Modular Information Hiding and Type-Safe Linking for C

Saurabh Srivastava, Michael Hicks, Jeffrey S. Foster, and Patrick Jenkins

All authors are with the University of Maryland, College Park.

Abstract

This paper presents CMOD, a novel tool that provides a sound module system for C. CMOD works by enforcing a set of four rules that are based on principles of modular reasoning and on current programming practice. CMOD's rules flesh out the convention that `.h` header files are module interfaces and `.c` source files are module implementations. Although this convention is well-known, existing explanations of it are incomplete, omitting important subtleties needed for soundness. In contrast, we have proven formally that CMOD's rules enforce both information hiding and type-safe linking.

To use CMOD, the programmer develops and builds their software as usual, redirecting the compiler and linker to CMOD's wrappers. We evaluated CMOD by applying it to 30 open source programs, totaling more than one million lines of code. Violations to CMOD's rules revealed more than a thousand information hiding errors, dozens of typing errors, and hundreds of cases that, although not currently bugs, make programming mistakes more likely as the code evolves. At the same time, programs generally adhere to the assumptions underlying CMOD's rules, and so we could fix rule violations with a modest effort. We conclude that CMOD can effectively support modular programming in C: it soundly enforces type-safe linking and information hiding while being largely compatible with existing practice.

Index Terms

Coding tools and techniques, C, modules/packages, information hiding, type-safe linking, CMOD, software reliability.

I. Introduction

Module systems allow large programs to be constructed from smaller, potentially reusable components. The hallmark of a good module system is support for *information hiding*, which allows components to conceal internal structure, while ensuring that component linking is *type safe*. This combination allows modules to be safely written and understood in isolation, enhancing the reliability of software [32].

While full-featured module systems are part of many modern languages (such as ML, Haskell, Ada, and Modula-3), the C programming language—still the most common language for operating systems, network servers, and other critical infrastructure—lacks direct support for modules. Instead, programmers typically think of `.c` source files as module implementations and use `.h` header files (containing type and data declarations) as module interfaces. Textually including a `.h` file via the `#include` directive is akin to "importing" a module.

Many experts recommend using this basic pattern [2], [16], [17], [18], [20], but to our knowledge, existing presentations of the basic pattern are too weak to ensure proper information hiding and type safe linking. As a result, programmers may be unaware of (or ignore) the pitfalls of using the pattern incorrectly, and thus may make mistakes (or cut corners) since the compiler and linker provide no enforcement. The result is the potential for link-time type errors and information hiding violations, which degrade programs' modular structure, complicate maintenance, and lead to defects.

As a remedy to these problems, this paper presents CMOD, a novel tool that enforces a sound module system for C based on existing practice. CMOD works by enforcing four programming rules that flesh out C's basic modularity pattern. We have proven formally that, put together, CMOD's rules ensure C programs obey information hiding policies implied by interfaces, and that linking modules together is type safe, i.e., that the types of shared symbols match across module boundaries.[1] To our knowledge, CMOD is the first system to enforce both properties for standard C programs. Related approaches (Section VI) either require linguistic extensions (e.g., Knit [28] and Koala [33]) or enforce type safe linking but not information hiding (e.g., CIL [24] and C++ "name mangling").

To evaluate how well CMOD's rules match existing practice while still strengthening modular reasoning, we ran CMOD on a suite of programs cumulatively totaling more than one million lines of code, split across 1478 source and 1488 header files. Rule violations revealed more than a thousand information hiding errors, dozens of typing errors, and hundreds of cases that, although not currently bugs, make programming mistakes more likely as the code evolves. Nevertheless, most programs follow the basic modularity pattern, and we found that making them compliant with CMOD's rules requires only a modest effort. CMOD is designed to be easy to use, requiring only that the programmer redirect the compiler and linker commands in their makefile to CMOD's wrappers. We found that building with our prototype implementation of CMOD takes roughly 4.1 times as long as the regular build process on average, with a median slowdown of 3.1, and we believe the overhead could be reduced to a minimal level with more engineering effort. These results suggest that CMOD can be integrated into current software development practice

[1]C's weak type system still allows programmers to violate type safety in other ways, e.g., by using unchecked casts. Other work, notably CCured [23] and Deputy [6], can be used to strengthen C's type system to eliminate these problems. CMOD complements these efforts, and vice versa.

at relatively low cost while enhancing software safety and maintainability.

In summary, the contributions of this paper are as follows:

- We present a set of four rules that ensure it is sound to treat header files as interfaces and source files as implementations (Section II). To our knowledge, no other work fully documents a set of programming practices that are sufficient for modular safety in C. While this work focuses on C, our rules should also apply to languages that make use of the same modularity convention, such as C++, Objective C, and Cyclone [14].
- We give a precise, formal specification of our rules and prove that they are sound, meaning programs that obey the rules follow the information hiding policies defined by interfaces and are type safe at link time (Section III).
- We present our implementation, CMOD (Section IV), and describe the results of applying it to a set of benchmarks (Section V). CMOD found over a thousand information hiding violations and dozens of typing errors, among other brittle coding practices. We found that bringing code into compliance with CMOD was generally straightforward.

An earlier version of this work was published in a workshop proceedings [31]. The current version improves on the prior work in several ways: The formalism now explicitly handles duplicate inclusion, which was assumed absent before—this seemingly small change required an almost complete revamping of the soundness proof; we found and addressed a subtle bug in our previous system due to recursive inclusion; CMOD now supports `.c` files that are `#included` but do not act as interfaces; we have added more discussion of our implementation; and we have expanded the experiments to include more and larger programs, nearly tripling the total lines of code considered.

II. Motivation and Informal Development

We begin our discussion by presenting C's modularity convention and informally introducing CMOD's rules for modular programming. Abstractly, we define a *module implementation* M to be a set of term and type definitions, and we define a *module interface* I to be a set of term and type declarations. Interfaces are used to declare the *exported* terms and types of a module—when one module wishes to refer to the definitions of another module M, it must do so through M's interface. Moreover, in most module systems, the compiler ensures that each module *implements* its interface, meaning that it exports any types and terms in the interface (and may define additional, private terms and types as well). These features ensure *separate compilation* when module implementations are synonymous with compilation units.

There are two key properties that make such a module system safe and effective. First, clients must depend only on interfaces rather than on particular implementations:

Property 2.1 (Information Hiding): Suppose that M implements interface I. Then if M defines a symbol g, other modules may only access g if it appears in I. If I declares an abstract type t, no module other than M may use values of type t concretely.

This property makes modules easier to reason about and reuse. In particular, if a client successfully compiles against interface I, it can link against *any* module that implements I. Consequently, M may safely be changed as long as it still implements I.

The second key property of a module system is that linking must be type-safe:

Property 2.2 (Type-Safe Linking): If module N refers to symbols in some interface I and M implements I, and M and N are individually type-safe, then the result of linking M and N together is type-safe.

The goal of CMOD is to define a backward-compatible module system for C that enjoys these two properties. The remainder of this section describes our approach.

A. Basic Modules in C

Our starting place is the well-known C convention in which `.c` *source files* act as separately-compiled implementations, and `.h` *header files* act as interfaces [2], [16], [17], [18], [20]. Fig. 1 shows a simple C program that follows this convention. In this code, header `bitmap.h` acts as the interface to `bitmap.c`, whose functions are called by `main.c`. The header contains an abstract declaration of type `struct BM` and declarations of the functions `init` and `set`. To use `bitmap.h` as an interface, the file `main.c` "imports" it with the directive `#include "bitmap.h"`, which the preprocessor textually replaces with the contents of `bitmap.h`. At the same time, `bitmap.c` also invokes `#include "bitmap.h"` to ensure its definitions match the header file's declarations.

This program properly hides information and links type-safely. Since both `main.c` and `bitmap.c` include `bitmap.h`, the C compiler ensures that the types of `init` and `set` match across the files. Furthermore, `main.c` never refers to `bitmap.c`'s symbol `private` and does not assume a definition for `struct BM` (thus treating it abstractly), since neither appears in `bitmap.h`.

bitmap.h

```
1  struct BM;
2  void init (struct BM **);
3  void set(struct BM *, int);
```

bitmap.c

```
4  #include "bitmap.h"
5
6  struct BM { int data; };
7  void init (struct BM **map) { ... }
8  void set(struct BM *map, int bit) { ... }
9  void private(void) { ... }
```

main.c

```
10 #include "bitmap.h"
11
12 int main(void) {
13   struct BM *bitmap;
14    init (&bitmap);
15   set(bitmap, 1);
16    ...
17 }
```

Fig. 1. Basic C Modules

B. Header Files as Interfaces

One of the key principles illustrated in Fig. 1 is that symbols are always shared via interfaces. In the figure, header `bitmap.h` acts as the interface to `bitmap.c`. Clients `#include` the header to refer to `bitmap.c`'s symbols, and `bitmap.c` includes its own header to make sure the types match in both places [18], [20]. CMOD ensures that linking in this way is mediated by an interface with the following rule:

Rule 1 (Shared Headers): Whenever one file links to a symbol defined by another file, both files must include a header that contains the declaration of that symbol.

The C compiler and linker do not enforce this rule, so programmers sometimes fail to use it in practice. Fig. 2 illustrates some of the common ways the rule is violated, based on our experience (Section V). One common violation is for a source file to fail to include its own header, which can lead to type errors. In Fig. 2, `bitmap.c` does not include `bitmap.h`, and so the compiler does not discover that the defined type of `init` (line 9) is different than the type declared in the header (line 2).

Another common violation is to import symbols directly in `.c` files by using `extern`, rather than by including a header. In the figure, line 15 declares that `private` is an external symbol,

bitmap.h

```
1  struct BM;
2  void init (struct BM **);
3  void set(struct BM *, int);
```

bitmap.c

```
4  /* bitmap.h not included */
5
6  struct BM { int data; };
7
8  /* inconsistent declaration */
9  void init (struct BM *map) { ... }
10 void set(struct BM *map, int bit) { ... }
11 void private(void) { ... }
```

main.c

```
12 #include "bitmap.h"
13
14 /* bad symbol import */
15 extern void private(void);
16
17 /* violating type abstr. */
18 struct BM { int *data; };
19
20 int main(void) {
21   struct BM *bitmap;
22    init (&bitmap);
23   set(bitmap,1);
24   private ();
25   bitmap−>data = ...;
26    ...
27 }
```

Fig. 2. Violations of Rules 1 and 2

allowing it to be called on line 24 even though it is not mentioned in `bitmap.h`. This violates information hiding, preventing the author of `bitmap.c` from safely changing the type of, removing, or renaming this function. It may also violate type-safe linking, e.g., when a local `extern` declaration assigns the wrong type to a symbol. We have seen both problems in our experiments. One way that the author of `bitmap.c` could prevent such problems would be to declare `private` as `static`, making it unavailable for linking. However, programmers often fail to do so. In some cases this is an oversight—for the benchmarks we used in our experiments, we found that on average 17% of a project's symbols could be declared `static`, and as many as 79% in the most extreme example. But in other cases, a symbol cannot be declared `static` because it should be available for linking to some, but not all, files.

Rule 1 admits several useful coding practices. One common practice is to use a single header as an interface for several source files (as opposed to one header per source file, as in Fig. 1). For example, the standard library header `stdio.h` often covers several source files, and to adhere to Rule 1, each source file would `#include "stdio.h"`. Another common practice is to have several headers for a single source file, to provide "public" and "private" views of the module [20]. In this case the source file would include both headers, while clients would include one or the other.

The last error in Fig. 2 is in `main.c`, which violates the information hiding policy of `bitmap.h` by defining `struct BM` on line 18. In this case the violation also results in a type error since the definitions on lines 6 and 18 do not match. Rule 1 does not prevent this problem because it refers to symbols and not types. Our solution is to treat type definitions in a manner similar to how the linker treats symbols. The linker requires in general that only one file define a particular function or global variable name. This ensures there is no ambiguity about the definition of a given symbol during linking. Likewise for types, we can require that there is only one definition of a type that all modules "link against," in the following sense.

We say that a type definition is *owned* by the file in which it appears. If the type definition occurs in a header file (and hence is owned by the header), then the type is *transparent*, and many modules may know its definition. In this case, "linking" occurs by including the header. Alternately, if the type definition appears in a source file (and hence is owned by that file), then the type is *abstract*: only that module, which implements the type's functions, should know its definition. CMOD requires that a defined type have only one owner, eliminating the problem in Fig. 2:

Rule 2 (Type Ownership): Each type definition in the linked program must be owned by exactly one source or header file.

Notice that this rule is again somewhat flexible, allowing a middle-ground between abstract and transparent types. In particular, the rule allows a "private" header to reveal a type's definition while a "public" header keeps it abstract. Files that implement the type and its functions include both headers, and those that use it abstractly include only the public one.

This notion of ownership makes sense for a global namespace in which type and variable names have a single meaning throughout a program. For variables, the `static` qualifier offers some namespace control, but C provides no corresponding notion for type names. While we could imagine supporting a `static` notion for types, we use our stronger rule because it is simple to implement, and we have found programmers generally follow this practice.

bitmap.h

```
1  struct BM;
2  #ifdef COMPACT
3   void init (struct BM *);
4  #else
5   void init (struct BM *, int size);
6  #endif
7  void set(struct BM *, int);
```

bitmap.c

```
8  #include "config.h"
9  #include "bitmap.h"
10
11 #ifdef COMPACT
12   struct BM { int map; }
13   void init (struct BM *map) { ... }
14   void set(struct BM *map, int bit) { ... }
15 #else
16   struct BM { int size; int *map; }
17   void init (struct BM *map, int size) { ... }
18   void set(struct BM *map, int bit) { ... }
19 #endif
```

config.h

```
20 #ifndef _CONFIG_H
21 #define _CONFIG_H
22 #ifdef __BSD__
23  #undef COMPACT
24 #else
25  #define COMPACT
26 #endif
27 #endif
```

main.c

```
28 #include "config.h"
29 #include "bitmap.h"
30
31 int main(void) {
32   struct BM *bmap;
33 #ifdef COMPACT
34    init (bmap);
35 #else
36    init (bmap, 7);
37 #endif
38   set(bmap, 1);
39    ...
40 }
```

Fig. 3. Using the Preprocessor for Configuration

C. Preprocessing and Header Files

Rules 1 and 2 are the core of CMOD's enforcement of information hiding and type-safe linking. However, for these rules to work properly, we must account for the actions of the preprocessor.

