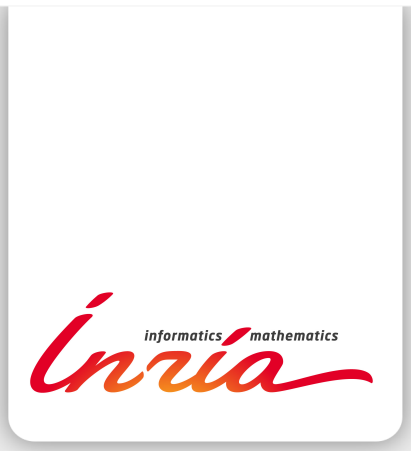
Modular C

Jens Gustedt



Modular C

Jens Gustedt

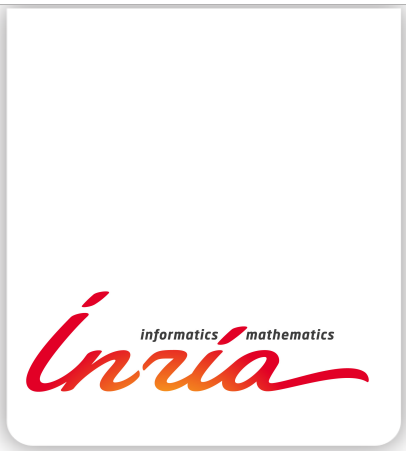
RESEARCH REPORT

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 — June 2015 — 34 pages

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 relation 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 `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.

Key-words: C, modularity, encapsulation

RESEARCH CENTRE
NANCY – GRAND EST

615 rue du Jardin Botanique
CS20101
54603 Villers-lès-Nancy Cedex

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 :

1. INTRODUCTION

Since decades, C is one of most widely used programming languages, see [BV 2015], 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 two important closely related features: modularity and reusability. 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 programmable 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 "modularity 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 modularity 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).

This paper is organized as follows. In the continuation of this introduction we will attempt to make our point about the two defaults that we try to tackle, namely lack of modularity and reusability. 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 round up 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.

1.1. Lack of modularity

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

It is intrusive. Other software projects that include C library headers just have to cope with the naming choices that have been made, there. 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. Some reserved names include an underscore, only reserve if the prefix is followed by uppercase, add a suffix such as `_t` ... Some of the reserved names are **struct**, **union** or **enum** tags, some are just plain identifiers. The rules for this are difficult to apprehend and to memorize and 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 each version so far added 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. *E.g*, no 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 **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 constant struggle with feature macros and include guards, in particular for operating system API that commonly add symbols to C standard headers.

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 enforce 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 doesn't lead to real code sharing but basically only switches off the type system to apply the same type agnostic code 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 uncontrolled 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 [Schwarz 2009]. 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 dis-

[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

ruptive 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.

1.3. C needs a specific approach

Many approaches have been proposed over the years to address these or similar odds. The may be split into two different 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 [Torvalds et al. 1996]. Such coding rules then can be made sound and effectively enforceable as has been shown by [Srivastava et al. 2008]. 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 [Srivastava et al. 2008], 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 [Gregor 2012]. Whereas that proposal tackles and solves problems with the include hierarchy it seems to add an extra layer of semantic complexity. Most importantly it doesn't 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 [Pike 2012], 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 owes much of its success to the fact that it cautiously watches to be backward and forward compatible: 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, the recent standard C11 [JTC1/SC22/WG14 2011] 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 present proposal for "Modular C" shares a lot of ideas with the one's 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 [Gregor 2012] have their own proper features and problems concerning modularization and a mixed discussion easily misses specific features that have to be addressed.

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.

1.4. 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 marcro 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 as they are necessary for usual 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 later allows to create parameterized data structures or functions easily.

Acyclic dependency:. Import of modules is not from source but uses a compiled object file. This enforces that the import relation between modules defines a directed acyclic graph. By that 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.

Exchangeability:. 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 doesn't 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 *name prefixes* to be able to interact with other, unknown, software

Listing 1. A module that exports two symbols and a type.

