CMOD: Modular Information Hiding and Type-Safe Linking for C*

Saurabh Srivastava, Michael Hicks, Jeffrey S. Foster
{saurabhs,mwh,jfoster}@cs.umd.edu

Contents

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, developing CMOD's rules revealed there are many subtleties in applying the basic pattern correctly. We have proven formally that CMOD's rules enforce both information hiding and type-safe linking. We evaluated CMOD on a number of benchmarks, and found that most programs obey CMOD's rules, or can be made to with minimal effort, while rule violations reveal brittle coding practices including numerous information hiding violations and occasional type errors.

*UMD, Tech Report CS-4816

1 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 still enforcing *type safety* across components. This combination allows modules to be safely written and reasoned about in isolation, enhancing the reliability of software [29].

While many modern languages define full-featured module systems (such as ML [3, 20], Haskell [14], Ada [1], and Modula-3 [10]), 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 implementations and use `.h` header files (containing type and data declarations) as interfaces. Textually including a `.h` file via the `#include` directive is akin to "importing" a module.

Many experts recommend using this basic pattern [2, 15, 16, 17, 19], but their recommendations are incomplete and, as it turns out, insufficient. To our knowledge, the basic pattern has not been previously developed to the point that proper information hiding and type safety are provable consequences. As a result, programmers may be unaware of (or ignore) the subtleties of using the pattern correctly, and thus may make mistakes (or cut corners), since the compiler and linker provide no enforcement. The result is the potential for 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 provides a sound module system for C by enforcing four rules that flesh out C's basic modularity pattern. In other words, CMOD aims to enable safe modular reasoning while matching existing programming practice as much as possible. We have proven formally that CMOD's four rules ensure that C programs obey information hiding policies implied by interfaces, and that programs are type safe at link time.[1] To our knowledge, CMOD is the first system to enforce both properties for standard C programs. Related approaches (Section 6) either require linguistic extensions (e.g., Knit [27] and Koala [30]) or enforce type safety but not information hiding (e.g., CIL [24] and C++ "name mangling").

To evaluate how well CMOD matches existing practice while still strengthening modular reasoning, we ran CMOD on a suite of programs cumulatively totaling 440K lines of code split across 1263 files. We found that most programs generally comply with CMOD's rules, and fixing the rule violations typically requires only minor changes. Rule violations revealed many information hiding errors, several typing errors, and many cases that, although not currently bugs, make programming mistakes more likely as the code evolves. These results suggest that CMOD can be applied to current software at relatively low cost while enhancing its safety and maintainability.

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

- We present a set of four rules that makes it sound to treat header files as interfaces and source files as implementations (Section 2). 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 [12].

- We give a precise, formal specification of our rules and prove that they are sound, meaning that programs that obey the rules follow the abstraction policies defined by interfaces and are type safe at link time (Section 3).

- We present our implementation, CMOD (Section 4), and describe the results of applying it to a set of benchmarks (Section 5). CMOD found numerous information hiding violations and several typing errors, among other brittle coding practices, and it was generally easy to bring code into compliance with CMOD.

[1] Throughout this paper, when we say *type safety*, we mean that types of shared symbols match across modules. C programmers can still violate type safety in other ways, e.g., by using casts. This could be addressed by using CCured [23] or Cyclone [12], which should combine seamlessly with CMOD.

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 }
```

Figure 1: Basic C Modules

2 Motivation and Informal Development

In most module systems, a module M consists of an *interface* M_I that declares exported values and types, and an *implementation* M_S that defines everything in M_I and may contain other, private definitions. Any component that wishes to use values in M_S relies only on M_I, and not on M_S. The compiler ensures that M_S *implements* M_I, meaning that it exports any types and symbols in the interface. 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 depend only on interfaces rather than particular implementations:

Property 2.1 (Information Hiding) *If M_S defines a symbol g, then other modules may only access g if it appears in M_I. If M_I declares an abstract type t, no module other than M_S 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 M_I, it can link against any module that implements that M_I, and M_S may safely be changed as long as it still implements M_I.

Second, linking a client of interface M_I with an implementation M_S of M_I must be type-safe:

Property 2.2 (Type-Safe Linking) *If module M_S implements M_I and module N_S is compiled to use M_I, then the result of linking M_S and N_S is type-safe.*

The goal of CMOD is to ensure that C modules obey these two properties. 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, 15, 16, 17, 19].

2.1 Basic Modules in C

Figure 1 shows a simple C program that follows the modularity 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 `#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 is both type-safe and properly hides information. Since both `main.c` and `bitmap.c` include the same header, the C compiler ensures that the types of `init` and `set` match across the files. Furthermore, `main.c` never refers to the symbol `private` and does not assume a definition for `struct BM` (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  /* bitmap.h not incl. */
5
6  struct BM { int data; };
7
8  /* inconsistent decl. */
9  void init(struct BM *map,
10           int val) { ... }
11 void set(struct BM *map,
12           int bit) { ... }
13 void private(void) { ... }
```

main.c

```
14 #include "bitmap.h"
15
16 /* bad symbol imports */
17 extern void private(void);
18
19 /* violating type abstr. */
20 struct BM { int *data; };
21
22 int main(void) {
23   struct BM bitmap;
24   init(&bitmap);
25   set(&bitmap);
26   private();
27   bitmap.data = ...;
28   ...
29 }
```

Figure 2: Violations of Rules 1 and 2

2.2 Header Files as Interfaces

One of the key principles illustrated in Figure 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 [17, 19]. CMOD ensures that linking 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 type of that symbol.*

The C compiler and linker do not enforce this rule, so programmers sometimes fail to use it in practice. Figure 2 illustrates some of the common ways the rule is violated, based on our experience (Section 5). One common violation is for a source file to fail to include its own header, which can lead to type errors. In Figure 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 17 declares that `private` is an external symbol, allowing it to be called on line 26 even though it is not mentioned in `bitmap.h`. This violates information hiding, preventing the author of `bitmap.c` from easily changing the type of, removing, or renaming this function. It may also violate type safety; i.e., 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, and in some cases this is because the symbol 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 the example). For example, the standard library header `stdio.h` often covers several source files. 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 [19]. In this case the source file would include both headers, while clients would include one or the other.

The last error in Figure 2 is in `main.c`, which violates the information hiding policy of `bitmap.h` by defining `struct BM` on line 20. In this case the violation results in a type error since the definitions on lines 6 and 20 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

bitmap.h

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

bitmap.c

```
8  #include "config.h"
9  #include "bitmap.h"
10
11 #ifdef COMPACT
12 /* defn's of
13       init (), set() */
14 #else
15 /* alt. defn's of
16       init (), set() */
17 #endif
```

config.h

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

main.c

```
26 #include "config.h"
27 #include "bitmap.h"
28
29 int main(void) {
30   struct BM *bmap;
31   init (bmap);
32   set(bmap, 1);
33   ...
34 }
```

```
gcc -c -D__BSD__ bitmap.c
gcc -c -D__BSD__ main.c
```

Figure 3: Using the Preprocessor for Configuration

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*, and only the module that implements the type's functions should know its definition. CMOD requires that a type have only one owner, thus forbidding the example in Figure 2:

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

Notice that this rule is again somewhat flexible, allowing a middle-ground between abstract and transparent types. In particular, this 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 imaging 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.

2.3 Preprocessing and Header Files

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

Consider the code shown in Figure 3, which modifies our example from Figure 1 to represent bitmaps in one of two ways (lines 1–5), depending on whether the `COMPACT` macro has been previously defined (line 21 or 23). 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). In general, we say that a file f_1 *depends on* file f_2 when f_1 uses a macro set by f_2. Here, `bitmap.h` depends on `config.h`.

Such preprocessor-based dependencies are very useful, since they allow programs to be configured for different circumstances. Unfortunately, they can unintentionally cause a header to be preprocessed differently depending on where it is included. In Figure 3, if we were to swap lines 8 and 9 but leave lines 26 and 27 alone, then `bitmap.c` and `main.c` may have different, incompatible definitions of `struct BM`. Thus, the preprocessor can undermine type safety and information hiding, even given Rules 1 and 2.

To solve this problem, we define two additional rules that aim to enforce the following principle:

Principle 2.3 (Consistent Interpretation) *Each header in the system must have a* consistent interpretation*, meaning that whenever modules linked together include a common header, the result of preprocessing the header is the same in both modules.*

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). The first new rule to enforce this principle is:

Rule 3 (Vertical Independence) *With the exception of a designated, initial* `config.h`*, header file inclusion must be vertically independent.*

We say two header files are *vertically dependent* if one depends on the other and both are `#include`d by the same source. In the example, `bitmap.h` is vertically dependent on `config.h`. Vertical dependencies are encouraged by some coding style guides [2], but we forbid them because 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 dependency 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 dependency in the example, we could convert it to a horizontal dependency 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 [11, 5] beginning on line 18, which causes any duplicate file inclusions to be completely ignored. Our semantics assumes no duplicate inclusion, and we check that it holds for our benchmarks.

Although we feel that vertical dependencies are bad practice in general, the headers in many large programs are vertically dependent on a `config.h` header. CMOD allows these dependencies as long as `config.h` is always included *first*. This ensures other included headers are consistently interpreted with respect to it. This is easy for the programmer to remember and for CMOD to check.

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 20–24 is not set within the file. Instead, it is either supplied by the system or set by a `-D` command-line argument to the compiler. 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 27) would produce different representations for type `struct BM`. We can prevent this by enforcing the following rule:

Rule 4 (Environment Compatibility) *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 undefined 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.

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

$m \in$ macro names $\quad g \in$ global var. names
$h \in$ file names $\quad t \in$ type names

Figure 4: Source Language

Rules 3 and 4 allow a program as a whole to be parameterized by `config.h` and the initial preprocessor environment. In essence, the program can be considered a very large functor [26]. Thus while CMOD allows individual headers to be parameterized, they must be consistently interpreted throughout the program. We have rarely found this to be a problem 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. We have found two exceptions in practice. The first is to support context-dependent information hiding by including or not including certain prototypes based on `#ifdef`s. While CMOD disallows this practice, one can use separate header files instead [19]. The second case is to `#include` a `.h` file containing parameterized code definitions (akin to a functor application) instead of using the file as an interface. This situation is rare, and so we do not handle it specially, though this may be an interesting future direction.

Note that while enforcing Principle 2.3 ensures headers are consistently interpreted, 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. Rule 2 ensures type definitions must always mean the same thing, but there is no such rule for symbols, which can be multiply-defined if declared `static`. Though it may be desirable to forbid dependencies on symbols, CMOD allows them, for two reasons. First, such dependencies do not impact type safety and information hiding. Second, extending CMOD to track such dependencies would add significant implementation complexity when compared to our current approach (Section 4), and in our experience, dependencies on symbols are rare. We leave such an implementation to future work.