Consider the code shown in Fig. 3, which modifies our example from Fig. 1 to represent bitmaps in one of two ways (lines 12–14 or 16–18), depending on whether the `COMPACT` macro has been previously defined (line 23 or 25). The value of `COMPACT` itself depends on whether

`__BSD__` is set, which is determined by the initial preprocessor environment when the compiler is invoked (more on this below). We say that a file f_1 *depends on* file f_2 when f_1 uses some macro `M` set by f_2. In this case we also say that f_1 depends on `M`. Here, `bitmap.h` depends on `config.h`.

Such preprocessor-based dependencies are very useful, since they allow programs to be configured for different circumstances. However, they can also unintentionally cause a header to be preprocessed differently depending on where it is included. In Fig. 3, if we were to swap lines 8 and 9 but leave lines 28 and 29 alone, then `bitmap.c` and `main.c` would have different, incompatible types for `init`, and `main.c` might therefore invoke `init` with the wrong arguments (line 34 or 36). Thus, the preprocessor can undermine information hiding and type-safe linking, even when files satisfy Rules 1 and 2.

To solve this problem, we introduce two additional rules, discussed below, to enforce the following principle:

Principle 2.3 (Consistent Interpretation): Each header in the system that is used as an interface must have a *consistent interpretation*, meaning that whenever the header mediates linking, to enforce Rule 1, or owns a type definition, to enforce Rule 2, the text produced by preprocessing the header is identical wherever it is included.

Enforcing this principle allows us to keep Rules 1 and 2 simple, and it makes it easier for programmers to reason about headers, since their meaning is less context-dependent (though not entirely, as we discuss below). This is the same principle underlying proper use of *precompiled headers* [27], and thus programs that adhere to CMOD's rules can also use such headers safely.

The first rule to ensure consistent interpretation enforces safe idioms for header file inclusion:

Rule 3 (Proper Inclusion): Header files that act as interfaces must be vertically independent, must ignore duplicate inclusions, and must avoid inclusion cycles.

We say that file h *is vertically dependent on* f if h depends on f and h is `#included` after processing f in the course of processing a given source file. This could happen, for example, when a source file first `#includes` h and then `#includes` some f that depends on h. In the example, `bitmap.h` is vertically dependent on `config.h`. As another example, a source file f could `#define` a macro that a header h it subsequently `#includes` depends on. Eliminating vertical dependencies ensures that the interpretation of a header is the same no matter the order the header is included in a source file.

a.h
```
#ifndef A_H
#define A_H
#include "b.h"
#ifdef X
  extern int x;
#else
  extern float x;
#endif
#endif
```

b.h
```
#ifndef B_H
#define B_H
#include "a.h"
#define X
#endif
```

Fig. 4. Pathological cyclic dependence between two headers

We forbid vertical dependencies because we believe they add unnecessary complication. In particular, the programmer must remember to always include the headers together, in some particular order. We believe a better practice is to convert vertical dependencies into *horizontal dependencies*, which are more self-contained. We say that two header files are *horizontally dependent* if one of the headers is dependent on *and* `#include`s the other. A horizontal dependence adheres to Principle 2.3 because a header always "carries along" the other headers on which it depends, ensuring a consistent interpretation.

If we wanted to remove the vertical dependence in the example, we could convert it to a horizontal dependence by moving line 8 just prior to line 1. However, notice that then `config.h` would be included twice in `main.c`, once directly and once via `bitmap.h`. The double inclusion is harmless because of the `#ifndef` pattern [7], [13] beginning on line 20, which causes any duplicate inclusions of `config.h` to be completely ignored. Our implementation requires that the `#ifndef` pattern is used in every header to eliminate duplicate inclusions.

The `#ifndef` pattern is essentially a kind of self-dependence. Somewhat surprisingly, such self-dependencies can result in violations of the consistent interpretation principle in the presence of recursive inclusion. Fig. 4(a) illustrates the issue. Here `a.h` first includes `b.h` and then, depending on the value of macro `X`, declares `x` to be either an `int` or a `float`. The header `b.h` first includes `a.h`, and then defines `X`. Given that we allow self-dependence, `a.h` is horizontally dependent on `b.h`, which is permitted, and there are no vertical dependencies.

Suppose that one source file contains #include "a.h" and another contains #include "b.h". In the first case, X will be defined (from the nested inclusion of b.h), and therefore x will have type int. In the second case, X will not be defined when a.h is included, since the second inclusion of b.h is nullified by the #ifndef pattern; hence x will be of type float. Thus these two files satisfy Rule 1 (including a common header), but disagree on the type of x, violating consistent interpretation.[2]

The root of the problem is that the two headers form an inclusion cycle, but the dependencies between them cause their interpretation to differ depending on which is included first. We can recover consistent interpretation while allowing self-dependence by forbidding cyclic header inclusion. Fortunately, this restriction does not appear to be onerous: none of our benchmarks had any cases of recursive inclusion. We did find one instance of recursion in limits.h from the GNU standard C library, but this particular case was both highly unusual and benign, involving GCC-specific preprocessor directives to include two files with the same name from different directories.

Note that Rule 3 allows vertical dependencies on files that are not meant to be interfaces. We have found this flexibility to be useful in practice. For example, the gawk distribution builds two executables, gawk and pgawk, where the latter performs extra profiling. To implement this, the developers #include the file eval.c in the file eval_p.c, first defining a macro to change its processing:

```
#define PROFILING
#include "eval.c"
```

Then gawk is linked with eval.c, and pgawk is linked with eval_p.c. We have seen similar parameterizations in other programs, including bison and gnuplot.

Clearly eval.c is vertically dependent on eval_p.c, since eval_p.c defines a macro that affects the processing of the eval.c. Nevertheless, this is not a Rule 3 violation because eval.c is not being used as an interface. CMOD makes this intuition precise by considering eval.c to be *inlined* within eval_p.c. Thus, when checking Rule 1 for pgawk, eval.c is not considered a shared header, and when checking Rule 2, any types textually appearing in eval.c are considered

[2]These subtle dependencies could be the reason that some coding style guides encourage vertical dependencies in lieu of horizontal ones [2].

owned by `eval_p.c`. In our implementation, we heuristically assume that files ending in `.h` are meant to be interfaces, while all other included files are not, and are thus treated as inlined. It would be interesting future work to discover this distinction based on usage, rather than filename extension.

Preventing vertical dependencies solves one problem with the preprocessor, but we also need to reason about the initial preprocessor environment. Recall that the `__BSD__` flag used in lines 22–26 of Fig. 3 is not set within the file. Instead, it is either supplied by the system or induced by a compiler command-line option (e.g., as an argument to `-D`).[3] If `bitmap.c` were compiled with this flag set and `main.c` were compiled without it, then the two inclusions of `bitmap.h` (lines 9 and 29) would produce different declarations of `init`. We can prevent this by enforcing CMOD's final rule:

Rule 4 (Consistent Environment): All files linked together must be compiled in a consistent preprocessor environment.

By *consistent* we mean that for any pair of linked files that depend on a macro `M`, the macro must be defined (or not defined) identically in the initial preprocessor environments for each file. Processing each module in a consistent environment ensures that all of its included headers (which by Rule 3 are not vertically dependent) are interpreted the same way everywhere, following Principle 2.3.

D. Discussion

In essence, Rules 3 and 4 allow the program—all its linked source files and their interfaces—to be treated as a very large functor [26], parameterized by the initial preprocessor environment and optionally by a uniformly-included `config.h` (see below). Thus while CMOD allows individual headers to be parameterized, they must be consistently interpreted throughout the program if they are to be treated as interfaces. Consistent interpretation works well in practice: Since a `.h` file acting as an interface represents a `.c` file that is typically compiled once, there is usually little reason to interpret the `.h` file differently in different contexts.

While we feel that vertical dependencies between interfaces are generally undesirable, the interfaces in many large programs are vertically dependent on a `config.h` header like the one

[3]Note that flags other than `-D` can affect the environment. For example, passing the `-O` flag causes the `__OPTIMIZE__` macro to be set and `__NO_INLINE__` to be un-set.

ti.h

```
1  typedef int T;
```

tp.h

```
2  typedef int *T;
```

a.h

```
3  extern T tvar;
```

ti.c

```
4  #include "ti.h"
5  #include "a.h"
6  /* tvar has type int */
```

tp.c

```
7  #include "tp.h"
8  #include "a.h"
9  /* tvar has type int* */
```

Fig. 5. Non-preprocessor dependency among header files

in Fig. 3. This is safe—that is, it ensures consistent interpretation—as long as `config.h` is always included *first* so that other included headers are consistently interpreted with respect to it. Thus, CMOD allows the programmer to optionally supply the name of a `config.h` file, and vertical dependencies on the `config.h` file are permitted. CMOD also checks that `config.h` is included first in every file. In essence, we can think of `config.h` as part of the initial macro environment, so this relaxation is in the spirit of Rule 4.

Note that while Principle 2.3 ensures consistent interpretation of headers, this does not imply that a header *means* the same thing wherever it is included. This is because a header is likely to refer to type definitions that precede it, and, more rarely, variable definitions if the header contains `static` (possibly inline) functions, or macro definitions that include code.

For example, consider the code in Fig. 5. Here the header files `ti.h` and `tp.h` each have a different definition of type `T` (lines 1 and 2), which is used in header `a.h` (line 3). Source file `ti.c` includes `a.h` after `ti.h`, and thus in this file `tvar` has type `int`. However, source file `tp.c` includes `tp.h` first, and thus in this file `tvar` has type `int *`. Notice that there are no vertical dependencies as we have defined them, since none of the three header files use any preprocessor directives, and thus produce the same text no matter where they are included. However, the meaning of `T` within `a.h` has changed, depending on which source file included the header.

Fortunately, allowing this situation to occur does not compromise either information hiding or type-safe linking. In particular, Rule 2 requires that every type is owned by exactly one file which, for our example, would preclude `ti.c` and `tp.c` from being linked together. Symbols

$$
\begin{array}{rccl}
\text{program} & \mathcal{P} & ::= & \cdot \mid f \circ \mathcal{P} \\
\text{fragment} & f & := & \cdot \mid s, f \\
\text{statements} & s & ::= & c \mid d \\
\text{preproc. commands} & c & ::= & \mathsf{def}\ m \mid \mathsf{undef}\ m \mid \mathsf{ifdef}\ m\ \mathsf{then}\ f\ \mathsf{else}\ f \\
 & & \mid & \mathsf{import}\ h \mid \mathsf{inline}\ h \mid \mathsf{end}\ h \\
\text{definitions} & d & ::= & \mathsf{let}\ g : \tau = e \mid \mathsf{extern}\ g : \tau \\
 & & \mid & \mathsf{lettype}\ t = \tau \mid \mathsf{type}\ t \\
\text{terms} & e & ::= & n \mid \lambda y : \tau .\ e \mid e\ e \mid y \mid g \\
\text{types} & \tau & ::= & t \mid \mathsf{int} \mid \tau \to \tau
\end{array}
$$

$$
\begin{array}{rr}
m \in \text{macro names} & g \in \text{global var. names} \\
h \in \text{file names} & t \in \text{type names} \\
y \in \text{local var. names} & n \in \mathbb{Z}
\end{array}
$$

Fig. 6. Source language

can be used a bit more flexibly, but are still safe. The standard linker forbids multiple definitions of exported symbols, while `static` symbol definitions cannot be linked against from different files, thus precluding any sort of type-safe linking or information hiding violation among them.

Another possible design point for CMOD would be to require `static` symbols to be singly-defined, just like exported symbols, to make code easier to understand. However, extending CMOD to track such dependencies would add significant implementation complexity when compared to our current approach (Section IV), and in our experience, dependencies on symbols are rare.

III. Formal Development

In this section, we describe CMOD precisely by formalizing its four rules on a small preprocessor and source language. Using this formalism, we prove that our rules are sound. Fig. 6 presents the core language. Here, a source program $\mathcal{P}$ consists of a list of fragments f.[4] At the top level of a program $\mathcal{P}$, a fragment represents a separately-compiled source file, where the program is what results from linking the fragments together. Syntactically, a fragment is

[4]The term *fragment* is due to Cardelli [3].

just an ordered list of program statements s, which may be either preprocessor commands c or definitions d.

Preprocessor commands c model those of the C preprocessor. The commands def m and undef m respectively define and undefine the preprocessor macro m from that point forward. The conditional ifdef m then f_1 else f_2 processes f_1 if m is defined, and otherwise processes f_2. Since each branch is a fragment, it may contain further preprocessor commands.

Our source language uses two distinct commands to model C's `#include` directive: import h represents importing a module interface, and inline h represents any other uses of file inclusion. Both commands cause the file h to be textually inserted, but import has two additional behaviors: First, any occurrences of import h after the first one are treated as no-ops; this models the `#ifndef` pattern (Section II-C), which avoids duplicate inclusion. Second, recursive imports are disallowed, which enforces part of Rule 3. In contrast, inline performs pure textual inclusion, allowing duplicate and recursive inlining, if present. In our implementation, we treat occurrences of `#include` as import when the name of the included file ends with extension `.h`, in which case we also check that it uses the `#ifndef` pattern and does not recursively include itself. All other uses of `#include` are modeled by inline. The last preprocessor command, end h, is inserted by the preprocessor to mark the end of an imported file, and never appears in a source program.

