts according to identifier names. There are only few tools in C that could be used for that purpose: preprocessor manipulations, when macro expansion uses the operators **#** or **##**, or a use of the reserved identifier **__func__** for a function name.

The following lemma follows directly from the definitions:

Lemma 5.9 *Let f be a mangling and P be a conforming C source without composed identifiers and* Modular C *directives. Then P is stable iff $f(P)$ is stable.*

1. Replace all occurrences of the string from the **separator** directive by ::.
2. Extract all **separator**, **module**, **import**, **slot** and **fill** directives from the code.
3. Generate a top-sort of the import graph and store it in "MODNAME—imports.txt".
4. Replace abbreviations from **module** and **import** directives.
5. Split the code at a **snippet** directive into a *regular* and a *snippet* part.
6. In the snippet, if any, prefix all slot identifiers by MODNAME:: and replace the snippet name by _Importer. Store the snippet in a file "MODNAME—snippet.c".
7. With the regular part:
   a. To produce M^+, let S_i, $i = 0, \ldots, n-1$ be the snippets for M, listed as their appearance in **import** directives. For all $i = 0, \ldots, n-1$:
      - Replace slot names in S_i by their **fill** definitions.
      - Replace _Importer by MODNAME.
      - Replace the corresponding **import** directive with S_i.
   b. Append predefined internal function and object definitions to M^+.
   c. In M^+, prefix all remaining file scope identifiers by MODNAME.
   d. Mangle all composed names to C source $P = f(M^+)$.
   e. Extract $\mathscr{P}(P)$, $\mathscr{D}(P)$ and $\mathscr{I}(P)$ and store $\mathscr{P}(P) \cdot \mathscr{D}(P)$ as file "MODNAME.h"
   f. In top-sort order, let $\mathscr{H}_0, \ldots, \mathscr{H}_{n-1}$ be the headers of imported modules. Compile the concatenation $\mathscr{H}_0 \cdots \mathscr{H}_{n-1} \cdot \mathscr{P}(P) \cdot \mathscr{D}(P) \cdot \mathscr{I}(P)$ to an object file "MODNAME.o".
8. Store the created files in an archive "MODNAME.a".
9. Create the necessary stubs for entry point and initialization, compile them into separate object files and include them in the archive.

Figure 2 The high level description of the replacement procedure for module MODNAME.

In the following we will not specify the mangling f explicitly anymore. We will assume that a such a mangling f is given and fixed.

We are now able to prove the first property of our replacement procedure. It follows directly from the lemma above and from the fact that most of the replacement procedure does not apply if there are no *Modular C* directives.

Lemma 5.10 *Let P be conforming without composed identifiers or* Modular C *directives. Then P is stable iff the replacement procedure compiles P to a valid archive file.*

Now if we have a valid archive file, we just have to worry what happens to the special functions that define entry points or similar.

Corollary 5.11 *Let P be as above and stable such that it has an **entry** but no **startup** or **atexit** directives, and such that it makes no reference to any external symbols. Then the archive file can be linked to a valid executable that is functionally equivalent to P.*

Now that we are able to compile some ordinary C code, we start to add *Modular C* directives. The first is the **module** directive itself:

Listing 7 Pseudo code for the stub functions that are generated during the replacement procedure: composed identifiers here have to be replaced by mangled ones, C library code should be replaced with code that implements the same effects directly with identifiers that are strictly reserved.

```
/* code compiled in a separate .o file, if needed */
static void _ModuleInitOnce(void) {
  /* Init imported modules in topol. order */
  IMPORT1::_ModuleInit();
  IMPORT2::_ModuleInit();
  ...

  /* Any of the following three is omitted if
     not provided in the module. */
  atexit(MODNAME::module_atexit);
  at_quick_exit(MODNAME::module_at_quick_exit);
  MODNAME::module_startup();
}
void MODNAME::_ModuleInit(void) {
  static once_flag = ONCE_FLAG_INIT;
  call_once(&once_flag, _ModuleInitOnce);
}

/* code compiled in a separate .o file, if needed */
extern void MODNAME::_ModuleInit(void);
extern int MODNAME::EntryFunction(int argc, char*argv[argc+1]);
int main(int argc, char*argv[argc+1]){
  MODNAME::_ModuleInit();
  return MODNAME::EntryFunction(argc, argv);
}
```