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

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

/* Exports proj+structs+list+head */
struct head {
  proj+structs+element* first;
  proj+structs+element* last;
};

/* The following are 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) {
  return h->first.val;
}
```

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:

```
#pragma CMOD separator +
#pragma CMOD module proj+structs+list
#pragma CMOD import proj+structs+element
```

This sets the module prefix to be composed of three identifier elements, namely `proj`, `structs` and `list`. Syntactically, these components are separated by a token that may (with some restrictions) be chosen for each module. Here the **separator** directive introduces the 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.

Observe that 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.

With the above specification, all file scope identifiers in that module are visible to the outside with the corresponding prefix. E.g, our module implements a function **init**, and to the outside that function is be visible as `proj`**+**`structs`**+**`list`**+init**. If the imported module `proj`**+**`structs`**+**`element` in turn 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* that is identifiers that no application code is allowed to use. For our purpose it is sufficient to target a particular subset of C's reserved names:

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.

RULE 1. *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 2.2. 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. *A module may freely use any identifier that is not strictly reserved as a component of a composed name.*

This rule also applies to the identifier **main** that looses its special meaning when used in a module. Instead, if a module is supposed to provide an entry point for execution, it should declare such an entry point via a directive:

```
#pragma CMOD entry unit_test
```

This declares a semantic similar to **main** for the function `unit_test`, which then must have a prototype that is equivalent to:

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

Any not strictly reserved identifier can be chosen as entry point. In particular, the traditional **main** is a valid choice.

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:

RULE 3. *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 doesn't start with the module's name. In the following we go a bit into details how this rule is assured.

C already has a convention that regulates the export of symbols, namely the *linkage* of identifiers. If not specified otherwise, *Modular C* just follows this concept:

RULE 4. *A module exports all its identifiers that have external linkage.*

Thus in the module of Listing 1 exports `proj✚structs✚list✚say_hello` and `proj✚structs✚list✚`**`init`**. Identifiers `proj✚structs✚list✚say_goodbye` or `proj✚structs✚list✚count` are not exported since they are declared **static**. In addition it uses the imported identifiers **puts** and **printf** from the module `C✚io`.
This can be seen as an extension of the "one definition rule":

RULE 5. *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.

RULE 6. *Identifiers are imported with the type of their definition.*

So far we may 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 TU, otherwise they are exported. Other identifiers (types and macros) are only exported if we say so, explicitly:

RULE 7. *Identifiers without external linkage are exported iff they are in an 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 or **inline** functions. Similar to the strategy that is applied by C++, Modular C overcomes the distinction between "tag" name space from an identifier name space.

RULE 8. *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.[4]

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

Definition 2.3. A **typedef** of an identifier `ID` is *canonical* if is of the form

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

RULE 9. *All tagged* **struct**, **union** *or* **enum** *declarations are accompanied by an appropriate canonical* **typedef**.

The function `top` that is defined at the end of Listing 1 shows another advantage of our approach. This function 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.

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**. 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 2.4. 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 2.5. 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`.

2.2. 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 5, 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*. It can be applied to every imported module and to the module name itself. It is inspired by a similar feature of the *Go* programming language. With

```
#pragma CMOD import elem=proj✚structs✚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`.

RULE 10. *Imported features can be accessed with two component names as* `abbrev✚name` *where* `abbrev` *is a short prefix that is chosen by the importer and* `name` *is the local name given by the exporting module.*

The directive

```
#pragma CMOD import printf = C✚io✚printf
```

now allows to refer to the C library IO function as **printf**. Here the saving of the `C✚io✚` prefix by itself is certainly not an important gain, but the real gain is modularity. If we want to replace C's IO module that we are using by another one, e.g, implementing the POSIX IO extensions, we just have to replace the above import directive with something like

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

Provided that the interfaces we use from an IO module are the same, no other code change is necessary.

2.3. 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 also can be freely used for a feature of the module 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:

```
1 #pragma CMOD module proj✚structs✚element
2 #pragma CMOD declaration
3 /* Exports proj✚structs✚element */
```