Core language definitions d model their C counterparts. The definition let $g : \tau = e$ binds the global name g to term e of type τ; this form represents C global variable and function definitions. Since we are interested in linking, the form of e itself is unimportant, and so we use simply-typed lambda calculus terms for convenience. The definition extern $g : \tau$ is analogous to C's `extern`, and declares the existence of global g of type τ, which is used in header files to import a symbol. The definition lettype $t = \tau$ is analogous to C's `struct` or `typedef` definitions, and defines a named type t to be an alias for τ. Finally, the definition type t declares that t may be used as a type name, which is analogous to a C `struct` declaration where the name of the `struct` is declared but no fields are given. We say that g and t are *defined* by let g and lettype $t = \tau$, while g and t are *declared* by extern $g : \tau$ and type t. Within a program we allow many declarations of a global variable or type name but only one definition.

Our formalism simplifies features of both C's preprocessor and proper language to make formal proofs more tractable. Section IV-B discusses the differences in more detail and argues that our formal soundness result still applies to the full C language.

A. Preprocessor Semantics

We begin by defining an operational semantics for our language. Our semantics has three phases. First, we generate *traces* by executing preprocessor commands and recording the sequence of actions. Having traces allows us to attribute actions to particular header files included within a larger evaluation, e.g., the trace that starts with the inclusion of h and ends with processing $\mathsf{end}\ h$ describes the contents of h. Second, we convert traces into *accumulators*, which contain (unordered) summary information, e.g., the set of macros defined in a file, or the types of each exported symbol. Most of CMOD's rules are specified as properties of accumulators, but specifying order-independence requires appealing to traces. Lastly, we *compile* the accumulator into an object file. In this subsection we discuss the first two phases, and defer compilation to Section III-C.

The rules for trace generation are given in Fig. 7. A *trace* $\tilde{f}$ consists of core language definitions and *trace commands* $\tilde{c}$, which represent the decisions that have been made during preprocessing. Trace commands $\mathsf{def}\ m$ and $\mathsf{undef}\ m$ represent the definition or undefinition of m, respectively, and $\mathsf{ifdef}\ m^+$ and $\mathsf{ifdef}\ m^-$ represent a conditional in which m was defined or not defined, respectively. The trace command $\mathsf{import}\ h$ records the inclusion of h due to import, and the trace command $\mathsf{nullimport}\ h$ represents a duplicate import of h that was nulled-out. Occurrences of inline are not separately recorded in the trace. Lastly, $\mathsf{end}\ h$ indicates the completion of h's preprocessing.

Trace generation is specified as a reduction from state to state, where a *state* has the form $\left\langle \tilde{f}; \mathcal{I}; \Delta; f \right\rangle$. Here $\tilde{f}$ is a trace of actions thus far, $\mathcal{I}$ is a set of header files that have been (possibly partially) preprocessed, Δ is a set of currently-defined macros, and f is the remaining source fragment to be preprocessed. Reduction judgments have the form $\mathcal{F} \vdash \left\langle \tilde{f}; \mathcal{I}; \Delta; f \right\rangle \longrightarrow \left\langle \tilde{f}'; \mathcal{I}'; \Delta'; f' \right\rangle$ where $\mathcal{F}$ is a *file system* that maps header names to fragments, and is used when an import or inline command is encountered. Preprocessing fragment f begins with $\tilde{f}$ set to the empty trace $(\cdot)$, $\mathcal{I}$ set to $\emptyset$, a given $\mathcal{F}$, and an initial (possibly empty) set of macro definitions Δ. We call this initial set of definitions the *initial environment*. In practice, Δ is supplied by the user on the command line when the compiler is invoked (e.g., by using `-D` options). The initial environment can therefore vary from one fragment to another, most typically for projects that build intermediate libraries, which might be compiled with some set of flags not used by the

$$
\begin{array}{rccl}
\text{trace} & \tilde{f} & := & \cdot \mid \tilde{s}, \tilde{f} \\
\text{trace statements} & \tilde{s} & ::= & \tilde{c} \mid d \\
\text{trace commands} & \tilde{c} & ::= & \mathsf{def}\ m \mid \mathsf{undef}\ m \mid \mathsf{ifdef}\ m^{+} \\
 & & \mid & \mathsf{ifdef}\ m^{-} \mid \mathsf{import}\ h \mid \mathsf{nullimport}\ h \mid \mathsf{end}\ h \\
\text{includes} & \mathcal{I} & \in & 2^{h} \\
\text{defines} & \Delta & \in & 2^{m} \\
\text{file system} & \mathcal{F} & : & h \rightarrow f
\end{array}
$$

[DEF]

$$\frac{\Delta' = \Delta \cup \{m\} \qquad \tilde{f}' = \tilde{f}, \mathsf{def}\ m}{\mathcal{F} \vdash \left\langle \tilde{f}; \mathcal{I}; \Delta; \mathsf{def}\ m, f \right\rangle \longrightarrow \left\langle \tilde{f}'; \mathcal{I}; \Delta'; f \right\rangle}$$

[UNDEF]

$$\frac{\Delta' = \Delta \setminus \{m\} \qquad \tilde{f}' = \tilde{f}, \mathsf{undef}\ m}{\mathcal{F} \vdash \left\langle \tilde{f}; \mathcal{I}; \Delta; \mathsf{undef}\ m, f \right\rangle \longrightarrow \left\langle \tilde{f}'; \mathcal{I}; \Delta'; f \right\rangle}$$

[IFDEF+]

$$\frac{m \in \Delta \qquad \tilde{f}' = \tilde{f}, \mathsf{ifdef}\ m^{+}}{\mathcal{F} \vdash \left\langle \tilde{f}; \mathcal{I}; \Delta; (\mathsf{ifdef}\ m\ \mathsf{then}\ f_{+}\ \mathsf{else}\ f_{-}), f \right\rangle \longrightarrow \left\langle \tilde{f}'; \mathcal{I}; \Delta; f_{+}, f \right\rangle}$$

[IFDEF-]

$$\frac{m \notin \Delta \qquad \tilde{f}' = \tilde{f}, \mathsf{ifdef}\ m^{-}}{\mathcal{F} \vdash \left\langle \tilde{f}; \mathcal{I}; \Delta; (\mathsf{ifdef}\ m\ \mathsf{then}\ f_{+}\ \mathsf{else}\ f_{-}), f \right\rangle \longrightarrow \left\langle \tilde{f}'; \mathcal{I}; \Delta; f_{-}, f \right\rangle}$$

[IMPORT]

$$\frac{h \notin \mathcal{I} \qquad \mathcal{I}' = \mathcal{I} \cup \{h\} \qquad \tilde{f}' = \tilde{f}, \mathsf{import}\ h}{\mathcal{F} \vdash \left\langle \tilde{f}; \mathcal{I}; \Delta; \mathsf{import}\ h, f \right\rangle \longrightarrow \left\langle \tilde{f}'; \mathcal{I}'; \Delta; \mathcal{F}(h), \mathsf{end}\ h, f \right\rangle}$$

[IMPORT-EMPTY]

$$\frac{h \in \mathcal{I} \qquad \mathsf{end}\ h \notin f \qquad \tilde{f}' = \tilde{f}, \mathsf{nullimport}\ h}{\mathcal{F} \vdash \left\langle \tilde{f}; \mathcal{I}; \Delta; \mathsf{import}\ h, f \right\rangle \longrightarrow \left\langle \tilde{f}'; \mathcal{I}; \Delta; f \right\rangle}$$

[EOH]

$$\frac{\tilde{f}' = \tilde{f}, \mathsf{end}\ h}{\mathcal{F} \vdash \left\langle \tilde{f}; \mathcal{I}; \Delta; \mathsf{end}\ h, f \right\rangle \longrightarrow \left\langle \tilde{f}'; \mathcal{I}; \Delta; f \right\rangle}$$

[INLINE]

$$\frac{}{\mathcal{F} \vdash \left\langle \tilde{f}; \mathcal{I}; \Delta; \mathsf{inline}\ h, f \right\rangle \longrightarrow \left\langle \tilde{f}; \mathcal{I}; \Delta; \mathcal{F}(h), f \right\rangle}$$

[TERM]

$$\frac{\tilde{f}' = \tilde{f}, d}{\mathcal{F} \vdash \left\langle \tilde{f}; \mathcal{I}; \Delta; d, f \right\rangle \longrightarrow \left\langle \tilde{f}'; \mathcal{I}; \Delta; f \right\rangle}$$

Fig. 7. Trace generation

main part of the project.

We briefly discuss the rules in Fig. 7. [DEF] and [UNDEF] add or remove m from the set of currently-defined macros Δ and record the command in the trace. [IFDEF+] and [IFDEF-] reduce to either f_+ or f_- depending on whether m has been defined or not, and record the decision in the output trace.

The semantics of import is given by the next two rules. [IMPORT] applies when the header h has not yet been preprocessed, in which case import h expands to the fragment $\mathcal{F}(h)$ with end h appended to it to mark the end of the file. The included file h is also added to $\mathcal{I}$ in the output state. [IMPORT-EMPTY], on the other hand, applies when h has already been preprocessed, in which case no expansion occurs and nullimport h is added to the trace. Since [IMPORT] ensures $(\mathsf{end}\ h) \in f$ while h is being preprocessed, [IMPORT-EMPTY] requires that $(\mathsf{end}\ h) \notin f$ to forbid h from recursively including itself, as required by Rule 3.

[EOH] simply records the marker end h in the trace. [INLINE] expands to the contents of $\mathcal{F}(h)$. Notice that we do not record any effect in the trace, nor do we tag the end of the file. The latter choice means that any definitions inside of h are attributed to the including file for purposes of rule checking. Lastly, [TERM] copies a definition d, which may be a let, extern, lettype, or type, into the trace.

Fig. 8 gives the rules for producing an *accumulator* from a trace. An accumulator $\mathcal{A}$ is a tuple that summarizes information about the core language program and macro usage. The first three components of the accumulator are lists that track information about the core language program: N maps global variables to their types; H maps global variables to their defining expressions; and T maps each type name t to its definition τ. In T, types are annotated with either the header file h in which the type was defined, or $\circ$ if it was defined in a source file rather than a header file. The next three components of the accumulator record information about symbols, namely the set of global variables that have been exported (E) by defining them with let, and imported (I) by referring to them in declarations or terms. Finally, the last three components of the accumulator record information about macros that have been changed ($\mathcal{C}$) or used ($\mathcal{U}$), and the set of type names that have been declared (Z). For macros in $\mathcal{C}$ or $\mathcal{U}$, we also record the file in which the change or use occurred.

The first two rules in Fig. 8 define the function *first-end*$(\tilde{f})$, which returns the file name from the leftmost, non-matched occurrence of end in $\tilde{f}$ ([IN-HEADER]), or $\circ$ if there is no such

$$\begin{array}{rrcl}
\text{symbols} & N & ::= & \cdot \mid g \rightarrow \tau,\ N \\
\text{heap} & H & ::= & \cdot \mid g \rightarrow e,\ H \\
\text{named types} & T & ::= & \cdot \mid t \rightarrow \tau^h,\ T \mid t \rightarrow \tau^\circ,\ T \\
\text{exports} & E & \in & 2^g \qquad \text{macro uses} \quad \mathcal{U} \in 2^{m\times h} \\
\text{imports} & I & \in & 2^g \qquad \text{type decls} \quad Z \in 2^t \\
\text{macro changes} & \mathcal{C} & \in & 2^{m\times h} \\
\text{accumulator} & \mathcal{A} & = & (N, H, T, E, I, \mathcal{C}, \mathcal{U}, Z)
\end{array}$$

[IN-HEADER]

$$\frac{\tilde f = \tilde f', \textsf{end}\ h, \tilde f'' \qquad \textsf{import}\ h \notin \tilde f' \qquad \forall h''.\ \textsf{end}\ h'' \in \tilde f' \implies \textsf{import}\ h'' \in \tilde f'}{h = \textit{first-end}(\tilde f)}$$

[IN-SOURCE]

$$\frac{\forall h.\ \textsf{end}\ h \in \tilde f \implies \textsf{import}\ h \in \tilde f}{\circ = \textit{first-end}(\tilde f)}$$

[DEF]

$$\frac{h = \textit{first-end}(\tilde f) \qquad \mathcal{A}' = \mathcal{A}[\mathcal{C} \leftarrow^{+} (m,h),\ \mathcal{U} \leftarrow^{+} (m,h)]}{\left\langle \mathcal{A}; \textsf{def}\ m, \tilde f \right\rangle \longrightarrow \left\langle \mathcal{A}'; \tilde f \right\rangle}$$

[UNDEF]

$$\frac{h = \textit{first-end}(\tilde f) \qquad \mathcal{A}' = \mathcal{A}[\mathcal{C} \leftarrow^{+} (m,h),\ \mathcal{U} \leftarrow^{+} (m,h)]}{\left\langle \mathcal{A}; \textsf{undef}\ m, \tilde f \right\rangle \longrightarrow \left\langle \mathcal{A}'; \tilde f \right\rangle}$$

[IFDEF±]

$$\frac{h = \textit{first-end}(\tilde f) \qquad \mathcal{A}' = \mathcal{A}[\mathcal{U} \leftarrow^{+} (m,h)]}{\left\langle \mathcal{A}; \textsf{ifdef}\ m^{\pm}, \tilde f \right\rangle \longrightarrow \left\langle \mathcal{A}'; \tilde f \right\rangle}$$

[IMPORT]

$$\frac{}{\left\langle \mathcal{A}; \textsf{import}\ h, \tilde f \right\rangle \longrightarrow \left\langle \mathcal{A}; \tilde f \right\rangle}$$