Lemma 5.12 *Let M be a conforming module without composed identifiers and as only* Modular C *directive a* ***module*** *directive that names the module with a non-composed valid identifier* `MODNAME`.

1. *The following are equivalent:*
   - *M is stable.*
   - *$f(M)$ is stable.*
   - *The replacement procedure compiles M to a valid archive file.*
2. *If M is stable,* `"MODNAME.h"` *is a valid conventional C header file that correctly declares all exported definitions of $f(M)$.*
3. *If M is stable and has an* ***entry****, the archive file can be linked to a valid executable that is functionally equivalent to the original program defined by M.*

5.3 Stitching modules into one project

Modules are not conceived as isolated, monolithic software, but as components that integrate into a whole system of other modules. To guarantee properties of a particular module M we will have to argue about a long list of modules that are imported by M. This list will usually contain all the modules that compose the C library and eventually other more project specific modules.

Definition 5.13 *An ordered set of valid modules $\mathscr{M} = (M_0, \dots, M_{n-1})$ forms a* consistent project *if the following hold:*

1. *All module names are mutually distinct.*
2. *All module imports stay in the set $\mathscr{M}$.*
3. *The order in $\mathscr{M}$ is consistent with all import graphs.*

Note that (2) implies that for all composed module names $N_0 :: \cdots :: N_{\ell-2} :: N_{\ell-1}$ there must also be a module with name $N_0 :: \cdots :: N_{\ell-2}$ in the set. Also, (3) implies that all import graphs for all modules in $\mathscr{M}$ are acyclic.

Now we are able to formulate the principal property of *Modular C*, namely that compilation of modules such as described by the replacement procedure is equivalent to the compilation of well specified regular TU.

Theorem 1 *Let $\mathscr{M} = (M_0, \dots, M_{n-1})$ be a consistent project with completed sources $\mathscr{M}^+ = (M_0^+, \dots, M_{n-1}^+)$. The following are equivalent.*

1. *For all $i < n$, M_i is a valid module such that all features to which M_i^+ refers are either defined by M_i^+ or exported by $M_0^+, \dots, M_{i-1}^+$.*
2. *For all $i < n$, $f(M_i^+)$ is a conforming TU such that all file scope identifiers are either defined by $f(M_i^+)$ itself or declared by some $\mathscr{P}(f(M_j^+)) \cdot \mathscr{D}(f(M_j^+))$ for some $j < i$.*
3. *For all $i < n$, the following concatenation of sources is stable:*

$$\mathscr{P}\left(f\left(M_0^+\right)\right) \cdot \mathscr{D}\left(f\left(M_0^+\right)\right) \cdots \mathscr{P}\left(f\left(M_{n-1}^+\right)\right) \cdot \mathscr{D}\left(f\left(M_{n-1}^+\right)\right) \cdot f\left(M_i^+\right)$$

4. *For all $i < n$, the replacement procedure compiles M_i to a valid archive file.*

As a first step for a proof, consider the case that none of the module has snippets.