Listing 2. The module of Listing 1 with abbreviations.

```
1  #pragma CMOD module head   = proj+structs+list
2  #pragma CMOD import elem   = proj+structs+element
3  #pragma CMOD import io     = C+io
4  #pragma CMOD import printf = C+io+printf
5  #pragma CMOD import puts   = C+io+puts
6
7  /* The following declarations are exported */
8  #pragma CMOD declaration
9
10 /* Exports proj+structs+list */
11 struct head {
12   elem* first;
13   elem* last;
14 };
15
16 /* From here on, only exported if
17    external linkage. */
18 #pragma CMOD definition
19
20 void say_hello(void){
21   puts("Hello world");
22 }
23 static unsigned count;
24 static void say_goodbye(void){
25   printf("on exit we see %u", count);
26 }
27 head* init(head* h) {
28   if (h) *h = (head){ 0 };
29   return h;
30 }
31 /* Exports proj+structs+list+top,
32    no conflict with ctype.h */
33 double top(head* h) {
34   return h->first.val;
35 }
```

```
4  struct proj+structs+element {
5    double data;
6    proj+structs+element* next;
7  };
```

With an abbreviation `elem` the code simplifies:

```
1  #pragma CMOD module elem=proj+structs+element
2  #pragma CMOD declaration
3  /* Exports proj+structs+element */
4  struct elem {
5    double data;
6    elem* next;
7  };
```

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`.

RULE 11. *The composite name of a module M can be referred to by the abbreviation given by an* **import** *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:

```
1  #pragma CMOD module another_project✚doit
2  #pragma CMOD import list = proj✚structs✚list
3
4  list* pop_or_push(list* l, double a) {
5    if (l && l->data == a) {
6      l = list✚pop(l);
7    } else {
8      l = list✚insert(l, a);
9    }
10   return l;
11 }
```

The use of the module name is not limited to types. E.g, a function can be in the center of interest of a module: we may introduce a module `C✚lib✚rand` that interfaces the C library function **rand** as `C✚lib✚rand` and **srand** as `C✚lib✚rand✚set`:

```
1  #pragma CMOD module      rand = C✚lib✚rand
2  #pragma CMOD declaration
3  int rand(void);
4  int rand✚set(unsigned int seed);
5  #define MAX C✚lib✚RAND_MAX
```

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

```
1  #pragma CMOD module      func = proj✚func
2  double func(double x){ ... }
3  double func✚deriv(double x){ ... }
4  double func✚integ(double low, double high){ }
```

2.4. Mangling

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

Definition 2.6. 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 doesn't 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 2.7. 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 2.8. 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.

RULE 12. *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 relationship that is depicted in Figure 1.

RULE 13. *The import relation 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.

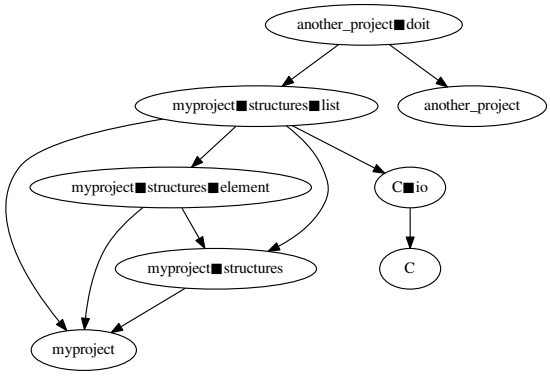


Fig. 1. The import dependency graph of the example module

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 relation 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 [Srivastava et al. 2008] for modularity and information hiding: all our explicit or implicit interface inclusion ensures that we only have what they call *horizontal dependency* and that there is no *vertical dependency*.

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. C++ has multiple problems here: include files tend to be monolithic, they include an exponential number of other files themselves, and the language is difficult to parse: some declarations may incur an exponential computational overhead at compile time.

This situation is not at all comparable for C. C compilers nowadays are able to parse C headers extremely fast. Their bottleneck usually is code generation and optimization, and so the problem that had been observed for C++ doesn't apply to C. So for Modular C we don't need to be restrictive.

3.1. Import is binary

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

RULE 14. *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 project might just provide the compiled objects of its modules to its clients. This then enables them to just execute executables that use that module, e.g, via dynamic linking, but also to compile their own executables that would use such a module. Any binary distribution of modules is, with our model, also a development distribution.

As a direct consequence we have the following rule:

RULE 15. *The import dependency graph for any module must be acyclic.*

Listing 3. A module with an initialization function.

```
1  #pragma CMOD module          proj+something
2  #pragma CMOD import  thrd = C+thrd;
3  #pragma CMOD import  mtx  = C+thrd+mtx;
4
5  #pragma CMOD init    module_init
6
7  #ifndef thrd+NO_THREADS
8  mtx bkl;
9  void module_init(void) {
10   mtx+init(&bkl, mtx+recursive);
11 }
12 #else
13 void module_init(void) {
14   // empty
15 }
16 #endif
```

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.