[IMPORT-EMPTY]

$$\frac{}{\left\langle \mathcal{A}; \textsf{nullimport}\ h, \tilde f \right\rangle \longrightarrow \left\langle \mathcal{A}; \tilde f \right\rangle}$$

[EOH]

$$\frac{}{\left\langle \mathcal{A}; \textsf{end}\ h, \tilde f \right\rangle \longrightarrow \left\langle \mathcal{A}; \tilde f \right\rangle}$$

[EXTERN]

$$\frac{\mathcal{A}' = \mathcal{A}[N \leftarrow^{+} (g \mapsto \tau)]}{\left\langle \mathcal{A}; \textsf{extern}\ g : \tau, \tilde f \right\rangle \longrightarrow \left\langle \mathcal{A}'; \tilde f \right\rangle}$$

[LET]

$$\frac{\begin{array}{c}\mathcal{A}' = \mathcal{A}[H \leftarrow^{+} (g \mapsto e),\ N \leftarrow^{+} (g \mapsto \tau), \\ E \leftarrow^{+} g,\ I \leftarrow^{+} \textsf{fg}\,(e)] \qquad g \notin \mathcal{A}^H\end{array}}{\left\langle \mathcal{A}; \textsf{let}\ g : \tau = e, \tilde f \right\rangle \longrightarrow \left\langle \mathcal{A}'; \tilde f \right\rangle}$$

[TYPE-DECL]

$$\frac{\mathcal{A}' = \mathcal{A}[Z \leftarrow^{+} t]}{\left\langle \mathcal{A}; \textsf{type}\ t, \tilde f \right\rangle \longrightarrow \left\langle \mathcal{A}'; \tilde f \right\rangle}$$

[TYPE-DEF]

$$\frac{h = \textit{first-end}(\tilde f) \qquad t \notin \mathcal{A}^T \qquad \mathcal{A}' = \mathcal{A}[T \leftarrow^{+} (t \mapsto \tau^h)]}{\left\langle \mathcal{A}; \textsf{lettype}\ t = \tau, \tilde f \right\rangle \longrightarrow \left\langle \mathcal{A}'; \tilde f \right\rangle}$$

Fig. 8. Accumulator generation

occurrence ([IN-SOURCE]).

The remaining rules define accumulator generation as a set of reduction rules on states $\left\langle \mathcal{A}; \tilde{f} \right\rangle$, where $\mathcal{A}$ is the accumulator thus far and $\tilde{f}$ is the remaining part of the trace. We write $\mathcal{A}[X \leftarrow^{+} x]$ for the accumulator that is the same as $\mathcal{A}$ except that its X component has x added to it. Accumulator generation starts with an accumulator whose components are all $\emptyset$, which we write $\mathcal{A}_\emptyset$, and all of the rules monotonically add to the accumulator.

[DEF] and [UNDEF] mark m as being changed and used; counting both as uses is most likely not required, but our proof technique relies on it [30]. [IFDEF$\pm$] marks m as being used. All three rules use *first-end* to determine in what file the macro change and/or use occurs. [IMPORT], [IMPORT-EMPTY], and [EOH] all have no effect on the accumulator. The last four rules handle declarations and definitions. [EXTERN] records the declaration of g and notes its type in N. Here we append the typing $(g \mapsto \tau)$ onto the list N, i.e., we do not replace any previous bindings for g. The compiler ensures that the same variable is always given the same type within a fragment (Section III-C). [LET] adds g to the set of defined global variables H, adds g's type to N, and adds any global variables mentioned in e (written $\mathrm{fg}\,(e)$) to the imports. Finally, [TYPE-DECL] declares a type, which is noted in Z, and [TYPE-DEF] defines a type, which is noted in T and tagged with the containing file using *first-end*.

B. CMOD *Rules*

We now formally specify the four rules presented in Section II. To state the rules more concisely, we use the following notation to describe a file's complete processing:

Definition 3.1 (Complete Preprocessing): We write $\Delta; \mathcal{F} \vdash f \rightsquigarrow \mathcal{A}; \mathcal{I}$ as shorthand for $\mathcal{F} \vdash \langle \cdot; \emptyset; \Delta; f \rangle \longrightarrow^{*} \left\langle \tilde{f}; \mathcal{I}; \Delta'; \cdot \right\rangle$ and $\left\langle \mathcal{A}_\emptyset; \tilde{f} \right\rangle \longrightarrow^{*} \langle \mathcal{A}; \cdot \rangle$.

CMOD's rules are shown in Fig. 9. To reduce notation, we write $\mathcal{A}^X$ for the X component of $\mathcal{A}$. The first three rules (parts (a)–(c)) assume there is a common initial macro environment Δ under which all fragments are preprocessed, and the fourth rule (part (d)) ensures this assumption makes sense. Fig. 9(a) defines the judgment $\Delta; \mathcal{F} \vdash \mathcal{R}_1(f_1, f_2)$, which enforces Rule 1: for each pair of fragments f_1 and f_2 in the program, any global variable defined in one and used in the other must be declared in a common header file. [RULE 1] uses auxiliary judgment $\Delta; \mathcal{F} \vdash g \xleftarrow{\text{decl}} \mathcal{I}$, which holds if g is declared by some header in the set $\mathcal{I}$, where we compute the declared variable names by preprocessing each header file h in isolation. Then for any variable name g

[SYM-DECL]

$$\frac{h \in \mathcal{I} \qquad \Delta; \mathcal{F} \vdash h \rightsquigarrow \mathcal{A}; \mathcal{I}' \qquad g \in \mathit{dom}(\mathcal{A}^N)}{\Delta; \mathcal{F} \vdash g \overset{\text{decl}}{\longleftarrow} \mathcal{I}}$$

[RULE 1]

$$\frac{\Delta; \mathcal{F} \vdash f_1 \rightsquigarrow \mathcal{A}_1; \mathcal{I}_1 \qquad \Delta; \mathcal{F} \vdash f_2 \rightsquigarrow \mathcal{A}_2; \mathcal{I}_2 \quad N = \left(\mathcal{A}_1^I \cap \mathcal{A}_2^E\right) \cup \left(\mathcal{A}_1^E \cap \mathcal{A}_2^I\right) \quad \forall g \in N \;.\; \Delta; \mathcal{F} \vdash g \overset{\text{decl}}{\longleftarrow} \mathcal{I}_1 \cap \mathcal{I}_2}{\Delta; \mathcal{F} \vdash \mathcal{R}_1(f_1, f_2)}$$

(a) Rule 1: Shared Headers

[NAMED-TYPES-OK]

$$\frac{(t \mapsto \tau^\circ) \in T_1 \implies t \notin \text{dom}\,(T_2) \quad (t \mapsto \tau^\circ) \in T_2 \implies t \notin \text{dom}\,(T_1) \quad \left(T_1(t) = \tau_1^{h_1}\right) \wedge \left(T_2(t) = \tau_2^{h_2}\right) \implies h_1 = h_2}{\vdash T_1, T_2}$$

[RULE 2]

$$\frac{\Delta; \mathcal{F} \vdash f_1 \rightsquigarrow \mathcal{A}_1; \mathcal{I}_1 \qquad \Delta; \mathcal{F} \vdash f_2 \rightsquigarrow \mathcal{A}_2; \mathcal{I}_2 \quad \vdash \mathcal{A}_1^T, \mathcal{A}_2^T}{\Delta; \mathcal{F} \vdash \mathcal{R}_2(f_1, f_2)}$$

(b) Rule 2: Type Ownership

[TRACE-INDEP]

$$\frac{\left\langle \mathcal{A}_\emptyset; \tilde{f}_1 \right\rangle \longrightarrow^* \langle \mathcal{A}_1; \cdot \rangle \qquad \left\langle \mathcal{A}_\emptyset; \tilde{f}_2 \right\rangle \longrightarrow^* \langle \mathcal{A}_2; \cdot \rangle \quad (m, h') \in \mathcal{A}_1^{\mathcal{C}} \wedge (m, h'') \in \mathcal{A}_2^{\mathcal{U}} \Rightarrow h' = h'' \quad (m, h') \in \mathcal{A}_1^{\mathcal{U}} \wedge (m, h'') \in \mathcal{A}_2^{\mathcal{C}} \Rightarrow h' = h''}{\tilde{f}_1 \otimes \tilde{f}_2}$$

[PARTIAL-INDEP]

$$\frac{\mathcal{F} \vdash \langle \cdot; \cdot; \Delta; f \rangle \longrightarrow^* \left\langle \tilde{f}_1, \mathsf{import}\ h; \mathcal{I}_1; \Delta_1; f_1 \right\rangle \quad \mathcal{F} \vdash \langle \cdot; \cdot; \Delta; f \rangle \longrightarrow^* \left\langle \tilde{f}_1, \tilde{f}_2, \mathsf{end}\ h; \mathcal{I}_2; \Delta_2; f_1 \right\rangle \quad \tilde{f}_1 \otimes \tilde{f}_2}{\Delta; \mathcal{F} \vdash f \otimes h}$$

[RULE 3]

$$\frac{\Delta; \mathcal{F} \vdash f \rightsquigarrow \mathcal{A}; \mathcal{I} \quad \forall h \in \mathcal{I} \;.\; \Delta; \mathcal{F} \vdash f \otimes h}{\Delta; \mathcal{F} \vdash \mathcal{R}_3(f)}$$

(c) Rule 3: Vertical Independence

[RULE 4]

$$\frac{\Delta_f; \mathcal{F} \vdash f \rightsquigarrow \mathcal{A}; \mathcal{I} \quad ((\Delta - \Delta_f) \cup (\Delta_f - \Delta)) \cap \mathcal{A}^{\mathcal{U}} = \emptyset}{\Delta; \mathcal{F} \vdash \mathcal{R}_4(f, \Delta_f)}$$

(d) Rule 4: Environment Compatibility

[ALL]

$$\frac{\forall f_1, f_2 \in \mathcal{P} \;.\; f_1 \neq f_2 \;.\; \Delta; \mathcal{F} \vdash \mathcal{R}_1(f_1, f_2) \quad \forall f_1, f_2 \in \mathcal{P} \;.\; f_1 \neq f_2 \;.\; \Delta; \mathcal{F} \vdash \mathcal{R}_2(f_1, f_2) \quad \forall f \in \mathcal{P} \;.\; \Delta; \mathcal{F} \vdash \mathcal{R}_3(f) \quad \forall f \in \mathcal{P} \;.\; \Delta; \mathcal{F} \vdash \mathcal{R}_4(f, \mathcal{E}(f))}{\Delta; \mathcal{E}; \mathcal{F} \vdash \mathcal{R}(\mathcal{P})}$$

(e) Rules 1–4 combined

Fig. 9. CMOD Rules

in N (which contains names imported by one fragment and defined by the other), it must be the case that $\Delta; \mathcal{F} \vdash g \overset{\text{decl}}{\longleftarrow} \mathcal{I}_1 \cap \mathcal{I}_2$, i.e., g is declared in a header file that both f_1 and f_2 include. By the consistent interpretation principle, enforced by Rules 3 and 4, we know that each file sees the *same* declaration of g.

Fig. 9(b) defines the judgment $\Delta; \mathcal{F} \vdash \mathcal{R}_2(f_1, f_2)$, which enforces Rule 2: each named type must have exactly one owner, either a source or a header. This rule examines two fragments, preprocessing each and using [NAMED-TYPES-OK] to check that the resulting type definition maps T_1 and T_2 are compatible. There are two cases. First, any type t in T_1 with no marked owner is owned by f_1, and thus should be abstract everywhere else, meaning t should not appear in T_2 (and vice versa). Note that we are justified in treating T_i as a map because the compiler forbids the same type name from being defined twice. Second, any type t appearing in both T_1 and T_2 is transparent and hence must be owned by the same header. Then by Rules 3 and 4, we know that τ_1 and τ_2 are the same.

Fig. 9(c) defines the judgment $\Delta; \mathcal{F} \vdash \mathcal{R}_3(f)$, which enforces the key part of Rule 3: any two headers h_1 and h_2 that are both included in some fragment must be vertically-independent. (The other requirements of Rule 3 are that recursive includes are forbidden and duplicate imports are nulled-out, and both of these are enforced by [IMPORT-EMPTY] from Fig. 7.) For each header h included in f, [RULE 3] checks $\Delta; \mathcal{F} \vdash f \otimes h$, defined by [PARTIAL-INDEP]. The first two premises of [PARTIAL-INDEP] preprocess f, resulting in the trace $\tilde{f}_1$ up to the (only) import of h, and the trace $\tilde{f}_2$ that contains the full processing of h. The last premise $\tilde{f}_1 \otimes \tilde{f}_2$, defined by [TRACE-INDEP], checks that the preprocessing steps taken in $\tilde{f}_1$ do not influence the steps taken in $\tilde{f}_2$. In particular, no macros changed in $\mathcal{A}_1$ (described by $\mathcal{A}_1^{\mathcal{C}}$) are used by h (described by $\mathcal{A}_2^{\mathcal{U}}$), unless the macro change and use occurred in the same file, and thus h is vertically-independent of any files that came earlier.

Notice that in [TRACE-INDEP], we also require that no macros used in $\mathcal{A}_1$ are changed by h, i.e., we forbid a use before a change. Although this restriction may be surprising, we include it for two reasons. First, it seems desirable to make programs as robust as possible against the reordering of headers, and a use-before-change among headers could become a vertical dependency if those inclusions are for some reason swapped. Second, without this restriction, our formalization of Rule 3 would not enforce consistent interpretation. The reason for this is rather subtle, and a full explanation can be found in the appendix.