Lemma 5.14 *Let $\mathscr{M} = (M_0, \dots, M_{n-1})$ be a consistent project without snippets, and $\mathscr{H}_0, \dots, \mathscr{H}_{n-1}$ be the headers files produced for them by the replacement procedure in Step 7e. The following are equivalent.*

1. *For all $i < n$, M_i is a valid module such that all features that it refers to are either defined by M_i or exported by $M_0, \dots, M_{i-1}$.*
2. *For all $i < n$, $f(M_i)$ is a conforming TU such that all file scope identifiers are either defined by $f(M_i)$ itself or declared in $\mathscr{H}_0 \cdots \mathscr{H}_{i-1}$.*
3. *For all $i < n$, the concatenation of sources $\mathscr{H}_0 \cdots \mathscr{H}_{i-1} \cdot f(M_i)$ is stable.*
4. *For all $i < n$, the replacement procedure compiles M_i to a valid archive file.*

Proof Sketch: A proof can be done by induction on n. For $n = 1$ the statement follows directly from Lemma 5.12.

Suppose that the statement holds for $n_0 - 1$. For the induction step, observe that the statement holds for $i < n_0 - 1$ because of the induction hypothesis. So it remains to be proven the statement holds for $i = n_0 - 1$. But then (3) suffices to show that M_i is valid and thus the equivalence holds. □

Proof of Theorem 1: With the previous lemma it suffices to show that the replacement procedure is also compatible in the presence of snippets. This follows from the following

- All abbreviation replacements are effected before splitting regular part and snippet.
- Identifiers from **slot** directives are effectively protected by renaming them to a name that is local to the exporting module. On the importer's side, these exporter-names are then "filled" with the correct names when importing snippet S_i in (7a).
- The `_Importer` identifier stands for the name of the importer and is replaced consistently when snippets are inserted. □

Other properties hold for the less central features of *Modular C*. We mention the following without proof:

Theorem 2 *If during initialization none of the* ***startup*** *functions exits preliminary through* ***exit***, ***thrd_create***, ***thrd_exit***, ***longjmp*** *or similar nor catches a trap and if all of them finish eventually, the overall initialization procedure is finite, race free and asynchronous signal safe.*

6 A structured view of the C library

The C library as it stands has the disadvantage that it drags a lot of historical baggage. That often makes its interfaces difficult to apprehend. *Modular C* provides the opportunity for a structured view to C library features, by still remaining backwards compatible. Strict encapsulation, makes such an approach feasible. We propose to keep the old interfaces accessible through modules. We foresee to do this in three levels:

1. Modules prefixed with `C::std` provide the symbols in snippets: *e.g.*, `C::std::lib` for `stdlib.h`. They remain source compatible when previous **#include** directives are replaced with such imports: the imported identifiers are accessible through the same local names as before. E.g, there would exist a local name **rand** that resolves to the symbol in the C standard library.
2. Modules with the same name but without `std` in the name export the identifier through external declaration, such as `C::lib` for `stdlib.h`. This would provide **rand** through the identifier `C::lib::`**rand**.
3. To achieve real modularity we also provide a structured view to the same functions and types through different modules.

To see the latter let us look into the interface of a module C::ulong that interfaces all that is to know about the standard type **unsigned long**. The tool Cmod_tabular that we have implemented for our reference implementation lists the following identifiers:

Terminal

```
> ./Cmod_tabular C#ulong.a
| C |ulong |++++++++++++++++++++++++++++++++
| C |ulong |abs | extern
| C |ulong |ALIGN | evaluated; macro
| C |ulong |max | <-C#snippet#minmax; macro
| C |ulong |MAX | evaluated; macro
| C |ulong |maxv | <-C#snippet#minmax; extern
| C |ulong |min | <-C#snippet#minmax; macro
| C |ulong |MIN | macro
| C |ulong |minv | <-C#snippet#minmax; extern
| C |ulong |RANK | evaluated; macro
| C |ulong |SIZE | evaluated; macro
| C |ulong |ulong | ->ulong
| C |ulong |WIDTH | macro
| C |ulong |zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
```

Other modules for the standard integer types implement the analogous identifiers. **MAX** corresponds to the type's maximum value, here **ULONG_MAX**, MIN to its minimum, **abs** to a function that returns the absolute value etc. For real and complex floating point types we applied a similar, adapted, strategy and provide interfaces in modules such as C::real::**float** or C::**complex**::**double**.