3 Formal Development

In this section we formally present CMOD's rules and prove that they are sound. Our rules are defined in terms of the source language in Figure 4. In this language, a source program $\mathcal{P}$ consists of a list of fragments f, each of which represents a separately-compiled source file. Fragments are themselves made up of a list of statements s, which may be either preprocessor commands c or core language definitions d.

Our preprocessor commands mimic the C preprocessor. The command $\mathsf{include}\ h$ inserts the fragment contained in file h and then preprocesses it. In our semantics we assume we are given a mapping from include file names to fragments. The commands $\mathsf{def}\ m$ and $\mathsf{undef}\ m$ define and undefine, respectively, the preprocessor macro m from that point forward. In our semantics, macros may only be used as boolean flags. The conditional $\mathsf{ifdef}\ m\ \mathsf{then}\ f_1\ \mathsf{else}\ f_2$ processes f_1 if m is defined, and otherwise processes f_2. Notice that since each branch is a fragment, it may contain further preprocessor commands.

The C preprocessor includes additional features not found in our language, including macro substitution and conditional forms such as `#if` and `#ifndef`. The C language also allows preprocessor commands to

occur anywhere in the text of the program, whereas our language forbids preprocessor commands inside of definitions. In Section 3.5, we argue that these additional features do not affect soundness.

Turning to the core language, the definition $\mathsf{let}\ g : \tau = e$ binds the global name g to term e, which has type τ. Terms e are simply-typed lambda calculus expressions that may refer to local variables y or global variables g. We use the lambda calculus instead of C as our base language because it is type safe and illustrates all of the necessary issues. The command $\mathsf{extern}\ g : \tau$ declares the existence of global g of type τ. This form is used in header files to import a symbol. The command $\mathsf{let\ type}\ t = \tau$ defines a named type t to be an alias for τ, while $\mathsf{type}\ t$ merely declares that t may be used as a type name. We say that g and t are *defined* by $\mathsf{let}\ g$ and $\mathsf{let\ type}\ t = \tau$ while g and t are *declared* by $\mathsf{extern}\ g : \tau$ and $\mathsf{type}\ t$. Within a program we allow many declarations of a global variable or type name but only one definition. Note that to keep the rules simpler, we do not model `static` definitions.

3.1 Preprocessor Semantics

Following C, our source language has a two-stage operational semantics. For each fragment, the preprocessor executes all of the preprocessor commands, conceptually producing a fragment consisting only of core language definitions. These fragments are then compiled into object files, which are combined with linking, and then the entire program is evaluated using a standard semantics.

The four CMOD rules are based on the output of an instrumented preprocessor, shown in Figure 5. Rather than perform substitutions to generate a new fragment consisting only of definitions (which would be closer to the semantics of the actual C preprocessor), our semantics constructs an *accumulator* $\mathcal{A}$ that contains both the core language definitions and other information needed to enforce CMOD's rules. In particular, the preprocessor is defined as a relation among *states* of the form $\langle h; \mathcal{A}; \Delta; x \rangle$, where h names the file currently being preprocessed, $\mathcal{A}$ is the accumulator, Δ is the set of currently-defined macros, and x is either a fragment or a statement.

Each top-level fragment in the program is preprocessed separately. Preprocessing fragment f begins with an initial (possibly empty) set of defined macros Δ_f, which in practice is supplied on the command line when the compiler is invoked. Δ_f may differ from one fragment to another. The *accumulator* $\mathcal{A}$ is a tuple that tracks the preprocessor events that have occurred thus far. The core language program is encoded as three lists in the accumulator: 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 remainder of the accumulator consists of the sets of global variables that have been exported (E) by defining them with let, imported (I) by using them in code, and declared (D) by extern or let; the set of macros $\mathcal{C}$ that have possibly been changed (defined or undefined); the set of macros $\mathcal{U}$ whose value has been tested; the set of types Z that have been declared; and finally the set of files $\mathcal{I}$ that have been included.

Reduction rules are of the form $\mathcal{F} \vdash \langle h; \mathcal{A}; \Delta; x \rangle \longrightarrow \langle h'; \mathcal{A}'; \Delta'; x' \rangle$. Here $\mathcal{F}$ represents the file system, which maps header file names to their corresponding fragments. Preprocessing fragment f begins with an accumulator whose components are all $\emptyset$ (which we write $\mathcal{A}_\emptyset$); an h component set to $\circ$; and a given $\mathcal{F}$ and an initial set of defines Δ_f.

In the rules in Figure 5, 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. We write $\mathcal{A}^X$ for the X component of $\mathcal{A}$. All of the rules increase the contents of the accumulator monotonically.

We discuss the preprocessor semantics briefly. [SEQ] reduces the first statement in a fragment. We abuse notation and write f', f as the concatenation of fragments f' and f, where $\cdot, f' = f'$ and $(s, f'), f'' = s, (f', f'')$. [INCL] looks up file name h in the file system and reduces to the corresponding fragment. It also inserts a special command $\mathsf{pop}\ h'$ where h' is the file currently being processed. When the preprocessor finishes reducing h, the [EOH] rule restores the current file to h'. Notice that the semantics become stuck if a header file is included twice, because then the premise $h \notin \mathcal{A}^{\mathcal{I}}$ of [INCL] is not satisfied. While nonstandard, this semantics simplifies the rule specification. In practice, programmers mostly use the `#ifndef` pattern (Section 2.3) to make duplicate file inclusion a no-op; our implementation of CMOD emits a warning if it discovers this practice is not followed.

$$
\begin{array}{rrcl}
\text{symbols} & N & ::= & \cdot \mid g \to \tau,\ N \\
\text{heap} & H & ::= & \cdot \mid g \to e,\ H \\
\text{named types} & T & ::= & \cdot \mid t \to \tau^h,\ T \mid t \to \tau^\circ,\ T \\
\text{exports} & E & \in & 2^g \\
\text{imports} & I & \in & 2^g \\
\text{symbol decls} & D & \in & 2^g \\
\text{macro changes} & \mathcal{C} & \in & 2^m \\
\text{macro uses} & \mathcal{U} & \in & 2^m \\
\text{type decls} & Z & \in & 2^t \\
\text{includes} & \mathcal{I} & \in & 2^h \\
\text{accumulator} & \mathcal{A} & = & (\mathcal{C}, I, T, \mathcal{I}, Z, E, N, D, \mathcal{U}, H) \\
\text{file system} & \mathcal{F} & : & h \to f \\
\text{defines} & \Delta & \in & 2^m
\end{array}
$$

$$
[\textsc{Seq}]\ \frac{\mathcal{F} \vdash \langle h; \mathcal{A}; \Delta; s \rangle \longrightarrow \langle h; \mathcal{A}'; \Delta'; f' \rangle}{\mathcal{F} \vdash \langle h; \mathcal{A}; \Delta; s, f \rangle \longrightarrow \langle h; \mathcal{A}'; \Delta'; f', f \rangle}
$$

$$
[\textsc{Incl}]\ \frac{h \notin \mathcal{A}^{\mathcal{I}} \qquad f = \mathcal{F}(h), \mathsf{pop}\ h' \qquad \mathcal{A}' = \mathcal{A}[\mathcal{I} \leftarrow^+ h]}{\mathcal{F} \vdash \langle h'; \mathcal{A}; \Delta; \mathsf{include}\ h \rangle \longrightarrow \langle h; \mathcal{A}'; \Delta; f \rangle}
$$

$$
[\textsc{Eoh}]\ \frac{}{\mathcal{F} \vdash \langle h'; \mathcal{A}; \Delta; \mathsf{pop}\ h \rangle \longrightarrow \langle h; \mathcal{A}; \Delta; \cdot \rangle}
$$

$$
[\textsc{Def}]\ \frac{\mathcal{A}' = \mathcal{A}[\mathcal{C} \leftarrow^+ m,\ \mathcal{U} \leftarrow^+ m] \qquad \Delta' = \Delta \cup \{m\}}{\mathcal{F} \vdash \langle h; \mathcal{A}; \Delta; \mathsf{def}\ m \rangle \longrightarrow \langle h; \mathcal{A}'; \Delta'; \cdot \rangle}
$$

$$
[\textsc{Undef}]\ \frac{\mathcal{A}' = \mathcal{A}[\mathcal{C} \leftarrow^+ m,\ \mathcal{U} \leftarrow^+ m] \qquad \Delta' = \Delta - \{m\}}{\mathcal{F} \vdash \langle h; \mathcal{A}; \Delta; \mathsf{undef}\ m \rangle \longrightarrow \langle h; \mathcal{A}'; \Delta'; \cdot \rangle}
$$

$$
[\textsc{Ifdef+}]\ \frac{m \in \Delta \qquad \mathcal{A}' = \mathcal{A}[\mathcal{U} \leftarrow^+ m]}{\mathcal{F} \vdash \langle h; \mathcal{A}; \Delta; \mathsf{ifdef}\ m\ \mathsf{then}\ f_+\ \mathsf{else}\ f_- \rangle \longrightarrow \langle h; \mathcal{A}'; \Delta; f_+ \rangle}
$$

$$
[\textsc{Ifdef-}]\ \frac{m \notin \Delta \qquad \mathcal{A}' = \mathcal{A}[\mathcal{U} \leftarrow^+ m]}{\mathcal{F} \vdash \langle h; \mathcal{A}; \Delta; \mathsf{ifdef}\ m\ \mathsf{then}\ f_+\ \mathsf{else}\ f_- \rangle \longrightarrow \langle h; \mathcal{A}'; \Delta; f_- \rangle}
$$

$$
[\textsc{Extern}]\ \frac{\mathcal{A}' = \mathcal{A}[D \leftarrow^+ g,\ N \leftarrow^+ (g \mapsto \tau)]}{\mathcal{F} \vdash \langle h; \mathcal{A}; \Delta; \mathsf{extern}\ g : \tau \rangle \longrightarrow \langle h; \mathcal{A}'; \Delta; \cdot \rangle}
$$

$$
[\textsc{Let}]\ \frac{\mathcal{A}' = \mathcal{A}[H \leftarrow^+ (g \mapsto e),\ N \leftarrow^+ (g \mapsto \tau), \qquad E \leftarrow^+ g,\ D \leftarrow^+ g,\ I \leftarrow^+ \mathrm{fg}\,(e)]}{\mathcal{F} \vdash \langle h; \mathcal{A}; \Delta; \mathsf{let}\ g : \tau = e \rangle \longrightarrow \langle h; \mathcal{A}'; \Delta; \cdot \rangle}
$$

$$
[\textsc{Type-Decl}]\ \frac{\mathcal{A}' = \mathcal{A}[Z \leftarrow^+ t]}{\mathcal{F} \vdash \langle h; \mathcal{A}; \Delta; \mathsf{type}\ t \rangle \longrightarrow \langle h; \mathcal{A}'; \Delta; \cdot \rangle}
$$

$$
[\textsc{Type-Def}]\ \frac{\mathcal{A}' = \mathcal{A}[T \leftarrow^+ (t \mapsto \tau^h)]}{\mathcal{F} \vdash \langle h; \mathcal{A}; \Delta; \mathsf{let\ type}\ t = \tau \rangle \longrightarrow \langle h; \mathcal{A}'; \Delta; \cdot \rangle}
$$

Figure 5: Operational Semantics for the Preprocessor

[DEF] and [UNDEF] add or remove m from the set of currently-defined macros Δ, and mark m as being changed and used. [IFDEF+] and [IFDEF-] reduce to either f_+ or f_- depending on whether m has been defined or not. In either case, we add m to the set of macros whose values have been used.