Also notice that this rule only refers to *imported* files, not *inlined* files. Since inlined files are not added to $\mathcal{I}$ and can never be type owners, they are not relevant to Rules 1 and 2. This means that they need not be consistently interpreted individually; rather, their contents are considered part of the including file, which may require a consistent interpretation if it is a header. Also notice that `config.h` files are forbidden by [RULE 3]. As mentioned earlier, our implementation allows the programmer to specify a `config.h` that all files must include first; the equivalent in our formal system is to start with an accumulator and initial Δ from preprocessing `config.h`.

Fig. 9(d) defines the judgment $\Delta; \mathcal{F} \vdash \mathcal{R}_4(f, \Delta_f)$, which enforces [RULE 4]: all fragments must be compiled in compatible environments. This rule holds if the initial environment Δ_f—in which f is assumed to have been compiled—agrees with Δ on those macros used by f (in $\mathcal{A}^{\mathcal{U}}$). This implies that preprocessing under Δ produces the same result as preprocessing under Δ_f.

Fig. 9(e) defines the judgment $\Delta; \mathcal{E}; \mathcal{F} \vdash \mathcal{R}(\mathcal{P})$, which holds if a program $\mathcal{P}$ satisfies Rules 1, 2, 3 in a common Δ that is consistent with $\mathcal{E}$ by Rule 4, where $\mathcal{E}$ maps each fragment to its initial environment (recall the initial environments may differ from one fragment to another). Thus if $\Delta; \mathcal{E}; \mathcal{F} \vdash \mathcal{R}(\mathcal{P})$ holds, then every pair of fragments in $\mathcal{P}$ must use shared headers for global variables, must have a single owner for each type definition, must use vertically-independent header files, and must be compiled in a consistent environment.

C. Formal Properties

To prove that the rules in Fig. 9 enforce Properties 2.1 and 2.2, we need to define precisely the effect of compilation and linking. Normally, a C compiler produces an object file containing code and data for globals, a list of exported symbols, and a list of imported symbols. To establish that linking is type-safe, we will also need to track type information about symbols. We use Glew and Morrisett's MTAL$_0$ typed object file notation [12], allowing us to appeal to their type safety result in our proof (though with some limitation as we discuss below). MTAL$_0$ typed object files have the form $[\Psi_I \Rightarrow H : \Psi_E]$, where H is a mapping from global names g to expressions e, and Ψ_I and Ψ_E are both mappings from global names to types τ. Here Ψ_I are the imported symbols and Ψ_E are the exported symbols.

We omit the full definition of compilation and linking as it is largely straightforward; details can be found in our companion technical report [30]. Fig. 10 shows the key rules. Rule [COMPILE] describes the object file produced by the C compiler from a fragment f, given

$$\frac{\Delta;\mathcal{F}\vdash f\leadsto(N,H,T,E,I,\mathcal{C},\mathcal{U},Z);\mathcal{I}\qquad\vdash N\qquad N\vdash H\qquad \Psi_E=N|_E\qquad \Psi_I=N|_{(I-E)}}{\Delta;\mathcal{F}\vdash f\xrightarrow{\text{comp}}[\Psi_I\Rightarrow H:\Psi_E]}\;[\textsc{Compile}]$$

$$\frac{\text{dom}\,(H_1)\cap\text{dom}\,(H_2)=\emptyset}{\begin{array}{l}\Delta;\mathcal{F}\vdash[\Psi_{I1}\Rightarrow H_1:\Psi_{E1}]\circ[\Psi_{I2}\Rightarrow H_2:\Psi_{E2}]\xrightarrow{\text{comp}}\\ \qquad[(\Psi_{I1}\cup\Psi_{I2})\setminus(\Psi_{E1}\cup\Psi_{E2})\Rightarrow H_1\cup H_2:\Psi_{E1}\cup\Psi_{E2}]\end{array}}\;[\textsc{Link}]$$

Fig. 10. Key Compiler and Linker Rules

an initial set of macro definitions Δ and a file system $\mathcal{F}$. The rule requires that following preprocessing, the global type environment N always assigns the same symbol the same type ($\vdash N$), and the code and data in the file are locally well-typed ($N\vdash H$; we discuss this judgment in more detail below).[5] Then the exported symbols Ψ_E are those that are defined (here $N|_E$ is the mapping N with its domain restricted to E), and the imported symbols Ψ_I are those that are declared but not defined. Rule [LINK] describes the process of linking two object files, which resolves imports and exports as expected. Because C's linker is untyped, there is almost no checking in this rule. The only thing required is that the two files not define the same symbols.

We can now formally state the information hiding and link-time type safety properties of CMOD. Proofs of the theorems in this section are in our companion technical report [30].

Observe that although each fragment f is preprocessed in its own initial Δ_f, by Rule 4 we can assume there is a single, uniform Δ under which each fragment produces the same result:

Lemma 3.2: $\Delta;\mathcal{F}\vdash\mathcal{R}_4(f,\Delta_f)$ implies that if $\Delta_f;\mathcal{F}\vdash f\leadsto\mathcal{A};\mathcal{I}$, then $\Delta;\mathcal{F}\vdash f\leadsto\mathcal{A};\mathcal{I}$; and if $\Delta_f;\mathcal{F}\vdash f\xrightarrow{\text{comp}}[\Psi_I\Rightarrow H:\Psi_E]$, then $\Delta;\mathcal{F}\vdash f\xrightarrow{\text{comp}}[\Psi_I\Rightarrow H:\Psi_E]$.

Thus below we assume a single Δ for all fragments. Moreover, given such a consistent environment, Rule 3 guarantees that header files are consistently interpreted:

Lemma 3.3 (Consistent Interpretation): If $\Delta;\mathcal{F}\vdash f\leadsto\mathcal{A};\mathcal{I}$ and $\Delta;\mathcal{F}\vdash\mathcal{R}_3(f)$ and $h\in\mathcal{I}$ and $\Delta;\mathcal{F}\vdash(\mathcal{F}(h),\mathsf{end}\ h)\leadsto\mathcal{A}_h;\mathcal{I}_h$, then $\mathcal{A}_h\subseteq\mathcal{A}$.

[5]Interestingly, because this rule refers to the accumulated results, the order of definitions and uses as they appear in the original fragment is irrelevant. Thus a fragment could legally use a variable before it is defined in the same file (assuming the use was type-safe). This formulation is simpler, and more flexible, than C's disallowance of forward references.

Thus wherever a header file is imported it produces the same result as if it were processed in isolation, and thus header files have the same meaning everywhere.

We begin with information hiding. First, observe that linking is commutative and associative, so that we are justified in linking files together in any order. Also, to be a well-formed executable, a program must have no free, unresolved symbols. Thus we can define the compilation of an entire program:

Definition 3.4 (Program Compilation): We write $\Delta; \mathcal{F} \vdash \mathcal{P} \xrightarrow{\text{comp}} [\emptyset \Rightarrow H : \Psi_E]$ as shorthand for compiling each fragment in $\mathcal{P}$ separately and then linking the results together to form $[\emptyset \Rightarrow H : \Psi_E]$.

Then we can prove that any symbol not in a header file is never imported, and thus is private.

Theorem 3.5 (Global Variable Hiding): Suppose $\Delta; \mathcal{E}; \mathcal{F} \vdash \mathcal{R}(\mathcal{P})$, suppose $\Delta; \mathcal{F} \vdash \mathcal{P} \xrightarrow{\text{comp}} [\emptyset \Rightarrow H_\mathcal{P} : \Psi_{E\mathcal{P}}]$, and suppose for all $f_i \in \mathcal{P}$ we have $\Delta; \mathcal{F} \vdash f_i \rightsquigarrow \mathcal{A}_{fi}; \mathcal{I}_{f_i}$, and for all $h_j \in \bigcup_i \mathcal{I}_{f_i}$ that $\Delta; \mathcal{F} \vdash \mathcal{F}(h_j) \rightsquigarrow \mathcal{A}_{hj}; \mathcal{I}_{h_j}$. Then for all $f_i \in \mathcal{P}$, $g \notin \bigcup_j \mathit{dom}(\mathcal{A}^N_{hj})$ implies $g \notin \Psi_{Ii}$ where $\Delta; \mathcal{F} \vdash f_i \xrightarrow{\text{comp}} [\Psi_{Ii} \Rightarrow H_i : \Psi_{Ei}]$.

This theorem says that if $\mathcal{P}$ obeys the CMOD rules and includes headers h_j (which have the same meaning everywhere by Lemma 3.3), then any symbol g that is not in $\mathit{dom}(\mathcal{A}^N_{hj})$ for any j (i.e., is not declared in any header file) is never imported.

For type names, we can prove a related property:

Theorem 3.6 (Type Definition Hiding): Suppose $\Delta; \mathcal{F} \vdash \mathcal{R}(\mathcal{P})$, and for some $f_i \in \mathcal{P}$ we have $\Delta; \mathcal{F} \vdash f_i \rightsquigarrow \mathcal{A}_i; \mathcal{I}_i$. If $(t \mapsto \tau^\circ) \in \mathcal{A}^T_i$ then for any fragment $f_j \in \mathcal{P}$ such that $f_i \neq f_j$ and $\Delta; \mathcal{F} \vdash f_j \rightsquigarrow \mathcal{A}_j; \mathcal{I}_j$, we have $t \notin \mathrm{dom}\left(\mathcal{A}^T_j\right)$. Also, if $(t \mapsto \tau^h) \in \mathcal{A}^T_i$, then $h \in \mathcal{I}_i$.

The first part of this theorem says that if $\mathcal{P}$ obeys the CMOD rules and contains fragment f_i, then any type t defined by f_i (and not in a header) is not defined by any other fragments $f_j \neq f_i$, which implies it must be treated abstractly by those fragments. The second part of the theorem says that if fragment f_i contains a declaration of a type t from header file h, then h must have been imported by f_i; and since by Lemma 3.3 headers have the same meaning everywhere, all fragments that get the type t from the same header assign it the same type. Together, Theorems 3.5 and 3.6 give us Property 2.1.

To show that linking is type-safe, we first prove that if the program compiles and passes the CMOD checks, then each pair of object files is well-formed and link-compatible. Well-formedness, according to the judgment $\vdash [\Psi_I \Rightarrow H : \Psi_E]$, implies that $[\Psi_I \Rightarrow H : \Psi_E]$'s definitions in H are well-typed internally and match the types given in Ψ_E, and that Ψ_E and Ψ_I are disjoint. Link-compatibility, according to the judgment $\vdash [\Psi_{Ii} \Rightarrow H_i : \Psi_{Ei}] \overset{\text{lc}}{\leftrightarrow} [\Psi_{Ij} \Rightarrow H_j : \Psi_{Ej}]$, implies that the types of imported and exported symbols common to the two files match and thus linking them will produce a well-formed object file.

Theorem 3.7 (Type-Safe Linking): Suppose $\Delta; \mathcal{E}; \mathcal{F} \vdash \mathcal{R}(\mathcal{P})$, and suppose $\Delta; \mathcal{F} \vdash \mathcal{P} \xrightarrow{\text{comp}} [\emptyset \Rightarrow H_{\mathcal{P}} : \Psi_{E\mathcal{P}}]$. Also suppose that for any $f_i, f_j \in \mathcal{P}$ that are distinct ($i \neq j$), it is the case that

$$\begin{array}{l} \Delta; \mathcal{F} \vdash f_i \xrightarrow{\text{comp}} [\Psi_{Ii} \Rightarrow H_i : \Psi_{Ei}] \\ \Delta; \mathcal{F} \vdash f_j \xrightarrow{\text{comp}} [\Psi_{Ij} \Rightarrow H_j : \Psi_{Ej}] \\ \Delta; \mathcal{F} \vdash [\Psi_{Ii} \Rightarrow H_i : \Psi_{Ei}] \circ [\Psi_{Ij} \Rightarrow H_j : \Psi_{Ej}] \xrightarrow{\text{comp}} O_{ij} \end{array}$$

Then $\vdash [\Psi_{Ii} \Rightarrow H_i : \Psi_{Ei}]$, $\vdash [\Psi_{Ij} \Rightarrow H_j : \Psi_{Ej}]$, and $\vdash [\Psi_{Ii} \Rightarrow H_i : \Psi_{Ei}] \overset{\text{lc}}{\leftrightarrow} [\Psi_{Ij} \Rightarrow H_j : \Psi_{Ej}]$.

Since this theorem holds for any two fragments in the program, we see that all fragments can be linked type-safely. Thus we have shown that Property 2.2 holds for CMOD.

One limitation of our proof strategy is that no named types t may appear in interfaces Ψ of MTAL$_0$ object files, only ground types. This limitation is reflected in our formalization in the premise $N \vdash H$ of the [COMPILE] rule—the judgment states that H must be well-formed under interface N, which will fail if N mentions any type names t. The full MTAL object file format supports type names, but its well-formedness rules are (unnecessarily) too restrictive to support CMOD [22]. Rather than attempt to fix MTAL, which is not our research focus, we claim that we can always replace type names t with their concrete definitions τ before applying [COMPILE]—by Rule 2, there is exactly one definition of each type name, making this replacement well-defined. Moreover, the lack of type names in object files does not impact our information hiding results, since Theorem 3.6 refers to a file's accumulator, not its compiled result.

IV. IMPLEMENTATION

We have implemented CMOD for the full C language [4]. We begin by describing how we enforce CMOD's rules, and then argue why we believe our implementation is sound, despite the increased complexity of C relative to our formal language.