RULE 16. *There is a topological sort of the dependency graph.*

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

RULE 17. *All exported symbols of all modules of the import dependency 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,

— 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 file name in which such a module might be implemented. Such a detail is left to the implementation.

3.2. Module initialization and cleanup

The strict dependency relation between modules also let us provide secondary features, module initialization and cleanup. Such features may be helpful for file scope objects that need to be initialized dynamically; C doesn't allow to call functions to initialize statically allocated objects. This is achieve by assigning the initialization role through a **init** directive:

```
#pragma CMOD init module_init
void module_init(void);
```

If such a directive is given, the function is guaranteed to be executed *before* any symbol (variable or function) of the module is accessed, and *after* all such **init** functions are

executed for all imported modules. Such a function must have a prototype as indicated in the example.

RULE 18. *The functions listed in* **init** *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.[5]

In contrast to the approach of the C standard with the type **once_flag**, this approach here has the advantage that it has only an overhead at startup of the application, where all the **init** functions are called. Thereafter, during the execution of the program itself there is no overhead at all and an application can be guaranteed to start all its operations in a well defined state.

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 as C library functions 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.

RULE 19. *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* **init** *functions.*

4. CODE SHARING

In addition to the interface sharing between modules that we described so far, we provide a second mechanism of direct code sharing through *code snippets*. This will be different from making a function body of a particular function of the context *exporter* visible to the importer. As we already have mentioned the existing approach of C with **inline** does that suitably well. We ease the use of that mechanism by emitting the corresponding function symbol in the TU that contains it.

4.1. Snippets

We propose a formalism to share code that will give rise to *different* functions (macros, types, constants ...) in the context of the *importer*. Let us start with a simple module `C+snippet+init` whose only purpose is to share code for a uniform **init** funtion with the importer.

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

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

```
1  #pragma CMOD module toto = proj+structs+toto
2  /* Import three snippets. */
3  #pragma CMOD import        C+snippet+init
4  #pragma CMOD import        C+snippet+alloc
5  /* A snippet with 3 slots that are filled. */
6  #pragma CMOD import vec  = C+tmpl+vector
7  #pragma CMOD fill vec+vType = vec23
8  #pragma CMOD fill vec+T     = toto
9  #pragma CMOD fill vec+size  = vsize
10
11 #pragma CMOD declaration
12 struct toto {
13   /* your favorite data */
14 };
15 #define vsize 23
16
17 #pragma CMOD definition
18
19 vec23 myState = { 4711, };
```

```
1  #pragma CMOD module init = C+snippet+init
2
3  /* All code hereafter is integrated as-is
4     to the importer. */
5  #pragma CMOD snippet  T = complete
6  T* init(T* x) {
7    if (x) *x = (T){ 0 };
8    return x;
9  }
```

The **snippet** directive introduces a local name, here `T`, that stands for the name of the importing module. The indication **complete** on the right requires that this module name corresponds to a complete type. 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*.

RULE 20. *There is at most one* **snippet** *directive in the source of a module.*

RULE 21. *The source code of snippets is integrated in declaration order at the end of the importing module.*

The later implies that a snippet is handled twice during compilation for Modular C. First, it is seen during the compilation of the module hat exports it. It is then set aside in the produced binary for a later integration to importers. Then, on import the snippet is integrated source to source into the importer. The native C compilers can then detect errors or inconsistencies during this source to source integration.

RULE 22. *The identifier (if any) that is specified in the* **snippet** *directive serves as an abbreviation for the name of the importing module.*

Listing 4 shows an example for imports of snippets. Here the modules `C+snippet+init` and `C+snippet+alloc` (see below) provide functions **init** and `alloc`. These are

visible as members of this module, that is as `proj+structs+toto+init` and `proj+structs+toto+alloc`, respectively. The module `C+tmpl+vector` that we will see in Section 4.2 provides a specialized type `vec23`.