The remaining rules handle declarations and definitions. The C preprocessor ignores these, but CMOD's preprocessor extracts information from them to enforce its rules. [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 C compiler ensures that the same variable is always given the same type within a fragment (Section 3.4). [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. Types in T are annotated with the current file h, which is $\circ$ if the current file is not a header.

3.2 CMOD Rules

We now formally specify the rules presented in Section 2. To state the rules more concisely, we introduce new notation to describe the final accumulator after preprocessing beginning from the empty accumulator:

Definition 3.1 (Partial Preprocessing) *We write $\Delta; \mathcal{F} \vdash f \rightsquigarrow \langle \mathcal{A}; f' \rangle$ as shorthand for $\mathcal{F} \vdash \langle \circ; \mathcal{A}_\emptyset; \Delta; f \rangle \xrightarrow{*} \langle h; \mathcal{A}; \Delta'; f' \rangle$, where $\xrightarrow{*}$ is the reflexive, transitive closure of the rules in Figure 5.*

Definition 3.2 (Complete Preprocessing) *We write $\Delta; \mathcal{F} \vdash f \rightsquigarrow \mathcal{A}$ as shorthand for $\Delta; \mathcal{F} \vdash f \rightsquigarrow \langle \mathcal{A}; \cdot \rangle$.*

CMOD's rules are shown in Figure 6. The first three rules assume there is a common initial macro environment Δ under which all fragments are preprocessed; the fourth rule ensures that this assumption makes sense. Figure 6(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 global variable name g in N, which contains any global variable names imported by one fragment and defined by the other, it must be the case that $\Delta; \mathcal{F} \vdash g \xleftarrow{\text{decl}} \mathcal{A}_1^{\mathcal{I}} \cap \mathcal{A}_2^{\mathcal{I}}$, i.e., g is declared in a header file that both f_1 and f_2 include.

Figure 6(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 types 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. Note that we are justified in treating T_i as a map because the C 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.

Figure 6(c) defines the judgment $\Delta; \mathcal{F} \vdash \mathcal{R}_3(f)$, which enforces Rule 3: any two headers h_1 and h_2 that are both included in some fragment must be vertically-independent. 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] calculate the accumulator $\mathcal{A}_1$ that results from preprocessing f up to the inclusion of h. The remaining premises check that the preprocessing of h within the initial environment can in no way be influenced by $\mathcal{A}_1$. No macros changed in $\mathcal{A}_1$ (described by $\mathcal{A}_1^{\mathcal{C}}$) are used by h (described by $\mathcal{A}_2^{\mathcal{U}}$); likewise, no macros changed by h (in $\mathcal{A}_2^{\mathcal{C}}$) are used by files that came earlier (in $\mathcal{A}_1^{\mathcal{U}}$). Put together, these conditions ensure that h is vertically-independent of any files that came earlier. Note that `config.h` files are forbidden by this rule. Our implementation requires all files to include the same `config.h` initially; the equivalent in our formal system is to start with an accumulator and initial Δ from preprocessing `config.h`.

Figure 6(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.

$$\text{[Sym-Decl]}\quad \frac{h \in \mathcal{I} \qquad \Delta; \mathcal{F} \vdash \mathcal{F}(h) \rightsquigarrow \mathcal{A} \qquad g \in \mathcal{A}^D}{\Delta; \mathcal{F} \vdash g \overset{\text{decl}}{\longleftarrow} \mathcal{I}}$$

$$\text{[Rule 1]}\quad \frac{\Delta; \mathcal{F} \vdash f_1 \rightsquigarrow \mathcal{A}_1 \qquad \Delta; \mathcal{F} \vdash f_2 \rightsquigarrow \mathcal{A}_2 \qquad N = \left(\mathcal{A}_1^I \cap \mathcal{A}_2^E\right) \cup \left(\mathcal{A}_1^E \cap \mathcal{A}_2^I\right) \qquad \forall g \in N \;.\; \Delta; \mathcal{F} \vdash g \overset{\text{decl}}{\longleftarrow} \mathcal{A}_1^{\mathcal{I}} \cap \mathcal{A}_2^{\mathcal{I}}}{\Delta; \mathcal{F} \vdash \mathcal{R}_1(f_1, f_2)}$$

(a) Rule 1: Shared Headers

$$\text{[Named-Types-Ok]}\quad \frac{\forall (t \mapsto \tau^\circ) \in T_1 \;.\; t \notin \text{dom}\,(T_2) \qquad \forall t \in \text{dom}\,(T_1) \cap \text{dom}\,(T_2) \;.\; T_1(t) = \tau_1^{h_1} \;\wedge\; T_2(t) = \tau_2^{h_2} \Rightarrow h_1 = h_2}{\vdash_\tau T_1, T_2}$$

$$\text{[Rule 2]}\quad \frac{\Delta; \mathcal{F} \vdash f_1 \rightsquigarrow \mathcal{A}_1 \qquad \Delta; \mathcal{F} \vdash f_2 \rightsquigarrow \mathcal{A}_2 \qquad \vdash_\tau \mathcal{A}_1^T, \mathcal{A}_2^T \qquad \vdash_\tau \mathcal{A}_2^T, \mathcal{A}_1^T \qquad f_1 \neq f_2}{\Delta; \mathcal{F} \vdash \mathcal{R}_2(f_1, f_2)}$$

(b) Rule 2: Type Ownership

$$\text{[Partial-Indep]}\quad \frac{\Delta; \mathcal{F} \vdash f \rightsquigarrow \langle \mathcal{A}_1; \mathsf{include}\ h, f' \rangle \qquad \Delta; \mathcal{F} \vdash \mathcal{F}(h) \rightsquigarrow \mathcal{A}_2 \qquad \mathcal{A}_1^{\mathcal{C}} \cap \mathcal{A}_2^{\mathcal{U}} = \emptyset \qquad \mathcal{A}_1^{\mathcal{U}} \cap \mathcal{A}_2^{\mathcal{C}} = \emptyset}{\Delta; \mathcal{F} \vdash f \otimes h}$$

$$\text{[Rule 3]}\quad \frac{\Delta; \mathcal{F} \vdash f \rightsquigarrow \mathcal{A} \qquad \forall h \in \mathcal{A}^{\mathcal{I}} \;.\; \Delta; \mathcal{F} \vdash f \otimes h}{\Delta; \mathcal{F} \vdash \mathcal{R}_3(f)}$$

(c) Rule 3: Vertical Independence

$$\text{[Rule 4]}\quad \frac{\Delta_f; \mathcal{F} \vdash f \rightsquigarrow \mathcal{A} \qquad ((\Delta - \Delta_f) \cup (\Delta_f - \Delta)) \cap \mathcal{A}^{\mathcal{U}} = \emptyset}{\Delta; \mathcal{F} \vdash \mathcal{R}_4(f, \Delta_f)}$$

(d) Rule 4: Environment Compatibility

$$\text{[All]}\quad \frac{\forall f_1, f_2 \in \mathcal{P} \;.\; \Delta; \mathcal{F} \vdash \mathcal{R}_1(f_1, f_2) \qquad \forall f_1, f_2 \in \mathcal{P} \;.\; \Delta; \mathcal{F} \vdash \mathcal{R}_2(f_1, f_2) \qquad \forall f \in \mathcal{P} \;.\; \Delta; \mathcal{F} \vdash \mathcal{R}_3(f)}{\Delta; \mathcal{F} \vdash \mathcal{R}(\mathcal{P})}$$

(e) Rules 1–3 combined

Figure 6: Cmod Rules

Finally, by [Rule 4], we can assume that there is a single Δ that all Δ_f's are compatible with. Figure 6(e) defines the judgment $\Delta; \mathcal{F} \vdash \mathcal{R}(\mathcal{P})$, which holds if a program $\mathcal{P}$ satisfies Rules 1, 2, and 3 in this common Δ. Thus if $\Delta; \mathcal{F} \vdash \mathcal{R}(\mathcal{P})$ 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, and must use vertically-independent header files.

3.3 Compilation and Linking

In order to prove that the rules in Figure 6 are sound, we need to precisely describe the compilation and linking process. The output of the normal C compiler is an object file containing code and data for globals, a list of global variables that are defined (exports), and a list of global variables that are used but not defined (imports). Since one of our goals is to prove type safety at link time, our formal compiler output uses Glew and Morrisett's MTAL_0 typed object file notation [9]. In this system, 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.

In Figure 7, [Compile] describes the object file produced by the C compiler from a fragment f, given an initial set of macro definitions Δ and a file system $\mathcal{F}$. The fragment is first preprocessed. In order to be compiled, the global type environment N must be consistent according to [WF-Map], meaning that the same symbol must always be assigned the same type. Moreover, the compiler ensures that all code and data is well-typed given the global type environment N and the type definitions T and declarations Z, as defined by [WF-Heap]. The first premise of [WF-Heap] requires that the same global symbol is not defined more

$$\text{[WF-Map]}\quad \frac{g_i = g_j \Rightarrow \tau_i = \tau_j}{\vdash g_1 \mapsto \tau_1, \ldots, g_p \mapsto \tau_p}$$

$$\text{[WF-Heap]}\quad \frac{\forall i, j \in [1..p]\ .\ g_i = g_j \Rightarrow i = j \qquad \forall i \in [1..p]\ .\ Z; T; N \vdash e_i : N(g_i)}{Z; T; N \vdash g_1 \mapsto e_1, \ldots, g_p \mapsto e_p}$$

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

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

Figure 7: Compiler and Linker Rules

than once, and the second premise ensures that each global symbol's type matches its type in N. Note that we omit the rules for typing expressions, as they are simply the standard lambda-calculus type checking rules. Assuming these checks succeed, then [Compile] produces an object file $[\Psi_I \Rightarrow H : \Psi_E]$. Here we write $N|_S$ to mean the mapping that is the same as N, but its domain is restricted to S, and with only one occurrence of each symbol (which is well-defined since we already checked that symbols are declared consistently). Thus in the output object file, the H component is just as in the preprocessing accumulator output, the exports Ψ_E contains the types for any defined symbols, and the imports Ψ_I contains the types for any symbols that were used (in I) but not defined (in E). Note that we are simplifying one issue, namely that in C, declarations must come before uses, which is not enforced here since type checking is done after all the information has been gathered. We could add this restriction by making the system slightly more complicated, but we believe it adds no interesting issues.