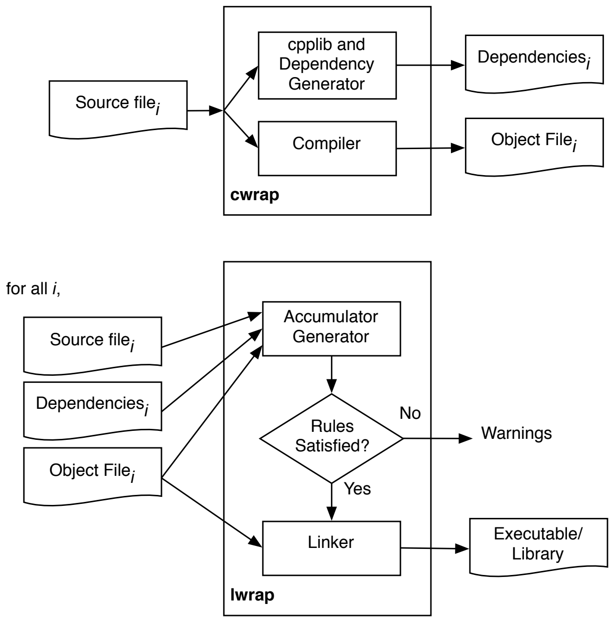

Fig. 11. CMOD architecture.

A. *Enforcing* CMOD*'s rules*

To use CMOD, programmers simply redirect their path to use special versions of the standard executables `gcc`, `ld`, and `ar`. Our versions of these programs determine whether a file is being compiled or linked, and then redirect execution to wrappers `cwrap` for compilation and `lwrap` for linking.

`cwrap` records a file f's macro dependencies observed during preprocessing (the $\mathcal{U}$ and $\mathcal{C}$ parts of the accumulator), along with the environment $\mathcal{E}(f)$ in which the file was compiled. To gather the dependencies, `cwrap` preprocesses the file using a modified `cpplib`, part of the `gcc` version 3 distribution, which invokes callbacks on various preprocessor events. The preprocessing environment $\mathcal{E}(f)$ consists the initial set of defined macros (`-D` arguments, plus `-O`, which sets some macros), as well as the specified include paths (`-I` arguments), needed by `lwrap` to re-preprocess the files. All of this information, along with the absolute path and timestamp information of each included header file, is stored in a *dependency* (`.D`) file used later to check CMOD's rules. After generating the `.D` file, `cwrap` runs `gcc` to generate the regular object file. Ideally, the dependency information would be embedded in the object file itself, but we leave this step to future work.

`lwrap` checks CMOD's rules on the given object files before passing them on to the linker. From rule [ALL] (Figure 9) we can see that to enforce Rules 1 and 2, all of the fragments that make up a program must be available so they be considered pair-wise. Thus it is natural to enforce these rules at link-time. While Rule 3 could be checked at compile-time, we find it simpler to check it at link time along with the other rules.

`lwrap` begins by attempting to synthesize a single global environment Δ, which according to [ALL], is used to check each of the rules. This environment is constructed from the environments Δ_f used to compile each file (as recorded in the `.D` files). In particular, for each macro name in the initial environment, `lwrap` checks that all files that use the macro agree on its setting in their initial environments. If so, `lwrap` adds the setting to the global environment, and otherwise `lwrap` aborts. If `lwrap` succeeds at creating this global environment, then Rule 4 is satisfied.

Given this global environment, `lwrap` checks Rules 1–3. For Rule 1, `lwrap` extracts symbol names directly from ELF object files and finds pairs of files such that one imports a symbol the other exports. `lwrap` then checks that both files import a common header file (determined by looking in the `.D` files) that declares the symbol. We use `ctags` [8] to compute the set of symbols declared in a header file, and we use the recorded timestamps for the headers to make sure they have not been modified since they were initially imported into the source files.

For Rule 2, `lwrap` preprocesses the source files corresponding to the linked object files and then runs `ctags` on the output of the preprocessor, producing a list of *(type name, file owner)* pairs. `lwrap` preprocesses each file in the global environment and with `#include` lines removed, so that type definitions listed by `ctags` are owned by the source file. With these results and the `ctags` information already computed for header files, `lwrap` combines and sorts the lists of pairs according to the type name. Then, using a linear pass over this sorted set of pairs, we flag definitions with multiple distinct owners.

This implementation approach requires that source `.c` files are needed at link time. This is problematic for libraries, which are not usually distributed with their sources. However, this problem can be remedied by gathering the `ctags` information when each file is *compiled*, and storing that in the `.D` file. Since Rule 4 checks that files were compiled in a consistent environment, we can be sure that the compile-time-generated `ctags` information would be consistent for all linked files. We leave such a change (along with the embedding of `.D` information into the `.o` files themselves) to future work.

Rule 3 imposes three requirements on interfaces: vertical independence, nulled duplicate inclusion, and non-recursive inclusion. As mentioned earlier, our implementation heuristically assumes that included files ending in `.h` are interfaces (included via import in the formalism), while other included files are not (included via inline in the formalism). To check vertical independence, `lwrap` uses `cpplib` to preprocess each source file in the global environment, tracking the macros that are changed and used. Whenever an interface file h is included, `lwrap` records the macros that are changed and used within h. When `lwrap` reaches the end of h, it checks that the set of macros changed (used) before h do not intersect the set of macros used (changed) in h.

The programmer may relax the vertical independence requirement for a `config.h` file specified in an environment variable. `lwrap` ensures that `config.h` is included first in every linked source file, so that it acts as an extension to the initial macro environment. For similar reasons, files `#include`d from within `config.h` are treated as inlined rather than imported; such files should only include configuration data, not interfaces that define the program's modular structure.

Checking that duplicate interface inclusions are nulled is straightforward. To optimize preprocessing time, `cpplib` already identifies the `#ifndef` pattern and notes the name of the macro used. CMOD checks that each processed header file uses this pattern, and confirms that the macro is never `#undef`ined prior to any subsequent re-inclusion. The latter check ensures that *all* duplicate inclusions of a header are nulled out. Finally, `lwrap` emits a warning if it encounters a recursive inclusion while processing each header. `cpplib` maintains a preprocessing stack (modeled by the end h markers in the formalism), so we simply check no file about to be included is present on the stack.

If an object has no `.D` file, as is (currently) the case with the system libraries, the object is precluded from CMOD's consideration. In particular, the enforcement of Rules 2–4 simply skips objects that have no `.D` information, while for Rule 1, in the case that an object file imports a symbol g defined in a library o with no `.D` file, CMOD skips consideration of that symbol.[6]

Since all rule checks occur at link-time, they can create a noticeable pause for a large program (as shown in the performance results in the next section). One way to reduce this pause would be

[6]Assuming we know which headers belong to which libraries, we could do slightly better by checking that the importing source file `#include`d *some* library header that declares g.

to judiciously cache relevant information from when the rules were last checked. For example, if an object file o has not changed and there is no reason to have recompiled it, e.g., due to a changed header or source file, then Rule 3 need not be rechecked. Moreover, if none of the object files from which o last imported its symbols have changed and these files are still linked with o, then assuming that the files are up-to-date CMOD need not recheck Rules 1 and 2 involving o.

Another possibility is to add compile-time well-formedness checks on files to reduce the cost of link-time checks. For example, we could (a) forbid declarations of non-local symbols in `.c` files, (b) forbid declarations of the same symbol in different `.h` files, and (c) require that each source file include exactly one header that declares the type of each symbol it exports. These checks should be sufficient to ensure that Rule 1 holds, and of them only (b) needs to be checked at link-time. The cost is that the checks are more restrictive than Rule 1 on its own. We are in the process of modifying our implementation to explore some of these ideas.

B. Soundness for Full C

The full C language contains many features not included in our formalism, and in this subsection, we argue that our implementation remains sound even in their presence.

The most significant difference between our formalism and C is that the full C preprocessor includes several additional directives, such as conditionals `#if` and `#ifndef`, token concatenation `##`, and macro substitution (e.g., `#define FOO(x) (x+1)`). Moreover, preprocessor commands in C may occur at arbitrary syntactic positions. Put together, these features would be extremely hard to add to our formal system. Nevertheless, we do not believe they affect the soundness of our implementation.

We can think of each header as a function whose input is a list of macro definitions and whose output is the preprocessed program text and a list of new macro definitions. Thus a header file's output is only affected by the definitions of macros it uses. In our formalism, a macro is used when it is changed or tested ([DEF], [UNDEF], [IFDEF+], and [IFDEF-]). Our implementation extends this idea by also counting macro references in other conditionals and macro substitutions as uses, and by counting non-boolean macro definitions as both changes and uses.

Thus, despite the complexity of the full C preprocessor, we can still track the "input" and "output" macros of a header. Moreover, it is also easy to extract the necessary type and declaration information to check the rules, because the rules, and our implementation, operate on the *preprocessed* files (for example, [RULE 1] preprocesses each fragment and the header file that contains the declaration). Thus, in both cases, [RULE 3] and [RULE 4] ensure consistent interpretation of header files, and therefore [RULE 1] and [RULE 2] correctly enforce information hiding and type-safe linking.

Another difference between our formalism and C is that our core language is lambda calculus. Since our focus is on linking and modularity, using lambda calculus is sufficient to model declarations, definitions, and variable references. Lambda calculus is also strongly typed, while C is not, e.g., type safety in C can be circumvented by unsafe casts. Thus our type safe-linking guarantee can be viewed as extending whatever type safety might be expected for a single C module to that of the entire program (as indicated in the definition of Property 2.2).

Finally, our formalism differs from C in its use of import and inline in place of C's `#include`. Our implementation checks that uses of `#include` match the semantics of one of these two directives. In particular, whenever a `.h` file is `#included` we ensure it uses the `#ifndef` pattern and that it never recursively includes itself, matching the semantics of import.

V. EXPERIMENTAL RESULTS

We applied CMOD to a variety of open source projects, with the goal of measuring how well they conform to CMOD's rules, and to determine whether rule violations are indeed problematic. We chose 30 open source projects of varying sizes (1.3k–165.1k lines of code), varying usage and stages of development (e.g., `xinetd`, `flex`, `gawk`, `bison`, `sendmail` and others are mature and widely used, while `zebra`, `mtdaapd`, and `retawq` are newer and less used), and varying reuse of modules among targets (`rcs`, `bc`, `gawk`, and `m4` have low reuse, while `mt-daapd`, `bison` and `vsftpd` have higher reuse). We believe the range of projects we looked at captures a representative set of common coding practices. We ran CMOD on a dual-processor 2.80GHz Xeon machine with 3GB RAM running the Linux 2.4.21-40.ELsmp kernel. We used `gcc` 3.2.3, GNU `ld/ar` 2.14.90.0.4, and `ctags` 5.4.

To separate preprocessor from source-language issues, we ran CMOD on each benchmark twice, using the following procedure. From the first run, we tabulated the Rule 3 and Rule 4 violations. We also examined warnings about header files not using the `#ifndef` pattern and for any such header, we manually added the pattern and verified that compilation was not affected. There were no warnings of recursive header inclusion, except for `limits.h` from the standard library, which as mentioned earlier is safe. We then fixed the Rule 3 and Rule 4 violations and reran CMOD to gather the Rule 1 and 2 violations.

A. Rule Violations

Table I summarizes the rule violations reported by CMOD. The first group of columns describes the benchmarks. For each program, we indicate whether it has a `config.h` file and list the number of *build targets* (executables or libraries); non-comment, non-blank lines of code; and `.c` and `.h` files. In the numerical totals, we count each file once, even if it occurs in multiple targets. The next two groups of columns indicate the number of rule violations, both in total and split across several categories, which we discuss next.

In the table, a Rule 1 violation corresponds to a symbol name and pair of files such that the files import and export the symbol without a mediating header. A Rule 2 violation occurs for each type name that has multiple definitions. A Rule 3 violation corresponds to a pair of files such that a change and use of a macro causes a vertical dependence between the files. Lastly, a Rule 4 violation corresponds to an object file compiled in an environment that is incompatible with the rest of the project. In the rule violation counts, we have not pruned duplicate violations for the same source in different targets. Any false positives due to inaccuracies in our implementation are listed in parentheses.

We believe most of the genuine rule violations constitute bad practice. In particular, they can complicate reasoning about the code, make future maintenance more difficult, and lead to bugs. We discuss each category of rule violation below.

Rule 1: We found 1970 total Rule 1 violations, which we break down further into three categories. The first category (C1, 1161 times) corresponds to cases where neither the client nor the provider include a header that declares a given symbol (i.e., the client "imports" a symbol using a local `extern`). As discussed in Section V-B, all violations in this category arguably violate information hiding.