As the identifier `T` in both our examples stands for the name of the importer, the functions have the prototypes:

```
proj+structs+toto* init(proj+structs+toto*);
proj+structs+toto* alloc(void);
```

The module `C+snippet+alloc` can be implemented in a similar way as `C+snippet+init`. Here we add one additional feature: it uses a function that operates on type `T`, namely **init**.

```
1  #pragma CMOD module alloc = C+snippet+alloc
2  #pragma CMOD import lib   = C+lib
3
4  /* All code hereafter is integrated
5     as-is to the importer.        */
6  #pragma CMOD snippet  T = complete
7
8  /* This snippet enforces that the importer
9     provides a function init.          */
10 #pragma CMOD declaration
11 extern T* init(T*);
12
13 #pragma CMOD definition
14 T* alloc(void) {
15   return init(lib+malloc(sizeof(T)));
16 }
```

Note that the module `C+snippet+alloc` doesn't know much about the context in which the function `alloc` will be placed by the import mechanism. It formulates an interface contract 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. Note that this second part of the interface contract doesn't need special syntax. A standard C declaration of a function prototype is sufficient for its specification.

RULE 23. *The exported symbols of a snippet contribute to the interface of the importer.*

4.2. Slots

The snippet mechanism above can only serve for situations were the exported code naturally concerns the elaboration of one feature, namely the one specified by the importer. It is insufficient if the module name is not the principal object of the snippet code or if the snippet code has to be parameterized with several entities.

```
1  #pragma CMOD module C+tmpl+vector
2
3  /* All code hereafter is integrated as-is
4     to the importer. */
5  #pragma CMOD snippet  none
6  /* The three slots that are defined here
7     parameterize the code as it is seen by
8     the importer.                          */
9  #pragma CMOD slot vType  = none
```

```
10 #pragma CMOD slot T      = complete
11 #pragma CMOD slot size   = none
12
13 #pragma CMOD declaration
14 typedef T vType[size];
```

Module `C+tmpl+vector` is parametrized with three identifiers, its *slots*. Through these it provides its importer with a new type, `vType` that is parameterized by another type `T` and a number `size`. For the integration of such a snippet, these three identifiers must be replaced ("filled") by three identifiers that the importer choses freely. Listing 4 shows how the three slots in the specification of `C+tmpl+vector` are filled:

`C+tmpl+vector`	`proj+structs+toto`
`vType`	`vec23`
`T`	`toto`
`size`	`vsize` with 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+T`	`proj+structs+toto`
`C+tmpl+vector+size`	`proj+structs+toto+vsize`

RULE 24. *All slots of all directly imported modules must be filled.*

In many points, this approach with snippets resembles the traditional *X macro* technique that we described above. It differs from that by our definition-is-declaration rule, there is no need to produce consistent generation of a header declarations and TU definitions. It also doesn't randomly intrude the user code with syntactically odd directives. The **fill** (and **slot**) directives can and should appear in the preliminaries of the user's source.

Generally, a snippet `S` can not be imported several times to the same module `M`, because this would lead to multiple definitions in `M`. E.g the repetition of the snippet of `C+snippet+alloc` would lead to a second definition of the function `alloc`, which is an error.

The module `C+tmpl+vector` avoids that. When using different abbreviations it could be imported several times.

Definition 4.1. A snippet is called a *template* if all the features that it specifies for the importer are named through **slot** directives.

Or said otherwise, it is a template if it doesn't pollute the name space of the importer.

RULE 25. *Templates can be imported several times into the same module* `M` *if each import uses a different abbreviation.*

4.3. Reach of snippets

Even though we have transitivity of the import relation, 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 26. *Any identifier* `N` *in a module* `B` *who's definition originates from a snippet of an* **import** *directive for module* `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. This could even lead to a name clash if other import path would have an **init** function in one of the snippets. Last but not least, we certainly can't expect any importer of `proj✚structs✚toto` to fill the slots of `C✚tmpl✚vector` with consistent declarations.

RULE 27. *A snippet section of a module* `A` *is only integrated by an explicit* **import** *directive for* `A`.

5. A FORMAL DESCRIPTION

One of the advantages of our approach is that much of it can be described by text rewriting, that is simple enough to allow for a formal proof of the validity of our approach.

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^+ relies on another feature that is not exported.
(5) After preprocessing (C phase 4), 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, $\mathcal{P}(P)$, the *preprocessor extract*, consist 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. If they are external, their definition has to appear in the declarative part (as in a traditional header

file) and a **extern inline** has to appear in the definition part to enforce their instantiation. Internal **inline** functions have just to appear once in the declaration extract with a **static inline** definition.