Rule [Link] describes the process of linking two object files. When object files are linked together, the imports, code and data, and exports are computed 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.

3.4 Formal Properties

We can now formally state the information hiding and link-time type safety properties of Cmod. The proofs for the theorems in this section are provided in the appendix. 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 3.3 *$\Delta; \mathcal{F} \vdash \mathcal{R}_4(f, \Delta_f)$ implies that if $\Delta_f; \mathcal{F} \vdash f \rightsquigarrow \mathcal{A}$, then $\Delta; \mathcal{F} \vdash f \rightsquigarrow \mathcal{A}$; and if $\Delta_f; \mathcal{F} \vdash f \xrightarrow{comp} [\Psi_I \Rightarrow H : \Psi_E]$, then $\Delta; \mathcal{F} \vdash f \xrightarrow{comp} [\Psi_I \Rightarrow H : \Psi_E]$.*

Thus below we assume a single Δ for all fragments.

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 completely link to have no free, unresolved symbols. Thus we can define the compilation of an entire program:

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

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

$$[\text{WF-Int}]\ \frac{i \neq j \Rightarrow g_i \neq g_j}{\vdash g_1 \mapsto \tau_1, \ldots, g_p \mapsto \tau_p}$$

$$[\text{Int-Sub}]\ \frac{p \geq q \qquad \vdash g_1 \mapsto \tau_1, \ldots, g_p \mapsto \tau_p}{\vdash g_1 \mapsto \tau_1, \ldots, g_p \mapsto \tau_p \leq g_1 \mapsto \tau_1, \ldots, g_q \mapsto \tau_q}$$

$$[\text{MTAL}_0\text{-WF-Obj}]\ \frac{\vdash \Psi_I \qquad \vdash \Psi_A \leq \Psi_E \qquad \Psi_I \cup \Psi_A \vdash H : \Psi_A \qquad \mathrm{dom}\,(\Psi_I) \cap \mathrm{dom}\,(\Psi_A) = \emptyset}{\vdash [\Psi_I \Rightarrow H : \Psi_E]}$$

$$[\text{MTAL}_0\text{-Compat}]\ \frac{\forall g \in \mathrm{dom}\,(\Psi_1) \cap \mathrm{dom}\,(\Psi_2)\ .\ \Psi_1(g) = \Psi_2(g)}{\vdash \Psi_1 \sim \Psi_2}$$

$$[\text{MTAL}_0\text{-LC}]\ \frac{\vdash \Psi_{I1} \sim \Psi_{I2} \qquad \vdash \Psi_{I1} \sim \Psi_{E2} \qquad \vdash \Psi_{I2} \sim \Psi_{E1} \qquad \mathrm{dom}\,(\Psi_{E1}) \cap \mathrm{dom}\,(\Psi_{E2}) = \emptyset}{\vdash [\Psi_{I1} \Rightarrow H_1 : \Psi_{E1}] \overset{\text{lc}}{\leftrightarrow} [\Psi_{I2} \Rightarrow H_2 : \Psi_{E2}]}$$

$$[\text{MTAL}_0\text{-Link}]\ \frac{\vdash [\Psi_{I1} \Rightarrow H_1 : \Psi_{E1}] \qquad \vdash [\Psi_{I2} \Rightarrow H_2 : \Psi_{E2}] \qquad \vdash [\Psi_{I1} \Rightarrow H_1 : \Psi_{E1}] \overset{\text{lc}}{\leftrightarrow} [\Psi_{I2} \Rightarrow H_2 : \Psi_{E2}] \qquad \mathrm{dom}\,(H_1) \cap \mathrm{dom}\,(H_2) = \emptyset}{\vdash [\Psi_{I1} \Rightarrow H_1 : \Psi_{E1}]\ \text{link}\ [\Psi_{I2} \Rightarrow H_2 : \Psi_{E2}] \rightsquigarrow [(\Psi_{I1} \cup \Psi_{I2}) \setminus (\Psi_{E1} \cup \Psi_{E2}) \Rightarrow H_1 \cup H_2 : \Psi_{E1} \cup \Psi_{E2}]}$$

Figure 8: MTAL_0 [9]

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

This theorem says that if $\mathcal{P}$ obeys the Cmod rules and includes headers h_j, then any symbol g that is not in $\mathcal{A}^D_{hj}$ for any j (i.e., is not declared in any header file) is never imported.

For type names, we can prove a related property: Any type name owned by a source fragment (a code file) has no concrete type in any other fragment.

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* $\Delta; \mathcal{F} \vdash f_i \rightsquigarrow \mathcal{A}_i$*. Further suppose that* $(t \mapsto \tau^\circ) \in \mathcal{A}^T_i$*. Then for any fragment* $f_j \in \mathcal{P}$ *such that* $f_i \neq f_j$ *and* $\Delta; \mathcal{F} \vdash f_j \rightsquigarrow \mathcal{A}_j$*, we have* $t \notin \mathrm{dom}\left(\mathcal{A}^T_j\right)$*.*

This theorem says that if $\mathcal{P}$ obeys the Cmod rules and contains fragment f_i, then any type t owned by f_i is not owned by any other fragments $f_j \neq f_i$. Together, Theorems 3.5 and 3.6 give us Property 2.1.

Finally, we show that Cmod enforces type safety at link time. Rather than show this directly, we reduce our system to MTAL_0, for which link-time type safety has been shown [9]. Figure 8 gives the rules for MTAL_0, which we discuss briefly from bottom to top. [MTAL_0-Link] says that the linking process is type safe if each object file is well-formed, if the two object files are link-compatible, and if the definitions in both files are disjoint. [MTAL_0-LC] defines link-compatibility, which simply requires that the exports and imports of the object files assign the same types to shared symbols (using [MTAL_0-Compat]) and that the same symbol is not exported twice. Finally, [MTAL_0-WF-Obj] defines well-formedness of an object file. This rule holds if there is some heap typing Ψ_A (a mapping from global names to types) such that H can be typed in $\Psi_I \cup \Psi_A$ to yield Ψ_A. This is shorthand for requiring that for each $g \in \mathrm{dom}\,(N)$, it is the case that

$\Psi_I \cup \Psi_A \vdash H(g) : \Psi_A(g)$. (As before, we omit the standard lambda calculus typing rules for expressions.) It must also be the case that $\Psi_A \leq \Psi_E$, meaning that the exports are a subset of the heap. Finally, nothing in the heap can be part of the imports.

MTAL_0 does not include type abstraction or type names. The full MTAL system [9] does, but for technical reasons is not quite strong enough to encode certain uses of abstract types in our system, though it should be possible to change it to do so [22]. However, notice that Rule 2 requires that a type name has the same definition everywhere. Thus we claim (without a formal proof) that the use of abstract types cannot violate type safety at link time. In essence, given a program with type abstraction that obeys the CMOD rules, there is only one way to concretize all abstract types in the program. In the following, we assume all types are expressed directly, and not through a possibly-abstract name.

To show that linking is type safe, we can prove that if the program compiles and passes the CMOD checks, then any pair of object files linked together satisfy [MTAL_0-LINK].

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

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

Then

$$\vdash [\Psi_{Ii} \Rightarrow H_i : \Psi_{Ei}] \text{ link } [\Psi_{Ij} \Rightarrow H_j : \Psi_{Ej}] \rightsquigarrow O_{ij}$$

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.

3.5 Handling Full C

The full C language includes several features not present in the formal system, such as conditionals `#if` and `#ifndef`, token concatenation `##`, and macro substitution (e.g., `#define FOO(x) (x+1)`). Moreover, C allows preprocessor commands at arbitrary syntactic positions. Put together, these additional features would be extremely hard to add to our formal system. Nevertheless, we claim that they do not affect the soundness of CMOD.

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-]). We can extend this idea to the full preprocessor by also counting as uses (1) macro references in other conditionals and (2) macro substitutions; 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 operate on the *preprocessed* files (for example, [RULE 1] preprocesses each fragment and the header file that contains the declaration). Thus even under the full C preprocessor, [RULE 3] and [RULE 4] ensure Principle 2.3, and therefore [RULE 1] and [RULE 2] correctly enforce information hiding and type safety.

4 Implementation

We have implemented CMOD for the full C language.[2] The two main parts of our implementation are tools called `cwrap` and `lwrap`, which are scripts that wrap the C compiler and linker, shown in Figure 9. `cwrap` uses preprocessor hooks (via `cpplib`, part of GCC) to capture `#included` file names, macro uses

[2] `http://www.cs.umd.edu/~saurabhs/CMod`

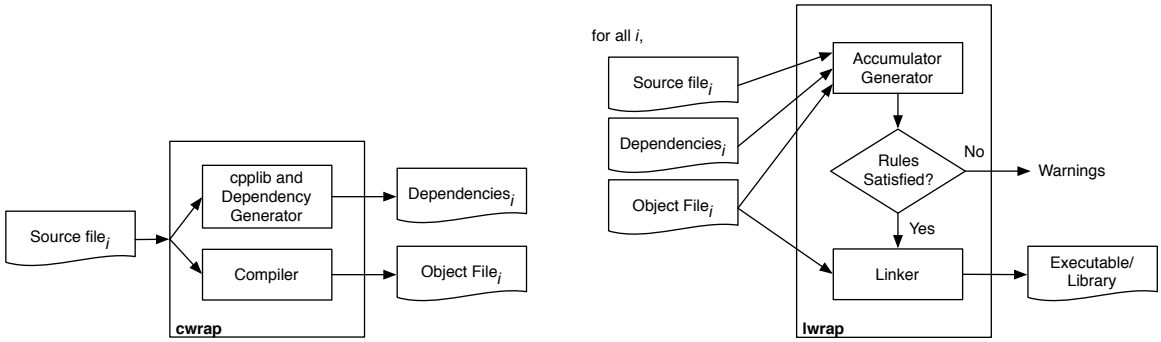

Figure 9: CMOD Architecture

Program	Targets	LoC	`.c`	`.h`	Description
gzip-1.2.4	1	5k	15	6	Compression utility
m4-1.4.4	2	10k	19	7	Macro language
bc-1.06	3	10k	19	12	Text-based calc
rcs-5.7	9	12k	25	4	Version control
vsftpd-2.0.3	1	12k	34	41	"Very Secure" FTPD
flex-2.5.4	2	16k	22	10	Code generation tool
xinetd-2.3.14	8	16k	60	68	Net services daemon
mt-daapd-0.2.4	1	18k	23	26	iTunes (DAAP) server
retawq-0.2.6c	1	21k	5	8	Text-based browser
bison-2.3	3	21k	57	94	Parser generator
jgraph-8.3	1	30k	9	4	Graphing Tool
gawk-3.1.5	4	30k	21	20	Pattern processor
openssh-4.2p1	13	52k	157	119	Secure Shell
gnuplot-4.0.0	4	80k	49	100	Plotting utility
zebra-0.94	8	107k	111	118	Routing daemon BE
Total	61	440k	626	637	

Figure 10: Benchmark Programs

and definitions, and the initial macro environment. Per-file symbol imports and exports are already stored in the generated ELF object files. During linking, `lwrap` uses `ctags` [6] to extract declaration and type information from the preprocessed source code generated during compilation. Put together, all these sources of information are sufficient for `lwrap` to check Rules 1–3.[3] To check Rule 4, CMOD attempts to synthesize a single global environment from the ones used to compile each file. It does this by unioning each file's local environment after restricting the local environments to only macros that are used. CMOD emits a warning if the synthesized global environment is not consistent with the local environments.