For integer types, we see that the functionality that is provided is already much richer than what the current C library provides. We have access to object-like macros for the size of the type, its integer rank, its width and to functional macros such as the minimum and maximum of a finite set of values (min and max) or to functions operating on arrays (minv and maxv).

This systematic approach allows us to group common features for different types together and provide type generic macros, implemented with C11's **_Generic** keyword, for all of them. E.g, the module C::math contains type generic macros that work for all arithmetic types such as **abs** or for all real types such as min and max. In our approach the implementation most of these type generic macros is almost trivial:

```
#define abs(X)                                  \
_Generic((X)+0,                                 \
 C::real::float:       C::real::float::abs,     \
 C::real::double:      C::real::double::abs,    \
 /* some more ... */                            \
 C::complex::ldouble: C::complex::ldouble::abs,\
 C::unsigned:          C::unsigned::abs,        \
 /* some more ... */                            \
```

```
9  C::llong:          C::llong::abs)(X)
```

Here the implementation makes the choice to select the expression only on the *promoted* type (the `+0` does that). By that no **default** branch of the generic expression is necessary and the compiler indicates an error if this macro is used with another type, as *e.g.*, a pointer type.

Let us now list some groups of interfaces that should receive a special treatment. This is not intended to prescribe a particular way how this must be done, but to demonstrate the potential of our approach. The exact realization of this project needs a lot of discussion and ideas from the community. In its current state, our proposal has more than 160 modules for the C library.

6.1 Basic types

As indicated we foresee one module for every basic type, and for convenience we add one for each of the four semantic **typedef**, namely **size_t**, **ptrdiff_t** and [u]**intmax_t**. Since with modules we don't have the problem of uniqueness of identifiers we propose to drop the suffix **_t** for the names of these types for this structured interface.

All sensible functionality of these types is interfaced and regrouped into type generic macros in `C::math` where possible. This concerns also all the mathematical functions from `math.h` or `tgmath.h`. `C::math` provides type generic interfaces for all of these, with **default** branches resolving to the variant in `C::real::`**double**.

6.2 String processing

A module `C::str` provides type generic interfaces for the string functions that in the current C library have prefix `str` (regular strings) or **wcs** (wide character strings). Some of the original functions have the additional problem that they drop **const**-qualifiers where they shouldn't.

```
double strtod(char const* s, char** end);
```

This function, *e.g.*, scans a **const**-qualified string `s` for a **double** number. If `end` is provided, after the scan `*end` will point to the ending position inside the same string. By that the **const**-qualification is lost. With type generic macros we may easily avoid that problem by using two different functions (respectively 4 including **wcs** variants)

```
double strtodm(char* s, char** end);
double strtodc(char const* s, char const** end);
```

and glue them together with a **_Generic** expression.

6.3 Feature test

The current approach for feature test of the C library is tedious and error prone. *E.g.* in the transition phase from C99 to C11, many compilers accepted options to switch to C11, whereas the corresponding C library didn't implement all the features, yet. In such situations it is not clear, if the compiler or the C library should provide a

__STDC_NO_THREADS__ macro, for example. So simple code such as the following didn't work.

```
#ifndef __STDC_NO_THREADS__
# include <threads.h>
#endif
```

Both types of errors occurred: an non existing header file causes an error if the compiler implementers haven't yet come to provide the macro, or a feature in an alternative C library would not be detected because the macro was set wrongly. Sorting out different versions and features of the implemented C standard is tedious, and often the only partly viable solution is a long list of special cases for different compilers, libraries and releases.

For *Modular C* we request that all the standard modules must exist. This can either be a module that provides the feature or announces through the feature macro that it is not implemented. We already have seen a use of that in Listing 3. It imports C::thrd, which is, as all imports, unconditional. The code itself then can be tuned with by asking for the preprocessor macro C::thrd::NO_THREAD.