Definition 5.4. For a conforming C source P, $\mathcal{D}(P)$, the *declaration extract*, consist of a preprocessed copy of P where all object definitions and all definitions of functions that are not **inline** are replaced by declarations that use appropriate **extern** or **static** linkage specification, and all **extern inline** declarations have been removed. In addition $\mathcal{D}(P)$ starts with canonical **typedef** for all tagged **struct** and **union** types, and all tagged **enum** types are immediately followed by canonical **typedef**.

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, $\mathcal{I}(P)$, the *definition extract*, consist of a copy of P where

— All macro definitions and includes have been removed.
— All file scope **typedef**, **struct**, **union** or **enum** definitions are removed.
— All definitions of **static inline** functions have been removed.
— All remaining definitions of **inline** functions have been replaced by appropriate **extern inline** declarations that are not definitions.

Definition 5.6. A conforming C source P is called *stable* if all the following hold:

(1) P and the concatenation $\mathcal{P}(P)\cdot\mathcal{D}(P)\cdot\mathcal{I}(P)$ are functionally equivalent.
(2) It doesn't contain **#include** directives.
(3) It doesn't contain explicit references to C library functions or objects.
(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 doesn't use preprocessor operators **#** and **##**.
(6) It doesn't use the reserved identifiers **__func__**, **__LINE__** or **__FILE__**.

Claiming stability for a C source may sound 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.

For the requirement in (1) that macros can be placed first, note first that because of (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 identifier 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 to be seen as a macro.

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 both **struct** definitions in $\mathcal{P}(P)\cdot\mathcal{D}(P)\cdot\mathcal{I}(P)$ is 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 later 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 $\mathcal{H}(P)$ 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 5. 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 they are defined) inserts `module_atexit` and `module_at_quick_exit` in the appropriate queues and calls the modules's own `module_init` function. Here, as an example, this uses the C standard data type **once_flag** to guarantee race freeness of the overall initialization. Implementa-

(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, $i = 0, \dots, n-1$ be the snippets for M, listed as their appearance in **import** directives. For all $i = 0, \dots, n-1$:
      — Replace slot names in S_i by their **fill** definitions.
      — Replace `_Importer` by `MODULENAME`.
      — Append the snippet to the regular part.
   (b) Append predefined internal function and object definitions to M^+.
   (c) In M^+, prefix all remaining file scope identifiers by `MODULENAME`.
   (d) Mangle all composed names to C source $P = f(M^+)$.
   (e) Extract $\mathcal{P}(P)$, $\mathcal{D}(P)$ and $\mathcal{I}(P)$ and store $\mathcal{P}(P) \cdot \mathcal{D}(P)$ as file `"MODNAME.h"`
   (f) In top-sort order, let $\mathcal{H}_0, \dots, \mathcal{H}_{n-1}$ be the headers of imported modules. Compile the concatenation $\mathcal{H}_0 \cdots \mathcal{H}_{n-1} \cdot \mathcal{P}(P) \cdot \mathcal{D}(P) \cdot \mathcal{I}(P)$ 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.

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

tions 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 addtion, 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`.

5.3. Formal proofs

In a first step, we have to show when mangling doesn't 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 5.7. *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.*

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 doesn't apply if there are no *Modular C* directives.

LEMMA 5.8. *Let P be a conforming C source without composed identifiers and* Modular C *directives. Then P is stable iff the replacement procedure compiles P to a valid archive file.*