Recall that our semantics assumes the same file is never included twice. CMOD checks that headers follow the `#ifndef` pattern, which prevents duplicate header inclusions, and emits a warning if the pattern is not followed. CMOD also assumes that system headers match their corresponding libraries, since the sources for these are not available when compiling the projects.

5 Experiments

We applied CMOD to a number of publicly available open source projects (Figure 10), with the goal of measuring how well they conform to CMOD's rules, and to determine whether rule violations are indeed problematic. We chose projects of varying sizes (5–107k lines of code), varying usage and stages of development (e.g., `xinetd`, `flex`, `gawk`, and `bison` 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 ran CMOD on a dual-processor 2.80GHz

[3]We could check Rule 3 entirely at compile-time, rather than link-time, but we have found it convenient to check all rules at once.

Program	Rule Violations				Prop. Viol.		Changes Required†								Build Time		
	Ru. 1	Ru. 2	Ru. 3	Ru. 4	Inf. Hid.	Typ.	Ru. 1 +	Ru. 1 -	Ru. 2 +	Ru. 2 -	Ru. 3 +	Ru. 3 -	Ru. 4 +	Ru. 4 -	Stock (s)	CMOD (s)	% ovr
gzip	2	-	1	-	2	-	×	×	-	-	2	2	-	-	1.0	2.1	120%
m4*	2	1	-	-	2	1	2	1	1f,2	2	-	-	-	-	3.3	5.6	54%
bc*	8	1 (1)	6	-	4	-	4	1	-	-	1f,89	86	-	-	2.4	4.5	86%
rcs*	-	1	-	-	-	-	-	-	6	6	-	-	-	-	3.1	13.2	331%
vsftpd	4	-	9	-	-	-	1	-	-	-	3	13	-	-	2.7	4.4	67%
flex	5	6	-	-	3	-	4	-	1	15	1	-	-	-	4.7	9.8	107%
xinetd*	10	3 (20)	-	-	3	-	5	1	1f,7	10	-	-	-	-	6.2	17.7	187%
mt-daapd	16	1	-	-	5	-	13	2	-	-	-	-	-	-	6.3	9.8	57%
retawq*	-	-	16	-	-	-	-	-	-	-	8f,10	12	-	-	5.6	7.8	39%
bison*	3	17	8	1	2	-	2	-	2f,6	140	16	10	3	3	9.9	18.8	89%
jgraph	56	-	-	-	54	-	46	2	-	-	-	-	-	-	1.0	1.6	79%
gawk*	41	-	22	-	38	-	29	5	-	-	6f,7	10	-	-	11.1	18.3	64%
openssh*	68 (38)	-	53	-	63	-	62	1	-	-	2f,133	127	-	-	28.3	163.8	479%
gnuplot*	×	×	353‡	-	-	-	×	×	×	×	×	×	-	-	28.9	41.2	42%
zebra*	139	-	53	-	64	5	64	10	-	-	27	6	-	-	32.9	86.6	163%
Total	354 (38)	30 (21)	168	1	240	6	232	23	4f,22	173	17f,286	266	3	3		(avg)	137%

*Has `config.h` file. †Line or file (f) additions (+) and deletions (-). ‡gnuplot count not included in total

Figure 11: Experimental Results

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. For the first run, we tabulated Rule 3 and Rule 4 violations, and examined any CMOD warnings about header files not using the `#ifndef` pattern. We manually verified that every flagged header was either harmless when included twice (e.g., it only contained prototypes), or that the header could never be included twice without a C compiler warning. We then fixed the Rule 3 and Rule 4 violations and reran CMOD to gather the Rule 1 and 2 violations.

Figure 11 summarizes our results. 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. Next we discuss the remaining columns, which count the number of rule violations, violations of Properties 2.1 (Information Hiding) and 2.2 (Type Safety), changes required to fix rule violations, and running time.

5.1 Rule Violations

Figure 11 lists the rule violation counts in the second group of columns, with the additional false positives due to inaccuracies in parentheses. We have not pruned duplicate violations for the same source in different targets. 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 dependency between the files. Lastly, a Rule 4 violation corresponds to a target whose linked object files were compiled in incompatible preprocessor environments.

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: Rule 1 violations are often dangerous, because they can permit a provider and client to disagree on the type of a symbol without generating an error at compile-time (as discussed in Section 2.2). We found 349 violations that seem problematic. The most common case is when a source file locally declares an extern symbol that does not appear in a header (240 times). As discussed in Section 5.2, these are arguably information hiding violations. The next most common Rule 1 violations occur when a provider `.c` file fails to `#include` a header containing the symbols it exports (81 times) or a client `.c` file locally declares a prototype

instead of `#include`ing a header file, even though there is a header with the symbol (28 times). Many of the first category of Rule 1 violations are due to `jgraph`, which heavily uses K&R-style implicit function declarations rather than prototypes.

The five remaining Rule 1 violations appear safe. Three of these are due to code files that are `#include`d in another file. Since the other file textually incorporates the first, it does not need a mediating header to ensure symbols have matching types, but Rule 1 requires this. The last two Rule 1 violations occur in `gzip`, which includes assembler sources that define exported symbols but cannot `#include` their header.

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. We found 6 violations in which the same type definition was 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 24 violations for practices that are safe. In one case, a type name is reused at two different types in different files. In this particular case each definition is local to a single file, so the code is safe. Enforcing a kind of `static` for types would eliminate this violation. In the remaining 23 violations, there are duplicate identical type definitions created in auto-generated code. This is not a pattern CMOD can easily recognize.

Rule 3: Rule 3 violations make it harder to reason about headers in isolation. There are a total of 33 Rule 3 violations that we think are bad practice. We found 31 violations that are vertical dependencies in which header files depend on the order they are included, which we have argued is undesirable. Two additional Rule 3 violations 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 135 violations are safe practices that CMOD does not recognize as such. 116 of the Rule 3 violations are due to limitations in modeling `config.h`. In particular, several programs have multiple global configuration files that are themselves `#include`d in `config.h`. Since CMOD only treats `config.h` specially, dependencies on these other headers are flagged as rule violations. We believe that Rule 3 could be relaxed to allow this case.

The other 19 of the violations occur when one file is included after a `#define` of a macro it depends on, and the file contains code definitions rather than an interface. This is a violation of Rule 3, but as mentioned in Section 2.3, this case could be handled specially.

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. Since these vertical dependencies are clearly intended, 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 total.

Rule 4: The one Rule 4 violation is caused by compiling a library and a source file that links with it using macro environments that differed for one macro name. We think this should be avoided, and in this case the violation was easily fixed.

False Positives: CMOD reported 38 Rule 1 violations that were false positives, meaning that CMOD issues a warning but the code does not actually violate the rule. The culprit was `ctags`, which sometimes fails to parse complex code, leaving CMOD with inaccurate information about source files. An example of where this happens is shown in Figure 12. CMOD also reported 21 false positives for Rule 2. Twenty of these reports are due to `xinetd`, in which library headers are copied after a library is built and then are included by library clients. CMOD does not know that the copied header should be treated as identical to the original header, and so complains about duplicate type definitions. The Rule 2 false positive in `bc` is due to a code parsing error in our implementation.

5.2 Property Violations

Of those rule violations we consider bad practice, some directly compromise Properties 2.1 (Information Hiding) and 2.2 (Type Safety). The middle columns in Figure 11 measure how often this occurs in our benchmarks.

```
56  void fatal(const CHAR *, ...) __dead
        __attribute__((format(printf, 1, 2)));
65  void cleanup_exit(int) __dead;
```

Figure 12: Example false positive in openssh. ctags fails to parse complicated C syntax, in this case the __dead attribute.

Information hiding violations degrade a program's modular structure, complicating maintenance and 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 programmer's intended 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 68% (240 out of 354) of the Rule 1 violations. There were no Rule 2 violations that showed information hiding problems.

There were a total of 6 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 interesting 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.

5.3 Examples of Suspect Coding Practices

Figure 13(a) shows two examples from `zebra`. In the first example (providers `if_netlink.c` and `rt_netlink.c` and the client `rt_netlink.c`), the client locally declares `interface_lookup_netlink()` and `netlink_route_read()` with the wrong return type. (Note that this particular typo may or may not be problematic, depending on whether it is important to check the return value of this function.) Ironically, in this case the client code even includes a comment describing where the original definition is from—yet the types are still wrong. In the second example (`bgpd.c` and `bgp_zebra.c`), the function `bgp_zebra_init` takes one argument but is called with no arguments, which could lead to strange behavior at run time.

Examples of duplicate type definitions are shown in Figure 13(b) and (c). Figure 13(b) shows an example from `m4` in which the typedef `boolean` is defined differently in two places. In this case although the types do not matches, values of these two different `boolean`s never intermix, and so the code is safe. We removed this rule violation by alpha-renaming the types; a notion of `static` for type definitions would also have eliminated the warnings, since most such types are intended to be file-local. Figure 13(c) shows a duplicate type definition in `flex`. Here the types happen to match across files, in this case because they are auto-generated. Although auto-generated matching definitions are safe, they are beyond the scope of CMOD, and in general duplicate type definitions are bad for code maintenance.