So the least that a provider of a C library has to do is to implement an almost empty module. There, for every optional feature it has just to provide the correct feature test macro.

6.4 Snippets and templates

Most of the modules that implement the features of the C library should not contain snippets, simply because we don't want to have the identifier name space of the user to be polluted by an ever growing amount of rules. Nevertheless we need some modules with snippets.

First of all, legacy code that already uses features of the C standard library should be able to continue to do so. For that purpose we propose a whole hierarchy starting with C::std::. These are designed to inject all features of a corresponding C library header directly to the importer. *E.g.* there is a module C::std::io module that interfaces all features of <stdio.h>. The modules of this hierarchy should only be used during transition from legacy code to *Modular C*.

But, the C standard library should also contain some modules that provide parameterized functions or types. As a convention we have chosen the prefix C::**snippet** for code that just provides an interface that is derived from the module name. If in addition there are **slot** directives that must be specialized by the importer with **fill** directives, we use the prefix C::tmpl.

Currently, there are C::**snippet**::init, C::**snippet**::alloc and C::**snippet**::minmax as already mentioned. Also, there are C::tmpl modules to wrap **qsort** and **bsearch** and to implement simple list data structures.

7 The transition from C to Modular C

7.1 A reference implementation

The relative simplicity of our approach has permitted to implement a first prototype of a compiler front end and C library interface relatively fast; in total we invested some months of qualified work force into it. The implementation was undertaken in a Linux environment and heavily relies on POSIX and Gnu tools: sh for the front end itself, sed and the C preprocessor for text processing, nm and objcopy for the manipulation of binary object files, ar for archive management. [13]

By integrating these tools, we were able to create a front end that should be mostly compiler and C library independent, if both comply to C11. We tested it with different versions of two different compilers, gcc [14] and clang.[15]

For the structure of our interface to the C library see the previous section.The tools that ease the creation of these wrapper modules are described below. We tested them with two different C libraries, glibc [16] and musl.[17]

7.2 Interfacing existing libraries

Modular C as we presented it so far only describes the language extension itself and not how existing software packages should be interfaced such that they fit into the framework. In particular, to provide an operational environment we have to interface the C library of the platform with module interfaces as described above. This has been done by specifying and implementing an additional set of directives, as can be seen in Listing 8.

Our idea is to capture the dependency from a traditional header file in a wrapper module. For relatively simple code, this can be achieved automatically: when fed with a .h file, a special compilation mode is triggered that generates a shallow wrapper module that is suitable to encapsulate the existing TU, .c. Such a wrapper, generated automatically or manually, may use some additional directives as follows.

One or several **mimic** directives may refer to a traditional header files. Information from these can then be extracted textually or by compiling short test programs. We provide an additional set of directives:

- **define** extracts a macro definition.
- **typedef** extracts a type.
- **alias** adds a mangled alias for an existing symbol to the symbol table.
- **defexp** evaluates its expression on the right at compile time and defines a macro with that computed value.

[13] Access to the sources is available at cmod.gforge.inria.fr
[14] http://gcc.gnu.org
[15] http://clang.llvm.org
[16] http://www.gnu.org/software/libc/
[17] http://musl-libc.org

Listing 8 An extract of the implementation of a C library interface.

```
#pragma CMOD module dbl    = C::real::double
#pragma CMOD import minmax = C::snippet::minmax

#pragma CMOD mimic <math.h>
#pragma CMOD mimic <float.h>
#pragma CMOD link —lm
...
#pragma CMOD define  NAN
#pragma CMOD typedef eval   = double_t
#pragma CMOD defexp  PI     = 4.0*atan(1.0)
#pragma CMOD alias   acos
...
#pragma CMOD declaration
typedef double dbl;
double (acos)(double);
```

- **defstruct** extracts predefined **struct** types and ensures size, alignment and exact offset positioning of all fields.