Listing 5. 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_init();
}
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);
}
```

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 5.9. *Let P be as above and stable, and such that it has an* **entry** *function but no* **init** *or* **exit** *functions, and such that it makes no reference to any external object or function. Then the archive file can be linked to a valid executable that is functionally equivalent to the original program defined by 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:

LEMMA 5.10. *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 contains an* **entry** *function, the archive file can be linked to a valid executable that is functionally equivalent to the original program defined by M.*

5.4. Stiching 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 5.11. An ordered set of valid modules $\mathcal{M} = (M_0, \dots, M_{n-1})$ forms a *consistent project* if the following hold:

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

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

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

THEOREM 5.12. *Let $\mathcal{M} = (M_0, \dots, M_{n-1})$ be a consistent project with completed sources $\mathcal{M}^+ = M_0^+, \dots, M_{n-1}^+$. 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 $\mathcal{P}(f(M_j^+)) \cdot \mathcal{D}(f(M_j^+))$ for some $j < i$.*
(3) *For all $i < n$, the following concatenation of sources is stable:*

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

(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 5.13. *Let $\mathcal{M} = (M_0, \dots, M_{n-1})$ be a consistent project without snippets, and $\mathcal{H}_0, \dots, \mathcal{H}_{n-1}$ 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 $\mathcal{H}_0 \cdots \mathcal{H}_{i-1}$.*
(3) *For all $i < n$, the concatenation of sources $\mathcal{H}_0 \cdots \mathcal{H}_{i-1} \cdot f(M_i)$ 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 5.10.

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 prove 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 5.12. 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 an 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 appended.

□

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

THEOREM 5.14. *If during initialization none of the* `module_init` *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* gives the opportunity to give a structured view to the 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 this in three levels:

(1) Modules prefixed with `C✚std` provide the symbols in snippets: e.g, `C✚std✚lib` for `stdlib.h`. That is, they remain mostly 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 later 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 |zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
```

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 such 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 ... */                           \
 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 will clearly indicate 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 unicity 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 will be 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` will provide 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 can 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 recent 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 implementors 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`.

E.g there are `C✚`**snippet**`✚`**init**, `C✚`**snippet**`✚alloc` and `C✚`**snippet**`✚minmax` that were already mentioned. 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 month 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.[6]

We only had to use one less common tool, `makeheaders` [Hipp 1993]. It has been crucial to implement Step 7e, declaration extraction, of the replacement procedure. `makeheaders` is relatively small but very efficient opensource C program that implements a partial C parser and extracts just the declarative parts from a C program. It also sorts the declarations in dependency order and adds the **typedef** for tag names. We only had to adapt it a little to comply to modern C: it dates from 1993 and didn't yet integrate additions of C99 and C11 in the parser. It has been of great help, and is by itself a proof for the longevity of C programs.

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

[6] Access to the sources is available at `cmod.gforge.inria.fr`

Listing 6. An extract of the implementation of a C library interface.

```
1  #pragma CMOD module dbl     = C+real+double
2  #pragma CMOD import minmax = C+snippet+minmax
3
4  #pragma CMOD mimic <math.h>
5  #pragma CMOD mimic <float.h>
6  #pragma CMOD link -lm
7  ...
8  #pragma CMOD define NAN
9  #pragma CMOD typedef eval=double_t
10 #pragma CMOD defexp PI=4.0*atan(1.0)
11 #pragma CMOD alias acos
12 ...
13 #pragma CMOD declaration
14 typedef double dbl;
15 double (acos)(double);
```

versions of two different compilers, `gcc`[7] and `clang`[8], and two different C libraries, `glibc`[9] and `musl`[10].

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 6.

The idea is to capture the dependency from a traditional header file within in a stub module. It uses a **mimic** directive to refer to the traditional header file, adds information for the linker with **link**, and then extracts all information from the header by either text matching or by compiling short test programs. Here we provide an additional set of directives:

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

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, the platform 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

[7] http://gcc.gnu.org
[8] http://clang.llvm.org
[9] http://www.gnu.org/software/libc/
[10] http://musl-libc.org

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**.

So 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.

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.

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. By following this example, it should be possible for an experienced programmer to interface any reasonable C application header within an hour of work.

Once that interfacing is achieved, 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 type (or more general for each functional unit) that we want to export. We start with those units that are at the core of our project and don't need to import other parts of the project.

```
1  #pragma CMOD module list = proj✚list
2  /* Add analogous imports for all standard headers. */
3  #pragma CMOD import        C✚std✚io
4
5  #pragma CMOD declaration
6
7  /* Put all type declarations and macros from "proj_list.h", here. */
8  struct proj_list {
9    proj_list* next;
10   /* something */
11 };
12
13 #pragma CMOD definition
14
15 /* Copy most of "proj_list.c", here.
16    Omit extern decl for inline functions. */
17 proj_list* proj_list_init(proj_list* l) {
```

```
18    /* do something */
19    return l;
20  }
```

Compile the resulting modules `proj.X` and `proj#list.X`. Once the compilation is successful we should find 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

```
0   > nm -C proj#list.a
1   proj#list.o:
2   0000...0000 W proj_list_proj_list_init
3   0000...0000 T _C::proj::list::proj_list_init
```

We see that we have in fact two entries in the symbol table, one for the mangled name (here retranslated into C++'s composite naming scheme) and an alias for the conventional name, here `proj_list_proj_list_init`.