5.4 Required Changes and Performance

We designed CMOD to enforce modular properties while remaining as backward compatible as possible. To evaluate the latter, we measured the effort required to make a program CMOD-compliant. The second-to-last group 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 inlining or deleting a whole file, usually because code was split across files to no apparent advantage.

if_netlink.c:

```
25  /* Extern from rt_netlink.c */ ⟵ Programmer's comment!
26  void interface_lookup_netlink ();
```

rtread_netlink.c:

```
25  /* Extern from rt_netlink.c */ ⟵ Programmer's comment!
26  void netlink_route_read ();
```

rt_netlink.c:

```
860 int
861 interface_lookup_netlink () { ...
    ...
896 int
897 netlink_route_read () { ...
```

bgpd/bgpd.c:

```
4905 void
4906 bgp_init ()
4907 {
4908        void bgp_zebra_init (); ⟵ Local prototype declaration.
           ...
4919        bgp_zebra_init ();⟵ Local call site with no arguments
```

bgpd/bgp_zebra.c:

```
980 void
981 bgp_zebra_init (int enable) { ... ⟵ Definition
```

(a) Rule 1: Symbol declarations with incorrect types in zebra

lib/regex.c:

```
257 typedef char boolean;
```

m4.h:

```
117 typedef enum { FALSE = 0, TRUE = 1 } boolean;
```

(b) Rule 2: Multiple, inconsistent type definitions in m4

scan.c:

```
34  #if defined __STDC_VERSION__ && ...
35  #include <inttypes.h>
36  typedef int8_t flex_int8_t;
37  typedef uint8_t flex_uint8_t;
    ...
42  #else
    ...
49  #endif/* ! C99 */
```

flexint.h:

```
17  #include <inttypes.h>
18  typedef int8_t flex_int8_t;
19  typedef uint8_t flex_uint8_t;
    ...
```

(c) Rule 2: Duplicate, but consistent type definitions in flex

Figure 13: Example Suspicious Coding Practices

We found it was generally straightforward to make a program comply with CMOD's rules, and most violations required changing only a few lines of code. Violations of Rules 1 and 2 were easy to fix by moving prototypes into headers, or creating headers where required. Violations of Rule 3 required various techniques to fix. Vertical dependencies were easy to fix by converting them into horizontal dependencies. In particular, if a pair of dependent headers always occurs together in consecutive order, then it is easy to move the `#include` of the first header into the second header. Files that do not act as interfaces but are `#include`d can be inlined, and duplicate macro definitions are easy to eliminate. We resolved other vertical dependencies by moving the dependent file into `config.h`, where appropriate. Note that very rarely this suppresses a Rule 1 violation, because now that header is included in more files.

There were four programs we did not bring into full compliance with CMOD. As mentioned earlier, `gzip` includes assembler sources that cannot `#include` header files. `gnuplot` relies on vertical dependencies that cannot be removed without fundamentally changing the design of the program. Lastly, `bc` and `mt-daapd` contain auto-generated type definitions that cause three Rule 2 violations, and which we did not attempt to fix.

Finally, the last three columns in Figure 11 measure the time taken to build the program without and with CMOD. The current prototype of CMOD adds noticeable but acceptable overhead to the compilation procedure. We believe that the performance could be improved with more engineering effort.

6 Related Work

As we stated in the introduction, although many experts recommend using `.h` files as interfaces and `.c` files as implementations [2, 15, 16, 17, 19], the details vary somewhat and are insufficient for full modular safety. King presents the core idea that header files should include declarations, and that both clients and implementations should `#include` the header [17]. McConnell recommends always having public and private headers for modules [19], and mentions using a single public header for a group of implementations, neither of which are 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 safety, 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 2.3, CMOD supports program-wide parameterization (e.g., via `config.h`), but not per-module parameterization, since it is tricky to do correctly in C and is relatively rare.

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. For example, Knit [27], Koala [30], and Click [21] are C and C++ extensions/add-ons that support this style of modular programming. Microsoft's Component Object Technologies (COM) model [4] provides similar facilities to construct dynamically linked libraries (DLLs). These 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.

Some systems aim to support type safety 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.

Finally, a number of researchers have studied the C preprocessor, but not as a means to enforce modularity. Favre [8] proposes a denotational semantics for CPP. Several researchers recommend curtailing or even eliminating the C preprocessor, due to its complexity [7, 18]. Lastly, a number of tools check for erroneous or questionable uses of `cpp` directives, including `lint` [13], PC-lint [25], and Check [28]. The detected bug patterns are fairly localized and generally concern problematic macro expansions.

7 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 safety. 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.

References

[1] J. Barnes. Ada 95 rationale: The language, the standard libraries. *Lecture Notes in Computer Science*, 1247, 1997.

[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] E. Chailloux, P. Manoury, and B. Pagano. *Développement d'applications avec Objective Caml*. O'Reilly, France, 2000.

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

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

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

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

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

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

[10] S. Harbison. *Modula-3*. Prentice-Hall, 1992.

[11] Once-only headers - the C preprocessor. gcc on-line documentation, section 2.4, `http://gcc.gnu.org/onlinedocs/gcc-4.1.1/cpp/Once_002dOnly-Headers.html#Once_002dOnly-Headers`.

[12] 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.

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

[14] S. P. Jones and J. Hughes, editors. *Haskell 98: A Non-strict, Purely Functional Language*. 1999.

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

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

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

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

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

[20] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

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

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

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

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

A Soundness

In this section we show that our rules from Figure 6 are sound for MTAL_0, assuming no type abstraction or type naming is present.

We begin by stating some lemmas about MTAL_0 (Figure 8).

Lemma A.1 (Preservation) *If* $\vdash O_1 \text{ link } O_2 \rightsquigarrow O$ *then* $\vdash O$

Lemma A.2 (Associativity of link) *If* $\vdash (O_1 \text{ link } O_2) \text{ link } O_3 \rightsquigarrow O$ *then* $\vdash O_1 \text{ link } (O_2 \text{ link } O_3) \rightsquigarrow O$.

Lemma A.3 (Commutativity of link) *If* $\vdash O_1 \text{ link } O_2 \rightsquigarrow O$ *then* $\vdash O_2 \text{ link } O_1 \rightsquigarrow O$.

Lemma A.4 *If* $\forall i, j, 1 \leq i, j \leq n, i \neq j$. $\vdash O_i$ link $O_j \rightsquigarrow O_{ij}$ *and if* π *is any permutation of* $\{1 \ldots n\}$ *then*

$$\vdash O_{\pi(1)} \text{ link } O_{\pi(2)} \text{ link } \ldots \text{ link } O_{\pi(n)} \rightsquigarrow O_{1\ldots n}$$

with $\vdash O_{1\ldots n}$.

We start by observing a property induced by Rule 4.

Lemma A.5 *If* $\Delta; \mathcal{F} \vdash \mathcal{R}_4(f, \Delta_f)$ *and* $\Delta_f; \mathcal{F} \vdash f \rightsquigarrow \mathcal{A}$ *then* $\Delta; \mathcal{F} \vdash f \rightsquigarrow \mathcal{A}$.

Proof By observation on the rules in Figure 5. The induction holds trivially for all operational sematics rules except for [IFDEF+] and [IFDEF-]. For those rules and by the premise of Rule 4 in Figure 6, it follows that the semantics rule are never applied with the value of macro being checked that is not common between Δ and Δ_f. Hence the hypothesis holds for [IFDEF+] and [IFDEF-] as well. □

Next we describe a basic property of [COMPILE] from Figure 7.

Lemma A.6 *If two fragments have the same preprocessed output then their compiled objects are the same. More formally, if* $\Delta; \mathcal{F} \vdash f_1 \rightsquigarrow \mathcal{A}$ *and* $\Delta; \mathcal{F} \vdash f_2 \rightsquigarrow \mathcal{A}$ *then* $\Delta; \mathcal{F} \vdash f_1 \xrightarrow{comp} O$ *iff* $\Delta; \mathcal{F} \vdash f_2 \xrightarrow{comp} O$

Proof By inspection of rule [COMPILE]. □

One key property the compiler gives us is that, by themselves, each compiled object file is well-formed (in isolation) according to the rules in Figure 8.

Lemma A.7 (Well-formed compiled objects) *If* $\Delta; \mathcal{F} \vdash f \xrightarrow{comp} [\Psi_I \Rightarrow H : \Psi_E]$ *then* $\vdash [\Psi_I \Rightarrow H : \Psi_E]$

Proof By assumption [COMPILE] holds, and so preprocessing f produces the result accumulator $(\mathcal{C}, I, T, \mathcal{I}, Z, E, N, D, \mathcal{U}, H)$. To show that [MTAL$_0$-WF-OBJ] holds, we need to identify a heap typing Ψ_A that satisfies the rule. We chose $\Psi_A = N|_{\neg \text{dom}(\Psi_I)}$ from the result accumulator, where $\neg \text{dom}(\Psi_I)$ is any symbol not in the domain of Ψ_I. We now can show that each of the premises of [MTAL$_0$-WF-OBJ] hold:

1. $\vdash \Psi_I$. By [COMPILE] we have $\Psi_I = N|_{(I-E)}$, and by definition there are no duplicate elements in the domain of Ψ_I.

2. $\vdash \Psi_A \leq \Psi_E$. By [COMPILE] we have $\Psi_E = N|_E$ and $\Psi_I = N|_{(I-E)}$. By construction we have $\Psi_A = N|_{\neg \text{dom}(\Psi_I)}$. But then any symbol in $\text{dom}(\Psi_E)$ must be in $\text{dom}(\Psi_A)$. Furthermore, we have $\vdash \Psi_A$ because by definition there are no duplicate elements in the domain of Ψ_A.

3. $\Psi_I \cup \Psi_A \vdash H : \Psi_A$. By [COMPILE] we have $N \vdash H$, and then by [WF-HEAP] we have $N \vdash e : N(g)$ where $H(g) = e$ (here we safely assume the same g appears at most once, which also holds by [WF-HEAP]). Further, since $\text{dom}(\Psi_I) \cup \text{dom}(\Psi_A) = \text{dom}(N)$ by construction, and since both are projections of N onto smaller domains, we have $\Psi_I \cup \Psi_A = N$. Thus for every $g \in \text{dom}(N)$, we have $\Psi_I \cup \Psi_A \vdash e : N(g)$. Then since $\text{dom}(\Psi_A) \subseteq \text{dom}(N)$, we have $\Psi_I \cup \Psi_A \vdash e : \Psi_A(g)$, which is the same as $\Psi_I \cup \Psi_A \vdash H : \Psi_A$.

4. $\text{dom}(\Psi_I) \cap \text{dom}(\Psi_A) = \emptyset$. This holds trivially, because by [COMPILE] we have $\Psi_I = N|_{(I-E)}$, and our choice of Ψ_A contains nothing in I in its domain.

□

Now we can prove type-safe linking. Our proof strategy will be to first prove that order-independent fragments can be freely rearranged. We will then use this result to show that if one file imports a symbol and one file exports a symbol, then the CMOD rules force the types to match. Finally, we will show that as a consequence, CMOD enforces type-safe linking.

In this proof, we will use $\mathcal{A}_1 \cup \mathcal{A}_2$ to denote the component-wise union of the two accumulators (which translates to concatenation for any mappings). We also overload the sequencing operator to chain fragments together, so that we may write f_1, f_2 to mean the statements in f_1 followed by the statements in f_2.