The directive **defexp** (for "define expression") proved to be quite practical. To evaluates the expression that is written on the RHS of the = we create a short C program that is run once to print out the result of the evaluation. By this, platform and type information is extracted only once, during the compilation of the interface modules, and the compiled modules present similar features as precompiled header files.

Most of these tools are implemented such that they encapsulate all necessary information in the compiled module. Aliases that are specified with the **alias** directive are implemented as linker replacement. In our example the (mangled) symbol C::real::**double**::**acos** would act as a link time alias for the C library function **acos**.

Overall, interfacing existing C libraries has minimal compile time and link time overhead but *no run time overhead*. As can be seen in some of the examples, we apply the similar renaming ideas to these directives. E.g, the **typedef** directive extracts the declaration of **double_t** from the standard header and then provides a local definition of a type C::real::**double**::eval. All of this is still surprisingly fast: due to the efficiency of the tools that we use a compilation of the about 120 module files that today compose our C library interface takes only 33 seconds on a modern laptop.

Besides the C library, another example for such an interface to an existing library that we implemented with Modular C is the *Gnu scientific library*, gsl. Here, the main burden was not the generation of wrappers itself, but to come up with a naming scheme that deals with the inconsistencies of gsl.

Listing 9 An example of the module structure for a project migration.

```
#pragma CMOD module list = proj::list
/* Add analogous imports for all standard headers. */
#pragma CMOD import        C::std::io

#pragma CMOD declaration
/* Put all types and macros from "proj_list.h", here. */

#pragma CMOD definition

/* Copy most of "proj_list.c", here.
   Omit extern decl for inline functions. */
```

7.3 A migration path for existing C projects

Migration of existing software projects to new tools is probably never simple. Nevertheless we think that for projects that are already well interfaced and structured the transition to *Modular C* should be feasible.

First of all, Modular C has a "compatibility" mode that ensures that most C files that just include some standard headers can be compiled and produce a valid compiled module. In particular, any legacy **#include** directive is replaced by a proper **import** that emulates it. Such a compiled legacy module uses the external file name for the module name, and can easily be imported by other Modular C code.

There are several possibilities when making some C software accessible for *Modular C*. The first would be to just provide wrapper modules, that is modules that just extract header information from the existing .h files of the project and ensure the proper linking with the unmodified library. This can be done in a similar way as we have described above for the C library and gsl. By following these example, it should be possible for an experienced programmer to interface any reasonable C application header within an hour of work.

The source migration of a C project then can be done as follows. Let us suppose that we have a project that already applies a suitable naming scheme to the interfaces that it provides. Say the overall project is called proj and it contains functions and types that use an underscore convention, *e.g.*, proj_list for a list type, proj_list_init for an initialization function etc. First we have to create a module for the top level of the project:

```
#pragma CMOD module proj
#pragma CMOD link -lproj
```

Then we start creating a module for each functional unit that we want to export. We start with those at the core of our project which don't need to import other parts of the project, see for example Listing 9. A successful compilation produces an archive proj-list.a.

Applying the nm utility on it may complain about some components in an unknown format but then it should show something similar to the lines

Terminal

```
> nm -C proj#list.a
proj#list.o:
0000...0000 W proj_list_proj_list_init
0000...0000 T _C::proj::list::proj_list_init
```

We have two entries in the symbol table, one for the mangled name (translated into C++'s composite naming scheme) and an alias for the conventional name, here proj_list_proj_list_init. We get rid of the duplicated name prefix by removing all proj_list_ prefixes in the source and then by substituting proj_list by list for the type itself.

```
#pragma CMOD module list = proj::list
/* Add analogous imports for all standard
   headers. */
#pragma CMOD import        C::std::io

#pragma CMOD declaration

/* Put all type declarations and macros from "proj_list.h", here
   ↪ . */
struct list {
  list* next;
  /* something */
};

#pragma CMOD definition

/* Copy most of "proj_list.c", here.
   Omit extern decl. for inline functions. */
list* init(list* l) {
  /* do something */
  return l;
}
```

By that the first the nm output should have changed to

Terminal

```
proj#list.o:
0000...0000 W proj_list_init
0000...0000 T _C::proj::list::init
```

By that, have created a Modular C object file that has all correct external symbols and that we can use instead of the one that was previously produced from proj_list.c.