Now let us get rid of the internal name prefix by first removing all `proj_list_` prefixes and then substituting `proj_list` by `list` for the type itself.

```
1  #pragma CMOD module list = proj+list
2  /* Add analogous imports for all standard
3     headers. */
4  #pragma CMOD import        C+std+io
5
6  #pragma CMOD declaration
7
8  /* Put all type declarations and macros from "proj_list.h", here. */
9  struct list {
10   list* next;
11   /* something */
12 };
13
14 #pragma CMOD definition
15
16 /* Copy most of "proj_list.c", here.
17    Omit extern decl. for inline functions. */
18 list* init(list* l) {
19   /* do something */
20   return l;
21 }
```

By that the first the `nm` output should have changed to

Terminal

```
0   proj#list.o:
1   0000...0000 W proj_list_init
2   0000...0000 T _C::proj::list::init
```

So we now have an object file that we can use instead of the one that was 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 compatible with the header files that we had for that, so our new version of the library written with modules can be used without problems.

Evidently, all these replacement so far only improved the ease of reading and writing our code, but not code reuse. But once this status is reached, the real refactoring of the code can begin: module imports from the `C+std+` hierarchy can be replace, macros and X macros can be rewritten as snippets, code parts that are structurally similar can be grouped together, repetitive code can be substituted by a single definition in a snippet.

8. CONCLUSION

Our proposition *Modular C* introduces modularity and reusability by naming schemes, only. 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*. Thereby 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. Thereby related features of a software can be grouped and accessed together, and even replaced seamlessly. This is done without introducing new syntactical constructs to the core language, and allows for more flexible grouping of features than classes in traditional object-oriented languages.

Another important missing feature was provided, *snippets*. These allow for easy-to-apply parametrized code reuse, similar to X macros or C++'s templates. Their advantage is that their code itself also is written in conventional C, no extra syntax is required.

Other features were added 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 initialization 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

Tiobe Software BV. 2015. (2015). http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html monthly since 2000.

Doug Gregor. 2012. Modules. Apple Inc.. (Dec 2012). http://llvm.org/devmtg/2012-11/Gregor-Modules.pdf

D. Richard Hipp. 1993. Makeheaders. (1993). http://www.hwaci.com/sw/mkhdr/

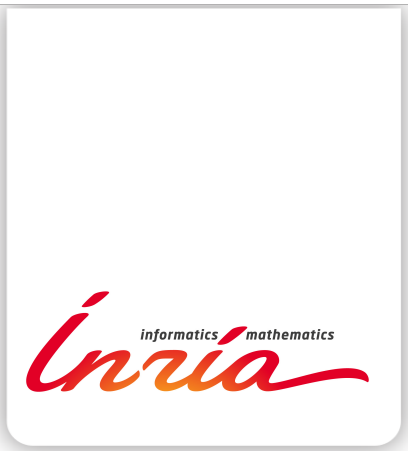
JTC1/SC22/WG14 (Ed.). 2011. *Programming languages - C* (cor. 1:2012 ed.). Number ISO/IEC 9899. ISO. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf

Rob Pike. 2012. Go at Google. In *Conference on Systems, Programming, and Applications: Software for Humanity, SPLASH '12*. 5–6. DOI:http://dx.doi.org/10.1145/2384716.2384720 see also http://talks.golang.org/2012/splash.article.

Keith Schwarz. 2009. Advanced Preprocessor Techniques. (2009). http://www.keithschwarz.com/cs106l/spring2009/handouts/080_Preprocessor_2.pdf

Saurabh Srivastava, Michael Hicks, Jeffrey. S Foster, and Patrick Jenkins. 2008. Modular Information Hiding and Type-Safe Linking for C. *IEEE Transactions on Software Engineering* 34, 3 (2008), 357–376.

Linus Torvalds and others. 1996. Linux kernel coding style. (1996). https://www.kernel.org/doc/Documentation/CodingStyle evolved mildly over the years.



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