We begin by describing the behavior of preprocessing a sequence of fragments:

Lemma A.8 (Preprocessing chains) *If* $\mathcal{F} \vdash \langle h; \mathcal{A}_\emptyset; \Delta; f_1 \rangle \longrightarrow^* \langle h_1; \mathcal{A}_1; \Delta_1; \cdot \rangle$ *then* $\mathcal{F} \vdash \langle h_1; \mathcal{A}_1; \Delta_1; f_2 \rangle \longrightarrow^* \langle h_2; \mathcal{A}_2; \Delta_2; \cdot \rangle$ *if and only if* $\mathcal{F} \vdash \langle h; \mathcal{A}_\emptyset; \Delta; (f_1; f_2) \rangle \longrightarrow^* \langle h_2; \mathcal{A}_2; \Delta_2; \cdot \rangle$

We also observe that preprocessing any fragment to completion leaves the name of the file currently being preprocessed unchanged, because [INCLUDE] inserts any necessary pop statements.

Lemma A.9 *If* $\mathcal{F} \vdash \langle h; \mathcal{A}; \Delta; f \rangle \longrightarrow^* \langle h'; \mathcal{A}'; \Delta'; \cdot \rangle$ *then* $h' = h$.

We use Lemmas A.8 and A.9 without comment in the remainder of the proof.

Next we state a trivial lemma, that preprocessing does not change any macro definitions that are not marked in the accumulator as changed.

Lemma A.10 *Suppose that* $\mathcal{F} \vdash \langle h_0; \mathcal{A}_0; \Delta_0; f_0 \rangle \longrightarrow^* \langle h_1; \mathcal{A}_1; \Delta_1; f_1 \rangle$. *Then* $\Delta_1(m) = \Delta_0(m)$ *for all* $m \notin \mathcal{A}_1^{\mathcal{C}}$.

The next lemma shows that the state of the accumulator "passes through" preprocessing of a fragment, given certain conditions on the fragment. We use this lemma later on to reason about order independence.

Lemma A.11 (Passthrough Property) *Suppose we have*

$$\mathcal{F} \vdash \langle h; \mathcal{A}_\emptyset; \Delta; f \rangle \longrightarrow^k \langle h_k; \mathcal{A}_k; \Delta_k; f_k \rangle$$
$$\mathcal{F} \vdash \langle h; \mathcal{A}; \Delta'; f \rangle \longrightarrow^* \langle h; \mathcal{A}_*; \Delta_*; \cdot \rangle$$

where $\longrightarrow^k$ *is* k *steps of reduction by the rules in Figure 5. Further suppose* $\Delta'(m) = \Delta(m)$ *for all* $m \in \mathcal{A}_k^{\mathcal{U}}$ *and* $\mathcal{A}^{\mathcal{C}} \cap \mathcal{A}_k^{\mathcal{U}} = \emptyset$ *and Then*

$$\mathcal{F} \vdash \langle h; \mathcal{A}; \Delta'; f \rangle \longrightarrow^k \langle h'_k; \mathcal{A}'_k; \Delta'_k; f'_k \rangle$$

where

$h'_k = h_k$, $\mathcal{A}'_k = \mathcal{A} \cup \mathcal{A}_k$, $f'_k = f_k$, *and*

$$\Delta'_k(m) = \begin{cases} \Delta_k(m) & m \in \left(\mathcal{A}_k^{\mathcal{U}} \cup \mathcal{A}_k^{\mathcal{C}}\right) \\ \Delta'(m) & \textit{otherwise} \end{cases}$$

Proof The proof is by induction on k. The base case $k = 0$ is trivial, since $\mathcal{A}_\emptyset \cup \mathcal{A} = \mathcal{A}$ and $\mathcal{A}_\emptyset^{\mathcal{U}} = \mathcal{A}_\emptyset^{\mathcal{C}} = \emptyset$. For the inductive case, assume the property holds for $k - 1$, that is, assume

$$\mathcal{F} \vdash \langle h; \mathcal{A}_\emptyset; \Delta; f \rangle \longrightarrow^{k-1} \langle h_{k-1}; \mathcal{A}_{k-1}; \Delta_{k-1}; f_{k-1} \rangle$$
$$\mathcal{F} \vdash \langle h; \mathcal{A}; \Delta'; f \rangle \longrightarrow^{k-1} \left\langle h_{k-1}; \mathcal{A} \cup \mathcal{A}_{k-1}; \Delta'_{k-1}; f_{k-1} \right\rangle$$

with $\Delta'_{k-1}(m) = \begin{cases} \Delta_{k-1}(m) & m \in \left(\mathcal{A}_{k-1}^{\mathcal{U}} \cup \mathcal{A}_{k-1}^{\mathcal{C}}\right) \\ \Delta'(m) & \text{otherwise} \end{cases}$

Then suppose we take one additional step of reduction:

$$\mathcal{F} \vdash \langle h_{k-1}; \mathcal{A}_{k-1}; \Delta_{k-1}; f_{k-1} \rangle \longrightarrow \langle h_k; \mathcal{A}_k; \Delta_k; f_k \rangle$$
$$\mathcal{F} \vdash \left\langle h_{k-1}; \mathcal{A} \cup \mathcal{A}_{k-1}; \Delta'_{k-1}; f_{k-1} \right\rangle \longrightarrow \langle h''; \mathcal{A}''; \Delta''; f'' \rangle$$

and consider the possible cases. If f is a sequence, then we apply [SEQ], which ultimately reduces to one of the other cases. Since the output accumulator and defines are the same as from the underlying statement reduction, there is nothing additional to show. We consider the other cases.

- ifdef. Suppose that f is an ifdef conditioned on m. In either case, we clearly have $\mathcal{A}'' = (\mathcal{A} \cup \mathcal{A}_{k-1})[\mathcal{U} \leftarrow^+ m] = \mathcal{A} \cup (\mathcal{A}_{k-1}[\mathcal{U} \leftarrow^+ m]) = \mathcal{A} \cup \mathcal{A}_k$. Since the set of defines does not change, our property on Δ'_k holds. We also clearly have $h'' = h_{k-1} = h_k$.

  Then there are two cases. If $m \in \mathcal{A}_{k-1}^{\mathcal{C}}$, then by induction we have $\Delta'_{k-1}(m) = \Delta_{k-1}(m)$, and therefore we clearly have $f'' = f_k$.

Otherwise if $m \notin \mathcal{A}^{\mathcal{C}}_{k-1}$, by Lemma A.10 we have $\Delta_{k-1}(m) = \Delta(m)$. Then since $m \in \mathcal{A}^{\mathcal{U}}_k$, by assumption we have $\Delta'(m) = \Delta(m)$, and we also have $m \notin \mathcal{A}^{\mathcal{C}}$. But then $m \notin (\mathcal{A} \cup \mathcal{A}_{k-1})^{\mathcal{C}}$, and thus by Lemma A.10 we have $\Delta'_{k-1}(m) = \Delta'(m)$. Putting this together, we have $\Delta_{k-1}(m) = \Delta(m) = \Delta'(m) = \Delta'_{k-1}(m)$. Thus we clearly have $f'' = f_k$.

Finally, observe that $\mathcal{A}^{\mathcal{U}}_k = \mathcal{A}^{\mathcal{U}}_{k-1} \cup \{m\}$, and by induction $\Delta'_{k-1}(m) = \Delta_{k-1}(m)$ for $m \in \left(\mathcal{A}^{\mathcal{U}}_{k-1} \cup \mathcal{A}^{\mathcal{C}}_{k-1}\right)$, and $\Delta'_{k-1}(m) = \Delta'(m)$ otherwise. But we have just argued above that $\Delta'_{k-1}(m) = \Delta_{k-1}(m)$, and $m \in \mathcal{A}^{\mathcal{U}}_k$ by [IfDef+] or [IfDef-]. And since $\Delta_k = \Delta_{k-1}$ and $\Delta'_k = \Delta'_{k-1}$, we have $\Delta'_k(m) = \Delta_k(m)$ for $m \in \left(\mathcal{A}^{\mathcal{U}}_k \cup \mathcal{A}^{\mathcal{C}}_k\right)$, and $\Delta'_k(m) = \Delta'(m)$ otherwise.

- includes. By the assumption the reduction of f under $\mathcal{A}$ never gets stuck, so both reductions can take a step, and trivially both produce the same accumulator since it is simply added to.
- extern, let, type, and let type. Trivial, since the accumulator is simply added to and the defines are not changed.
- def. Clearly we have $\mathcal{A}'' = \mathcal{A} \cup \mathcal{A}_k$ by applying induction and observing that [Def] only adds to the macro uses and changes in the accumulator, and clearly $f'' = f_k$. We also clearly have $h'' = h_k$, and we have $\mathcal{A}^{\mathcal{U}}_k \cup \mathcal{A}^{\mathcal{C}}_k = \mathcal{A}^{\mathcal{U}}_{k-1} \cup \mathcal{A}^{\mathcal{C}}_{k-1} \cup \{m\}$. By induction we have $\Delta'_{k-1}(m) = \Delta_{k-1}(m)$ for $m \in \left(\mathcal{A}^{\mathcal{U}}_{k-1} \cup \mathcal{A}^{\mathcal{C}}_{k-1}\right)$, and $\Delta'_{k-1}(m) = \Delta'(m)$ otherwise. Further, $\Delta_k(m) = \Delta_{k-1}(m) = \mathit{true}$ by [Def]. Thus we have $\Delta'_k(m) = \Delta_k(m)$ for $m \in \left(\mathcal{A}^{\mathcal{U}}_k \cup \mathcal{A}^{\mathcal{C}}_k\right)$, and $\Delta'_k(m) = \Delta'(m)$ otherwise.
- undef. Similar to previous case.
- pop. Trivial, since clearly $h'' = h_k$, because they are set to the same value by [Eoh].

□

Given this lemma, we can now show that, assuming order-independence, the placement of an include file does not affect its behavior.

Lemma A.12 (Separate Reduction) *Suppose preprocessing f evaluates the statement $s =$ include h:*

$$\mathcal{F} \vdash \langle \cdot; \mathcal{A}_\emptyset; \Delta; f \rangle \longrightarrow^* \langle h_1; \mathcal{A}_1; \Delta_1; (s, f_2) \rangle \tag{A.1}$$

$$\mathcal{F} \vdash \langle h_1; \mathcal{A}_1; \Delta_1; (s, f_2) \rangle \longrightarrow^* \langle h_1; \mathcal{A}_2; \Delta_2; f_2 \rangle \tag{A.2}$$

Also assume that $\mathcal{F}(h)$ can be separately preprocessed:

$$\mathcal{F} \vdash \langle \cdot; \mathcal{A}_\emptyset; \Delta; \mathcal{F}(h) \rangle \longrightarrow^* \langle \cdot; \mathcal{A}_h; \Delta_h; \cdot \rangle \tag{A.3}$$

Then if $\Delta; \mathcal{F} \vdash \mathcal{R}_3(f)$, it is the case that $\mathcal{A}^N_h \subseteq \mathcal{A}^N_2$.