Analogously, we can then proceed to transform all units that will import proj :: list in a similar way. At the end we should find ourselves with a collection of .a files that implement the same symbols as our traditional library. It still is API and ABI compatible with the original library.

Once this transition is completed, all features of Modular C may be used for new developments or for maintenance.

7.4 Implementing new projects in Modular C

To show the practical advantages of our approach, we give the example of two projects that have been entirely implemented with Modular C. This reports on personal experience, so by nature much of the following is biased and should be taken with the necessary precautions. One of the projects, EiLck,[18] had first been developed without a particular need of specific Modular C features, but see below. The second arbogast [3], uses sophisticated features such as snippets, precompilation, code unrolling and expression rewriting, that would have made it difficult to be implemented with traditional C alone.

EiLck (Exclusive–Inclusive Linear Locks) is a standalone library for an inter-thread lock and data access mechanism, that has been implemented newly from the ground in Modular C. It consists of about 30 source files that implement a relatively flat module hierarchy. These

- implement lock objects (on heap, as files or segments) and lock handles,
- hide the implementation specific details (bindings to Linux' futex calls)
- give several simple examples for usage with different thread libraries (POSIX, C11 and OpenMp)
- provide interface stubs for native C and C++ projects.

The implementation of EiLck has been full of agreeable surprises, it went even more smoothly than expected: the traditional approach of dividing the project in functional units was easily modeled and enforced throughout the development. The absence of implicit typing and indirections eases compiler optimization and as a result we have reliable and efficient binaries.

After the core of the new library had been finished, it was easy to add type generic interfaces in Modular C (via snippets) and C++ (via template stubs). In a recent development, a new module eilck :: task and extension directives **amend** and **insert** have been added. They implement a system for automatic memory access classification and task parallelization that would have been difficult to implement directly in C or C++ with the same development and execution efficiency.

arbogast is a new tool for higher order automatic differentiation (HOAD) of C code. It uses a lot of Modular C features such as generic interfaces with snippets

[18] http://cmod.gforge.inria.fr/eilck.html

(Taylor polynomials with different base types) and their explicit instantiation, code replication, or compile-time constant expansion to produce efficient executables. But most importantly, the facility of import replacement allows to write numerical Modular C code that can be used "directly" or as "differentiated" code: a symbolic type `T` in such code can seamlessly be *filled* by using **double** to compute numerical quantities, or by using a Taylor polynomial type to compute a series of derivatives up to a chosen degree. The Modular C directive "**context**" that had not been presented here, allows to mark expressions such that all arithmetic operations (`+`, `*` etc) can be rewritten as Taylor operations if necessary.

All of this would have been difficult to implement in C alone. The principal competitor system uses C++ with operator overloading and is about an order of magnitude slower than `arbogast` on many examples.

8 Conclusion

Our proposition *Modular C* adds modularity and reusability to standard C by introducing a simple naming scheme. It allows a free choice of non-reserved identifiers for all features that are exported by a module and stitches these together by a set of *directives*. The change to C is minimal and in fact we were able to prove the equivalence of code written for *Modular C* to a large subset of valid C code. Related features of a software project can be grouped and accessed together. This is done without introducing new syntactical constructs to the core language, and allows for a grouping of features that is more flexible than classes in traditional object-oriented languages by *e.g.* centering the focus of a module around a single function. Parameterized code reuse is provided through *snippets*, small parts of code that can be injected to an importing module and that are written in conventional C, no extra syntax is required.