The next two categories correspond to cases in which there does exist a header with a declaration of the symbol, but only the client (C2, 292 times) or only the provider (C3, 517 times) includes the header. We consider these Rule 1 violations to be dangerous because they permit a provider and client to disagree on the type of a symbol without generating a compile-time error (as discussed in Section II-B). However, they are not information hiding violations,

Program	Tgts	kLoC	`.c`	`.h`	Total Rule Violations				Rule Violations by Category									Prop. Viol.	
					Rule 1	Rule 2	Rule 3	Rule 4	Rule 1			Rule 2			Rule 3			Inf. Hid.	Typ.
									C1	C2	C3	C4	C5	C6	C7	C8	C9		
spell-1.0	1	1.3	4	3	2	-	-	-	2	-	-	-	-	-	-	-	-	2	-
time-1.7*	1	1.4	6	5	3	-	-	-	3	-	-	-	-	-	-	-	-	3	1
which-2.16*	2	2.0	6	4	4	-	-	-	4	-	-	-	-	-	-	-	-	4	-
jgraph-8.3	1	4.2	9	3	56	-	-	-	54	2	-	-	-	-	-	-	-	54	-
gzip-1.2.4	1	5.2	14	6	2	-	1	-	2	-	-	-	-	-	1	-	-	2	-
m4-1.4.4*	2	9.8	19	5	4	1	1	-	2	1	1	1	-	-	-	1	-	2	3
bc-1.06*	3	10.0	19	12	8	1(1)	3	-	4	1	3	-	-	1	3	-	-	4	1
gnuchess-5.07	1	12.0	32	9	2	-	-	-	2	-	-	-	-	-	-	-	-	2	-
vsftpd-2.0.3	1	11.6	34	41	4	-	9	-	-	4	-	-	-	-	9	-	-	-	-
rcs-5.7*	9	12.3	25	4	-	1	2	-	-	-	-	-	1	-	-	-	-	-	-
sed-4.1*	1	14.3	10	16	1	-	2	-	1	-	-	-	-	-	2	-	-	1	-
nano-2.0.3	1	14.5	15	3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
less-382*	3	14.6	33	8	197	-	-	-	197	-	-	-	-	-	-	-	-	197	6
flex-2.5.4*	2	16.4	22	10	5	6	-	-	3	-	2	6	-	-	-	-	-	3	-
xinetd-2.3.14*	8	16.3	60	65	10	4	-	-	3	6	1	3	1	-	-	-	-	3	-
mt-daapd-0.2.4	1	17.8	23	24	16	1	-	-	6	7	3	-	-	1	-	-	-	6	-
make-3.81*	1	18.3	24	12	23	-	-	-	23	-	-	-	-	-	-	-	-	23	-
retawq-0.2.6c*	1	21.2	5	9	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
bison-2.3*	3	20.8	59	68	3	17	8	2	2	1	-	2	-	15	8	-	-	2	-
wget-1.9*	1	21.6	30	24	21	-	19	-	17	4	-	-	-	-	19	-	-	17	-
fileutils-4.1*	23	29.4	87	60	583	2	33	-	128	155	300	-	2	-	32	1	-	128	12
gawk-3.1.5*	2	30.5	22	20	41	-	4	-	38	3	-	-	-	-	4	-	-	38	-
apache-2.3.1*	13	31.7	53	38	87	-	29	-	87	-	-	-	-	-	27	-	2	87	-
screen-4.0.2*	1	37.6	32	18	330	-	9	-	328	2	-	-	-	-	8	1	-	328	-
openssh-4.2p1*	13	52.8	157	83	68(38)	5	3	-	63	3	2	3	2	-	3	-	-	63	-
gnuplot-4.0.0*	4	80.4	49	110	x	x	315	x	x	x	x	x	x	x	x	x	x	x	-
zebra-0.94*	8	107.4	111	84	143	-	3	-	64	63	16	-	-	-	1	2	-	64	5
mc-4.5.55*	10	121.1	158	408	137	11	239	-	82	30	25	10	1	-	239	-	-	82	-
bind-9.3.4*	35	156.3	233	286	20	3	529	58	8	10	2	1	1	1	32	-	497	8	2
sendmail-8.14.0*	9	165.1	127	50	200	2	35	-	38	-	162	2	-	-	25	10	-	38	-
Total	162	1057.9	1478	1488	1970	54	1244	60	1161	292	517	28	8	18	413	15	499	1161	30

*Has `config.h` file. ^{x}gnuplot violations not resolved

TABLE I

EXPERIMENTAL RESULTS: RULE AND PROPERTY VIOLATIONS because the symbol appeared in a header file and thus was clearly meant to be exported.

Rule 2: Rule 2 violations are due to multiple definitions of the same type name, which can lead to type mismatches and information hiding violations. Most violations (C4, 28 times) occurred because the same type definition is duplicated in several files. As with most code duplication, this is dangerous because the programmer must remember to update all definitions when changing the type.

We also found some violations that may be considered safe. In several cases (C5, 8 times), the same type *name* is reused in different files. In these cases each definition is local to a single file, so the code is safe. Allowing a `static` notion for types would eliminate these violations. In the remaining cases (C6, 18 times) type definitions were replicated by automatic code generation, which essentially eliminates the danger of using incompatible definitions. This is a pattern that CMOD does not recognize.

Rule 3: Rule 3 violations make it harder to reason about headers in isolation. There are a total of 428 Rule 3 violations that we think are bad practice. 413 (C7) are due to vertical dependencies between headers, which we have already argued are undesirable, and 15 (C8) occur because the same macro is `#define`d in two different header files. In these cases the macros are actually defined to be the same—the code appeared to have been duplicated between the files, which makes maintenance harder.

The remaining Rule 3 violations (C9, 499 times) are safe practices that CMOD does not recognize as such. All such violations for `bind` occur because it uses a `.h` file that contains only source code, and that code is parameterized by macros defined earlier. This file is clearly not intended to be an interface file, and these warnings are easy to eliminate by renaming the file to end in `.c`, so that CMOD treats it as inlined rather than imported. The last two violations in this category are from `apache`, which auto-generates headers that are vertically dependent on the source files in which they appear. Because of the auto-generation this is safe.

We did not discover any cases of interface files not properly using the `#ifndef` pattern.

One program, `gnuplot`, has a very large number of vertical dependencies. `gnuplot` uses special `.trm` files as both headers and sources, depending on CPP directives. Effectively compilation is structured to have preprocessing evaluate files to sources or headers depending on the context, something that runs contrary CMOD's assumption that `.c` files are modules and `.h` files are interfaces. Because of this mismatch, we did not attempt to fix the violations, and thus we do not measure Rule 1 or 2 violations for `gnuplot`, nor do we include them in the totals.

Rule 4: All of the Rule 4 violations (60 times) are due to project libraries that are linked with files compiled in incompatible macro environments. In these cases, there were string-valued macros that were passed in as command-line arguments, and were different for different targets. This is a harmless practice, and could be addressed by relaxing Rule 4 to only hold for macros used in header files, since only headers need be consistently interpreted.

False Positives: CMOD reported a total of 39 false positives, meaning that CMOD issued a warning but the code does not actually violate the rule. All false positives were due to `ctags`. The 38 cases for Rule 1 occurred because `ctags` could not parse some complex code in the `openssl/evp.h` system header. The 1 case for Rule 2 occurred because `bc` contains some code that `ctags` also cannot parse.

B. Property Violations

Of those rule violations we consider bad practice, some directly compromise Properties 2.1 (Information Hiding) and 2.2 (Type-Safe Linking). The last two columns in Table I measure how often this occurs in our benchmarks.

Information hiding violations degrade a program's modular structure, complicating maintenance and potentially leading to defects. To determine what constitutes an information hiding violation, we need to know the programmer's intended policy. Since this is not explicitly documented in the program, here we assume that header files define the policy. In particular, following Property 2.1, we consider as public any symbol mentioned in a header file, and any type defined in a header file. Likewise, we consider as private any symbol never mentioned in a header, and any type mentioned in a header file but defined in a source file.

By this measure, some Rule 1 and 2 violations are not information hiding errors, e.g., when a `.c` file fails to include its own header(s), or when an identical type definition appears in several headers. Information hiding violations by our metric constitute roughly 59% of the Rule 1 violations. There were no Rule 2 violations that showed information hiding problems.

There were a total of 30 link-time type errors in our benchmarks. All of the errors were due to Rule 1 violations in which a client locally declared a prototype and got its type wrong. The most striking type errors were found in `zebra`. Clients incorrectly defined prototypes for four functions, in two cases using the wrong return type and in two cases listing too few arguments.

No header is defined to include prototypes for these four functions, and hence these were also information hiding violations. Ironically, in the cases where the return type was wrong, the client code even included a comment describing where the original definition is from—yet the types in the local declaration were still incorrect.

C. Required Changes

We designed CMOD to enforce modular properties while remaining backward compatible. To evaluate the latter, we measured the effort required to make a program CMOD-compliant. Table II lists the changes required to fix rule violations and presents performance numbers. For each project the first set of columns list the number of additions (+) and deletions (-) of files (f) and lines of code (no unit) required to eliminate the CMOD warnings. One file change corresponds to manually inlining or deleting a whole file, usually because code was split across files to no apparent advantage. The last set of columns list the build times without and with CMOD and the total slowdown, computed as the ratio of CMOD's time over the regular build time.

We found it was generally easy to make a program comply with CMOD's rules, and fixing most violations required only straightforward changes. Although some of the numbers in Table II suggest we needed to change many source lines, high change counts are mostly do to search-and-replace operations applied to large code bases. Using the warnings reported by CMOD and general knowledge about C programming, we were able to fix most violations almost mechanically, with little time or effort.

Rule 1 violations could be fixed in a variety of ways depending on the category they fell in. We fixed C1 violations, in which symbols are imported but not declared in a shared header, by inserting a declaration in an appropriate header file. Two of these violations could not be fixed because they are due to assembler sources that define exported symbols; these files cannot `#include` a header that declares them, since the code is not written in C. We fixed the remaining violations, in which a header containing a symbol declaration is not included by the provider (C2) or client (C3), by simply adding the missing `#include`.

We fixed Rule 2 violations due to duplicate definitions (C4 and C6) by consolidating the definitions into an appropriate file. For two programs, `bc` and `mt-daapd`, we did not attempt to fix the violations because they were in auto-generated code. Since C does not provide a notion

		Changes Required†								Build Time		
Program	kLoC	Rule 1		Rule 2		Rule 3		Rule 4		Stock(s)	CMOD(s)	Slowdown
		+	-	+	-	+	-	+	-			
spell-1.0	1.3	2	2	-	-	-	-	-	-	0.3	1.5	5.0
time-1.7	1.4	1f+10	4	-	-	-	-	-	-	0.4	2.6	6.5
which-2.16	2.0	5	2	-	-	-	-	-	-	0.6	3.0	5.0
jgraph-8.3	4.2	87	40	-	-	-	-	-	-	1.2	2.9	2.4
gzip-1.2.4	5.2	2	-	-	-	2	7	-	-	1.2	4.5	3.8
m4-1.4.4	9.8	11	2	-	1	60	60	-	-	3.9	9.1	2.3
bc-1.06	10.0	3	-	-	-	1f,7	7	-	-	2.9	9.5	3.3
gnuchess-5.07	12.0	3	-	-	-	-	-	-	-	6.5	17.8	2.7
vsftpd-2.0.3	11.6	1	-	-	-	10	20	-	-	3.1	9.7	3.1
rcs-5.7	12.3	-	-	-	-	2	3	-	-	3.8	22.6	5.9
sed-4.1	14.3	2	-	-	-	5	3	-	-	3.8	9.0	2.4
nano-2.0.3	14.5	-	-	-	-	-	-	-	-	4.8	12.7	2.6
less-382	14.6	162	131	-	-	-	-	-	-	4.6	14.7	3.2
flex-2.5.4	16.4	4	-	-	1f	-	-	-	-	5.7	17.1	3.0
xinetd-2.3.14	16.3	6	-	10	12	-	-	-	-	7.4	29.7	4.0
mt-daapd-0.2.4	17.8	7	-	-	-	-	-	-	-	6.9	16.5	2.4
make-3.81	18.3	45	7	-	-	-	-	-	-	6.3	17.7	2.8
retawq-0.2.6c	21.2	-	-	-	-	-	-	-	-	5.8	9.2	1.6
bison-2.3	20.8	2	-	4	4	1f+15	134	1	1	11.0	34.5	3.1
wget-1.9	21.6	46	17	-	-	-	-	-	-	7.0	17.5	2.5
fileutils-4.1	29.4	1f+1	-	-	-	21	6	-	-	13.4	120.2	9.0
gawk-3.1.5	30.5	1f+26	17	-	-	1f+6	16	-	-	12.0	26.5	2.2
apache-2.3.1	31.7	3f+193	84	-	-	35	24	-	-	6.7	46.8	7.0
screen-4.0.2	37.6	1f+7	2	-	-	40	28	-	-	13.3	27.5	2.1
openssh-4.2p1	52.8	70	9	52	54	1f,68	84	-	-	32.7	99.7	3.0
gnuplot-4.0.0	80.4	x	x	x	x	x	x	-	-	31.8	71.3	2.2
zebra-0.94	107.4	78	25	-	-	6	12	-	-	38.2	132.5	3.5
mc-4.5.55	121.1	1f+99	68	73	64	175	132	-	-	59.9	1072.5	17.9
bind-9.3.4	156.3	12	3	9	9	2f+453	198	362	69	81.4	158.7	1.9
sendmail-8.14.0	165.1	1f+10	3	5	10	2f+92	153	-	-	28.3	141.4	5.0

†Line or file (f) additions (+) and deletions (-)

TABLE II

EXPERIMENTAL RESULTS: CHANGES REQUIRED AND RUNNING TIMES

of a `static` type, we fixed instances of locally-scoped type name reuse (C5) by alpha renaming.

For Rule 3, vertical independence violations (C7) required various techniques to fix. In general, since CMOD's Rule 3 warnings report the macro that caused the dependency, the offending files, and the locations they were included, we found it easy to come up with fixes without looking at much code. Files that do not act as interfaces but that CMOD thinks are imported can be renamed or manually inlined. When a source file defines a macro that parameterizes a following header, we found it easiest to duplicate the header once for each time it was used. For a header parameterized by a boolean-valued macro, in the worst case we duplicated the header once for each setting of the macro. For macros that expand to strings, we replace the macro with its expansion inside the header. In cases where the dependency involved a macro that was used throughout the project, we moved the definition or file into `config.h`. As mentioned earlier, `gnuplot` relies on vertical dependencies that cannot be removed without fundamentally changing the design of the program, and so we did not fix those.

Duplicate macro definitions (C8) were eliminated by consolidating them into an appropriate header. Recall that the remaining violations (C9) correspond to safe practices. We fixed the warnings for `bind` by renaming a `.h` file to a `.c` file to cause it to be inlined. We did not attempt to fix the last two warnings, in `apache`, which are caused by auto-generated code.

Rule 4 violations involved conflicting compilation environments (`-D` flags) in `bind` and `bison`. To fix these, we examined the source code of these projects to determine whether the difference in macro environments was intentional. In both cases, the macros were defining string constants, and so we could fix the violations by moving these definitions into the source files.