Proof Expanding out (A.2), we have

$$\begin{aligned} \mathcal{F} \vdash \langle h_1; \mathcal{A}_1; \Delta_1; (s, f_2) \rangle \longrightarrow \\ \left\langle h; \mathcal{A}_1[\mathcal{I} \leftarrow^+ h]; \Delta_1; (\mathcal{F}(h), \mathsf{pop}\ h_1, f_2) \right\rangle \\ \mathcal{F} \vdash \left\langle h; \mathcal{A}_1[\mathcal{I} \leftarrow^+ h]; \Delta_1; (\mathcal{F}(h), \mathsf{pop}\ h_1, f_2) \right\rangle \longrightarrow^* \\ \langle h_1; \mathcal{A}_2; \Delta_2; f_2 \rangle \end{aligned} \tag{A.4}$$

Then because $\Delta; \mathcal{F} \vdash \mathcal{R}_3(f)$ and $h \in \mathcal{A}^{\mathcal{I}}$, we have $\Delta; \mathcal{F} \vdash f \otimes h$. Therefore by [Partial-Indep], we have $(\mathcal{A}_1[\mathcal{I} \leftarrow^+ h])^{\mathcal{C}} \cap \mathcal{A}^{\mathcal{U}}_h = \emptyset$. Then by Lemma A.10, we have $\Delta_1(m) = \Delta(m)$ for all $m \in \mathcal{A}^{\mathcal{U}}_h$. Moreover, by (A.4), we know the reduction of $\mathcal{F}(h)$ did not get stuck using accumulator $\mathcal{A}_1[\mathcal{I} \leftarrow^+ h]$, macro environment Δ_1, and current header file h. Now suppose we have

$$\mathcal{F} \vdash \langle h; \mathcal{A}_\emptyset; \Delta; \mathcal{F}(h) \rangle \longrightarrow^* \langle h; \mathcal{A}_{h*}; \Delta_{h*}; \cdot \rangle \tag{A.5}$$

Notice that by (A.3), this reduction cannot get stuck. Then we can apply the Passthrough Property (Lemma A.11) on (A.5) and (A.4):

$$\mathcal{F} \vdash \langle h; \mathcal{A}_1[\mathcal{I} \leftarrow^+ h]; \Delta_1; \mathcal{F}(h) \rangle \longrightarrow^* \langle h; \mathcal{A}_1[\mathcal{I} \leftarrow^+ h] \cup \mathcal{A}_{h*}; \Delta'_{h*}; \cdot \rangle$$

But then by (A.4), we have $\mathcal{A}_2 = \mathcal{A}_1[\mathcal{I} \leftarrow^+ h] \cup \mathcal{A}_{h*}$. But observe that $\mathcal{A}^N_h = \mathcal{A}^N_{h*}$ (changing the current include file doesn't change the symbol types, by the rules in Figure 5), and thus $\mathcal{A}^N_h \subseteq \mathcal{A}^N_2$. □

Given this lemma, we can now show that if two fragments satisfy the conditions of [Rule 1] and the commonly-included header is order-independent, then any symbol exported by one and imported by the other has the same type in both.

Lemma A.13 (Consistent Typing) *Let fragment f_e export g and let fragment f_i import g, and let both f_e and f_i include a common header fragment $f_h (= \mathcal{F}[h])$ that declares the variable:*

$$\begin{array}{lll} \Delta; \mathcal{F} \vdash f_e \rightsquigarrow \mathcal{A}_e, & g \in \mathcal{A}^E_e, & h \in \mathcal{A}^{\mathcal{I}}_e \\ \Delta; \mathcal{F} \vdash f_i \rightsquigarrow \mathcal{A}_i, & g \in \mathcal{A}^I_i, & h \in \mathcal{A}^{\mathcal{I}}_i \\ \Delta; \mathcal{F} \vdash f_h \rightsquigarrow \mathcal{A}_h, & g \in \mathcal{A}^D_h & \end{array}$$

Then if

$$\begin{array}{ll} \Delta; \mathcal{F} \vdash f_e \xrightarrow{comp} O_e & \Delta; \mathcal{F} \vdash \mathcal{R}_3(f_e) \\ \Delta; \mathcal{F} \vdash f_i \xrightarrow{comp} O_i & \Delta; \mathcal{F} \vdash \mathcal{R}_3(f_i) \end{array}$$

all hold, then $\mathcal{A}^N_h(g) = \mathcal{A}^N_e(g) = \mathcal{A}^N_i(g)$, i.e., g maps to the same type in each fragment.

Proof By assumption, preprocessing f_e and f_i will eventually preprocess the statement $s = \mathsf{include}\ h$. Thus we have:

$$\begin{array}{l} \mathcal{F} \vdash \langle \cdot; \mathcal{A}_\emptyset; \Delta; f_e \rangle \longrightarrow^* \langle h_{1e}; \mathcal{A}_{1e}; \Delta_{1e}; (s, f_{2e}) \rangle \\ \mathcal{F} \vdash \langle h_{1e}; \mathcal{A}_{1e}; \Delta_{1e}; (s, f_2) \rangle \longrightarrow^* \langle h_{1e}; \mathcal{A}_{2e}; \Delta_{2e}; f_{2e} \rangle \\ \mathcal{A}_{2e} \subseteq \mathcal{A}_e \end{array}$$

Then by Lemma A.12, we have $\mathcal{A}^N_h \subseteq \mathcal{A}^N_{2e}$. But then since $\mathcal{A}^N_{2e} \subseteq \mathcal{A}^N_e$, we have $\big(g \mapsto \mathcal{A}^N_h(g)\big) \in \mathcal{A}^N_e$. The by [Compile], we have $\vdash \mathcal{A}^N_e$, and therefore $\mathcal{A}^N_e(g) = \mathcal{A}^N_h(g)$. Similarly we can show that $\mathcal{A}^N_i(g) = \mathcal{A}^N_h(g)$, because f_i included the same file. □

Theorem A.14 (Type-Safe Linking) *Suppose $\Delta; \mathcal{F} \vdash \mathcal{R}(\mathcal{P})$, and suppose $\Delta; \mathcal{F} \vdash \mathcal{P} \xrightarrow{comp} [\emptyset \Rightarrow H_{\mathcal{P}} : \Psi_{E\mathcal{P}}]$. Also suppose that for any $f_i, f_j \in \mathcal{P}$ that are distinct ($i \neq j$), it is the case that*

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

Then

$$\vdash [\Psi_{Ii} \Rightarrow H_i : \Psi_{Ei}]\ \text{link}\ [\Psi_{Ij} \Rightarrow H_j : \Psi_{Ej}] \rightsquigarrow O_{ij}$$

Proof By assumption [Link] holds. Observe that the linked file form (O_{ij}) in [Link] is the same as in [MTAL$_0$-Obj], so we just need to show the hypotheses of this rule. To show [MTAL$_0$-Obj], we first need to show each object file is well-formed, which follows by Lemma A.7. The last premise of [MTAL$_0$-Obj], disjointness of the domains of H_i and H_j, is the same as the premise of [Link], so that also holds. Thus we only need to show link compatibility, or $\vdash [\Psi_{Ii} \Rightarrow H_i : \Psi_{Ei}] \overset{\text{lc}}{\leftrightarrow} [\Psi_{Ij} \Rightarrow H_j : \Psi_{Ej}]$. We show each premise of [MTAL$_0$-LC] in turn.

- $\mathrm{dom}\,(\Psi_{Ei}) \cap \mathrm{dom}\,(\Psi_{Ej}) = \emptyset$. Notice $\mathrm{dom}\,(\Psi_{Ei}) = E_i$ by [Compile] and $E_i \subseteq \mathrm{dom}\,(H_i)$, which we observe holds because the rules in Figure 5 only add symbols to E that are also added to $\mathrm{dom}\,(H)$ (see rule [Let]). Similarly, $\mathrm{dom}\,(\Psi_{Ej}) = E_j \subseteq \mathrm{dom}\,(H_j)$. Then since by [Link] we have $\mathrm{dom}\,(H_i) \cap \mathrm{dom}\,(H_j) = \emptyset$, we have $\mathrm{dom}\,(\Psi_{Ei}) \cap \mathrm{dom}\,(\Psi_{Ej}) = \emptyset$.

- $\vdash \Psi_{Ii} \sim \Psi_{Ej}$. By assumption, we have $\Delta; \mathcal{F} \vdash \mathcal{R}(\mathcal{P})$. Thus by [All], we have in particular

$$\Delta; \mathcal{F} \vdash \mathcal{R}_1(f_i, f_j)$$
$$\Delta; \mathcal{F} \vdash \mathcal{R}_3(f_i) \quad \Delta; \mathcal{F} \vdash \mathcal{R}_3(f_j)$$

  Let $\mathcal{A}_i$ and $\mathcal{A}_j$ are the accumulators from the preprocessing of f_i and f_j, respectively. Now consider some g in $\mathrm{dom}\,(\Psi_{Ii}) \cap \mathrm{dom}\,(\Psi_{Ej})$. Then we have $g \in \mathcal{A}_i^I$ and $g \in \mathcal{A}_j^E$. Further, by [Rule 1] we have $\Delta; \mathcal{F} \vdash g \xleftarrow{\text{decl}} \mathcal{A}_i^{\mathcal{I}} \cap \mathcal{A}_j^{\mathcal{I}}$. Then by [Sym-Decl] there exists some $h \in \mathcal{A}_i^{\mathcal{I}} \cap \mathcal{A}_j^{\mathcal{I}}$ such that $\Delta; \mathcal{F} \vdash \mathcal{F}(h) \rightsquigarrow \mathcal{A}$ and $g \in \mathcal{A}^D$. Then we can apply Lemma A.13 to yield $\mathcal{A}_i^N(g) = \mathcal{A}_j^N(g)$. But then we have $\Psi_{Ii}(g) = \Psi_{Ej}(g)$ by [Compile], and therefore $\vdash \Psi_{Ii} \sim \Psi_{Ej}$ holds.

- $\vdash \Psi_{Ij} \sim \Psi_{Ei}$. Symmetric argument to the previous case.

- $\vdash \Psi_{Ii} \sim \Psi_{Ij}$. Let us consider some $g \in \mathrm{dom}\,(\Psi_{Ii}) \cap \mathrm{dom}\,(\Psi_{Ij})$. Then we assume that $\exists f_k \in \mathcal{P}$ s.t. $\Delta; \mathcal{F} \vdash f_k \xrightarrow{\text{comp}} [\Psi_{Ik} \Rightarrow H_k : \Psi_{Ek}]$ and $g \in E_k$, i.e., we assume that some fragment exports the symbol g, because we assumed the fully-compiled program had no unresolved symbols. Then by the same argument as the previous two cases we can conclude $\vdash \Psi_{Ii} \sim \Psi_{Ek}$ and $\vdash \Psi_{Ij} \sim \Psi_{Ek}$, since the Cmod rules hold for the whole program. Then we have $\Psi_{Ek}(g) = \Psi_{Ii}(g)$ and $\Psi_{Ek}(g) = \Psi_{Ij}(g)$, and therefore $\Psi_{Ii}(g) = \Psi_{Ij}(g)$.

□