Modular C has additional features that fall out from the global approach almost effortlessly, but can nonetheless be helpful tools for future software projects. *E.g.*, there is a simple dynamic module startup and cleanup scheme and also a structured approach to the C library. The latter enables us to avoid some shortcomings of the existing C library interface, namely problems with **const** qualification and lack of type safety for generic functions such as **qsort**.

Last but not least we have shown that these ideas can be implemented effectively on top of existing C platforms and that existing software projects can be migrated to *Modular C* with reasonable effort.

References

[1] M. Blume and A. W. Appel. Hierarchial modularity. *ACM Transactions on Programming Languages and Systems*, pages 812–846, 1999.

[2] Tiobe Software BV, 2015. monthly since 2000. URL: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.

[3] Isabelle Charpentier and Jens Gustedt. Arbogast: Higher order automatic differentiation for special functions with Modular C. *Optimization Methods and Software*, pages 1–25, February 2018. URL: https://hal.inria.fr/hal-01307750.

[4] Bdale Garbee *et al.* (editors). A brief history of debian. Web page, 2017. URL: https://www.debian.org/doc/manuals/project-history/.

[5] S. I. Feldman. Make – a program for maintaining computer programs. In *Unix Programmer's Manual*. Bell Laboratories, 1979.

[6] Doug Gregor. Modules. Apple Inc., Dec 2012. URL: http://llvm.org/devmtg/2012-11/Gregor-Modules.pdf.

[7] D. Richard Hipp. Makeheaders, 1993. URL: http://www.hwaci.com/sw/mkhdr/.

[8] JTC1/SC22/WG14, editor. *Programming languages - C*. Number ISO/IEC 9899. ISO, cor. 1:2012 edition, 2011. URL: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf.

[9] Chris Laffra. Where did eclipse come from? FAQ, 2006. URL: https://wiki.eclipse.org/.

[10] Rob Pike. Go at google. In *Conference on Systems, Programming, and Applications: Software for Humanity, SPLASH '12*, pages 5–6, 2012. see also http://talks.golang.org/2012/splash.article. URL: http://doi.acm.org/10.1145/2384716.2384720, doi: 10.1145/2384716.2384720.

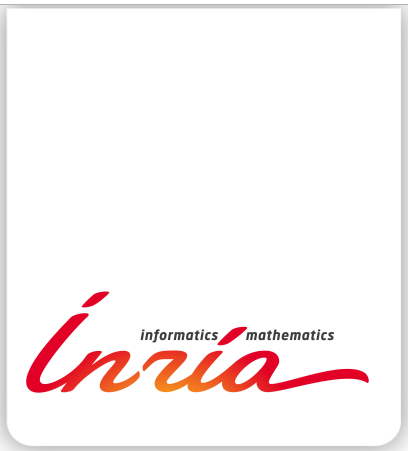
[11] Alastair Reid, Matthew Flatt, Leigh Stoller, Jay Lepreau, , and Eric Eide. Knit: component composition for systems software. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation (OSDI'00)*, volume 4. USENIX Association, 2000.

[12] Keith Schwarz. Advanced preprocessor techniques, 2009. URL: http://www.keithschwarz.com/cs106l/spring2009/handouts/080_Preprocessor_2.pdf.

[13] Saurabh Srivastava, Michael Hicks, Jeffrey. S Foster, and Patrick Jenkins. Modular information hiding and type-safe linking for C. *IEEE Transactions on Software Engineering*, 34(3):357–376, 2008.

[14] Richard M. Stallmann. Emacs – the extensible, customizable, self-documenting display editor. Technical Report 519a, MIT AI Lab, 1981. URL: https://dspace.mit.edu/bitstream/handle/1721.1/5736/AIM-519A.pdf.

[15] Linus Torvalds et al. Linux kernel coding style, 1996. evolved mildly over the years. URL: https://www.kernel.org/doc/Documentation/CodingStyle.



RESEARCH CENTRE
NANCY – GRAND EST

615 rue du Jardin Botanique

CS20101

54603 Villers-lès-Nancy Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399