D. Performance

Finally, the last three columns in Table II measure the time taken to build the program without and with CMOD, for the fixed versions of the projects. The times reported are the median of five trials. The current prototype of CMOD adds noticeable overhead to the compilation procedure: the average slowdown is 4.1 times, while the median slowdown is 3.1 times (with `mc` and `fileutils` being the major outliers). There are three main performance issues in our current implementation, all of which should be addressable with more engineering effort. First, large projects tend to be built around libraries. CMOD performs rule checking at link time on all linked objects, including libraries—and thus if the same libraries are reused in many different targets, their internal files are repeatedly checked. The repeated checking of libraries is the main source of overhead for `fileutils` and `mc`. Second, programs tend to include many headers that they do not actually need. This significantly increases the sizes of the accumulators (Section III-C) that CMOD computes, which makes operations involving those accumulators slower. Finally, much of the overhead derives from the prototype nature of the implementation, which combines scripts with native code programs, and is at times indiscriminate with disk usage and recomputations to make things simpler. We leave as future work the task of optimizing the implementation, e.g., by using memoization and caching (as discussed in the prior section), reducing disk accesses, and having fewer native calls to reduce interprocess communication.

VI. Related Work

As stated in the introduction, although many experts recommend using `.h` files as module interfaces and `.c` files as module implementations [2], [16], [17], [18], [20], the details vary somewhat and are not sufficient to enforce soundness. King presents the core idea that header files should include declarations, and that both clients and implementations should `#include` the header [18]. McConnell recommends always having public and private headers for modules [20], and mentions using a single public header for a group of implementations; neither idea is discussed in most sources. The Indian Hill style guide rather confusingly recommends both that "header files should not be nested" (i.e., recommends vertical dependencies, something we think is bad practice), and recommends using `#ifndef` to prevent multiple inclusions, which should never happen if there are no nested headers. None of these publications, nor any other publication we could find, discussed sufficient requirements to ensure information hiding and type-safe linking, leading us to believe that the subtleties are not widely known.

There is a large design space of module systems [26], which are part of many modern languages such as ML, Haskell, Ada, and Modula-3. In common with CMOD, these languages support information hiding via transparent and abstract types, and multiple interfaces per implementation. They ensure type-safe linking, and most (but not all) support separate compilation. They also provide several useful mechanisms not supported by CMOD, due to its focus on backward compatibility.

First, ML-like languages support functors, which can be instantiated several times in the same program. As discussed in Section II-C, CMOD supports program-wide parameterization (e.g., via the initial environment and optionally `config.h`), and a form of per-module parameterization by textually including code (modeled by the inline directive in our formal account).

Second, most module systems also support hierarchical namespace management. Since CMOD builds on existing C programming practice, it inherits C's global namespace, with limited support for symbol hiding via `static`, and no support for hiding type names. C++ namespaces address this limitation to some extent, and we believe they could safely coexist with CMOD.

Lastly, in CMOD and many module systems, linking occurs implicitly by matching the names of imports and exports. Some systems, however, express linking explicitly, for a greater degree of abstraction and reuse. Some examples are the Configuration Manager (CM) [1] for Standard ML and Units [11] for Scheme. There are also explicit linking systems for C and/or C++, including Knit [28] (which is based on Units), Koala [33], and Click [21]. Microsoft's Component Object Technologies (COM) model [5] provides similar facilities to construct dynamically linked libraries (DLLs). The C-based systems assume that the basic C module convention is used correctly and build on top of it, and so CMOD may be viewed as complementary.

Vandevoorde [34] proposes a module system for C++. The proposed module system adds module import and export syntax to the language, rather than using the preprocessor. Thus macro interference (i.e., vertical dependencies) between modules is eliminated. Vandevoorde's system also addresses some other issues, such as providing stronger information hiding than even private class members support and improving compiler performance. However, using this new module system requires modifying source code, whereas CMOD works with existing C programs and is provably sound.

Some systems for C and C++ aim to support type-safe linking but not information hiding. C++ compilers embed type information in symbol names during compilation, a practice called "name mangling." Although designed to support overloading, name mangling can also enforce link-time type safety. Since names include type information, when a client and provider agree on a name, they also agree on types. This is not always reliable, however, since mangled `struct` types do not include field information, which could therefore disagree. CIL [24] is a parsing toolkit for C that can combine several C sources into a single file. In so doing, it complains if it finds that two files disagree on the definition of a type or symbol. It would find all of the type errors that we discovered in our experiments, but none of the information hiding violations.

Finally, a number of researchers have studied the C preprocessor, though not as a means to enforce modularity. Favre [10] proposes a denotational semantics for CPP. Several researchers recommend curtailing or even eliminating the C preprocessor, due to its complexity [9], [19]. Lastly, a number of tools check for erroneous or questionable uses of cpp directives, including `lint` [15], PC-lint [25], and Check [29]. The detected bug patterns are fairly localized and generally concern problematic macro expansions.

VII. Conclusions

We have described CMOD, a module system for C that ensures type-safe linking and information hiding while maintaining compatibility with existing practice. CMOD enforces a set of four rules. At a high level, Rule 1 makes header files equivalent to regular modular interfaces; Rule 2 checks for consistent use of type names and type abstraction; and Rules 3 and 4 control preprocessor interactions. We showed formally that these rules in combination with the C compiler form a sound module system that supports information hiding and ensures type-safe linking. Our experiments show that in practice, violations of our rules reveal dangerous coding idioms, violations of information hiding, and type errors. Fortunately, we found that for most programs, rule violations are rare and can be fixed fairly easily. Thus CMOD brings the benefits of modular programming to C while still being practical for legacy systems.

Put back in final version.

Acknowledgments

The authors would like to thank Bhargav Kanagal (for contributions to the proof), Matthew Fluet, Nikhil Swamy, Greg Morrisett, Trevor Jim, Neal Glew, Iulian Neamtiu, Stephen Tse, Emery Berger, Mike Furr, and the anonymous reviewers for their suggestions on earlier drafts of this paper. This research was supported in part by NSF CCF-0430118 and NSF grant CCF-0346989.

References

[1] M. Blume and A. W. Appel. Hierarchical modularity. *TOPLAS*, 21(4):813–847, 1999.

[2] L. Cannon, R. Elliott, L. Kirchoff, J. Miller, R. Mitze, E. Schan, N. Whittington, H. Spencer, D. Keppel, and M. Brader. *Recommended C Style and Coding Standards*. sixth edition, 1990.

[3] L. Cardelli. Program fragments, linking, and modularization. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pages 266–277, New York, NY, USA, 1997. ACM Press.

[4] `http://www.cs.umd.edu/projects/PL/CMod/`.

[5] COM: Component object model technologies. `http://www.microsoft.com/com/default.mspx`.

[6] J. Condit, M. Harren, Z. Anderson, D. Gay, and G. C. Necula. Dependent types for low-level programming. In *Proceedings of European Symposium on Programming (ESOP)*, 2007.

[7] B. Cox and A. Novobilski. *Object Oriented Programming: An Evolutionary Approach*. Addison-Wesley, 1991.

[8] Exhuberant ctags. `http://ctags.sourceforge.net/`.

[9] M. D. Ernst, G. J. Badros, and D. Notkin. An empirical analysis of C preprocessor use. *IEEE Transactiosn on Software Engineering*, 28(12), 2002.

[10] J.-M. Favre. CPP Denotational Semantics. In *SCAM*, 2003.

[11] M. Flatt and M. Felleisen. Units: Cool modules for HOT languages. In *Proceedings of SIGPLAN International Conference on Programming Language Design and Implementation*, pages 236–248. ACM, June 1998.

[12] N. Glew and G. Morrisett. Type-safe linking and modular assembly language. In *POPL*, 1999.

[13] Once-only headers - the C preprocessor. `http://gcc.gnu.org/onlinedocs/cpp/Once_002dOnly-Headers.html#Once_002dOnly-Headers`.

[14] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, 2002.

[15] S. Johnson. Lint, a C program checker. Technical Report 65, Bell Labs, Murray Hill, N.J., Sept. 1977.

[16] B. W. Kernighan and R. Pike. *The Practice of Programming*. Addison-Wesley Professional, 1999.

[17] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, 2nd edition, 1988.

[18] K. N. King. *C Programming: A Modern Approach*. W. W. Norton & Company, Inc., 1996.

[19] B. McCloskey and E. Brewer. ASTEC: a new approach to refactoring C. In *FSE*, 2005.

[20] S. McConnell. *Code Complete*. Microsoft Press, 1993.

[21] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The Click modular router. In *SOSP*, 1999.

[22] G. Morrisett. Personal communication, July 2006.

[23] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: Type-Safe Retrofitting of Legacy Software. *TOPLAS*, 27(3), May 2005.

[24] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *CC*, pages 213–228, 2002.

[25] PC-lint/FlexeLint. `http://www.gimpel.com/lintinfo.htm`, 1999. Product of Gimpel Software.

[26] B. C. Pierce, editor. *Advanced Topics in Types and Programming Languages*. MIT Press, 2005.

[27] Precompiled headers - using the GNU Compiler Collection (GCC). `http://gcc.gnu.org/onlinedocs/gcc/Precompiled-Headers.html`.

[28] A. Reid, M. Flatt, L. Stoller, J. Lepreau, and E. Eide. Knit: Component composition for systems software. In *OSDI*, 2000.

[29] D. Spuler and A. Sajeev. Static detection of preprocessor macro errors in C. Technical Report 92/7, James Cook University, Townsville, Australia, 1992.

[30] S. Srivastava, M. Hicks, and J. S. Foster. Appendix to CMod: Modular Information Hiding and Type-Safe Linking for C. Technical Report CS-TR-4874, University of Maryland, College Park, 2007.

[31] S. Srivastava, M. Hicks, and J. S. Foster. Modular Information Hiding and Type-Safe Linking for C. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Language Design and Implementation*, pages 3–13, Nice, France, Jan. 2007.

[32] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.

[33] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala component model for consumer electronics software. *IEEE Software*, 2000.

[34] D. Vandevoorde. Modules in C++. Technical Report N2073=06-0143, JTC1/SC22/WG21 – The C++ Standards Committee, Sept. 2006. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2073.pdf`.

PLACE PHOTO HERE

Saurabh Srivastava is a doctoral candidate in the Department of Computer Science at the University of Maryland, College Park. His research interests are in program analysis and specifically formal verification and specification inference. He is interested in tools and techniques for proving program correctness and finding faults. In the context of the CMOD project he has explored techniques that retrofit a module system over C.

PLACE PHOTO HERE

Michael Hicks is an Associate Professor in the Department of Computer Science and the University of Maryland Institute for Advanced Computer Studies (UMIACS), at the University of Maryland, College Park. His research focuses on developing programming language technology–ranging from static and dynamic analysis tools applied existing programming languages, to new domain-specific languages—to help programmers build software that is reliable, available, and secure.

PLACE PHOTO HERE

Jeffrey S. Foster is an Assistant Professor in the Department of Computer Science and the University of Maryland Institute for Advanced Computer Studies (UMIACS) at the University of Maryland, College Park. His research aims to give programmers practical new tools to help improve the quality and security of their programs. His research interests include programming languages, program analysis, constraint-based analysis, and type systems.

PLACE PHOTO HERE

Patrick Jenkins graduated from the University of Maryland, College Park in 2006 with degrees in Computer Science and Mathematics. He currently works for a technology startup that secures funding for non-profits and charities.

h1.h

```
1  #ifdef A
2  #define B
3  #endif
```

h2.h

```
10 #define A
11 #include "h1.h"
12 #ifdef B
13 extern int x;
14 #endif
```

a.c

```
15 #include "h1.h"
16 #include "h2.h"
17 extern float x;
```

b.c

```
18 #include "h1.h"
19 #include "h2.h"
20 int x;
```

Fig. 12. Example showing need to forbid use-before-change (`ifndef` omitted for clarity)

APPENDIX

Recall that in Section III, the last hypothesis of rule [TRACE-INDEP] in Fig. 9 was somewhat surprising. To understand the need for this restriction, consider the code in Fig. 12. In this example, header `h1.h` defines `B` if `A` is already defined. Header `h2.h` defines `A` and then includes `h1.h`—thus if `h2.h` is preprocessed in isolation, then after line 11, both `A` and `B` are defined. Therefore the test on line 12 is true, and line 13 declares `x` to be an `int`.

However, consider what happens during preprocessing of `a.c`, on the right side of the figure. Here `h1.h` is included first, and since `A` is not defined, it has no effect; in particular, it does not define `B`. Then on line 16, we include `h2.h`, and in preprocessing that file, the inclusion on line 11 is skipped, because it is a duplicate (assume the `ifndef` pattern is present, though we have omitted it for clarity). Thus, since `B` is undefined, the declaration on line 13 does *not* occur. And therefore the declaration on line 17 succeeds at compile time, and in `a.c`, the variable `x` is a `float`. A similar thing happens in `b.c`, which compiles with no warnings and produces a file that assumes `x` is an `int`.

Thus we have a link-time type inconsistency. However, notice that [RULE 1] accepts this code, because `a.c` and `b.c` include a common header `h2.h`, and *in isolation*, `h2.h` declares the type of `x`. The problem here is that when included in `a.c` and `b.c`, `h2.h` does not actually produce any declarations, and so while it is consistently interpreted with respect to the inclusions that actually occur in the code, [RULE 1] additionally requires that a header be consistently interpreted also when preprocessed in isolation.

CMOD solves this problem with the last hypothesis of [TRACE-INDEP], which says that for two traces $\tilde{f}_1$ and $\tilde{f}_2$ to be independent, not only must changed macros in $\tilde{f}_1$ not be used in $\tilde{f}_2$, but used macros in $\tilde{f}_1$ must not be changed in $\tilde{f}_2$. It may be possible to eliminate this restriction by changing [SYM-DECL] to use the traces generated while preprocessing fragments, rather than preprocessing a header in isolation. However, as we stated earlier, it seems better to ensure header files are consistently interpreted everywhere as they are in the initial environment, to forbid confusing examples like Fig. 12.