Defining and Enforcing C's Module System

Saurabh Srivastava Michael Hicks Jeffrey S. Foster Bhargav Kanagal
University of Maryland, College Park
{saurabhs,mwh,jfoster,bhargav}@cs.umd.edu

Abstract

Programming language module systems are an important tool for managing the complexity of large software systems. The C programming language is used to build many such systems, yet it lacks a proper module system. Instead, C programmers typically follow a convention that treats `.h` header files as interfaces, and `.c` source files as modules. This convention can be effective, but there are many subtleties in using it correctly, and the compiler and linker provide no enforcement mechanism. As a result, misuse of the convention can lead to hard-to-find bugs, can make reasoning about code in isolation more difficult, and can complicate future code maintenance.

This paper presents CMOD, a module system for C that ensures type-safe separate compilation and information hiding. Our approach is to identify and enforce the circumstances under which C's current modular programming convention is sound. The result is four rules that, using an operational semantics for an idealized preprocessor, we have proven are sufficient to guarantee the desired modularity properties. We have implemented CMOD for the full C language and applied it to a number of benchmarks. We found that most of the time legacy programs obey CMOD's rules, or can be made to with minimal effort, and rule violations often result in type errors or brittle code. Thus CMOD brings the benefits of modular programming to C while still supporting legacy systems.

1. Introduction

Programming language module systems allow programmers to construct large programs from smaller, potentially-reusable components that can be written and reasoned about in isolation [20]. Informally, a module is a collection of programming language definitions—for functions, types, variables, etc.—and a program is a collection of modules that are linked together. Most module systems support some level of namespace management, separate compilation, and information hiding. The first is usually achieved via a module-level naming hierarchy. The latter two are usually achieved through the use of *interfaces* (or *signatures*), which describe the externally-visible definitions of a module. Interfaces can safely stand in for the module while separately compiling its clients, ensuring the program will be type-correct when it is linked.

Modern languages, such as ML [3, 17], Haskell [15], Ada [1], and Modula-3 [11] have full-featured module systems. In contrast, the C programming language—still the most common language for building operating systems, network servers, and other critical infrastructure—lacks an explicit module system. Instead, a convention has evolved over time in which programmers informally treat `.c` source files as modules and `.h` header files as interface files. Header files typically contain type and variable declarations, so that when they are textually included into a client source file, via the preprocessor directive `#include`, the compatibility of these declarations with their uses is enforced by the core language compiler.

Unfortunately, the compiler and linker do not enforce that this convention is used safely, which can lead to bugs and complicate software maintenance. For example, a source file can import a type or value definition directly, e.g., via an `extern` declaration, even if the corresponding definition is not explicitly exported in a header file. This violates information hiding and could also violate type safety, since standard linking is by name only and does no type checking across linked files. For that matter, a source file may neglect to include its own header, allowing an implementation to disagree with its interface. Preprocessing complicates matters further, because a program may conditionally process program text depending on whether a preprocessor macro has been defined. As a result, preprocessor definitions from one included header file may inadvertently influence header files that are included later.

This paper presents a module system for C called CMOD that is rooted in C programming practice. Our observation is that standard C practice is almost safe, but there are many subtleties in using it correctly. Thus the core of CMOD is four rules that, when followed, forbid undesirable programming practice and ensure that it is sound to reason about header files as interfaces and compilation units as modules. As a result, CMOD ensures programs are type-safe when linked and properly enforce information hiding. The primary CMOD rule requires that module imports and exports are mediated by interfaces. More precisely, when one file uses a symbol defined in another file, both files must include a common header file declaring that symbol. This rule ensures that symbols are consistently typed everywhere and that non-exported symbols are private. Two additional CMOD rules ensure that, even with preprocessing, top-level header files are treated uniformly across all compilations units, and hence are safe to interpret as interfaces. Finally, the last rule ensures that all compilation units agree on definitions of the same type name, and that type abstraction is not violated.

We have formalized CMOD's rules and proven that they are correct. Our formalism uses an operational semantics for an idealized preprocessor that allows fragments of C code to be conditionally included and allows one file to be included in another. The four rules are specified in terms of instrumented preprocessor output. Using a specification of link-time type safety based on MTAL [10], we prove that if our rules are followed, then the program will be type safe at link time. We also prove that information hiding policies implied by interfaces are properly enforced.

We have built an implementation of CMOD and applied it to a range of C programs **TODO:** comprising XXX lines of code. We found that programs generally comply with CMOD's rules, indicat-

ing that programmers tend to safely use the standard modularity convention. We examined the cases where CMOD's rules are violated and found several typing errors along with many cases that, although not currently bugs, make future programming mistakes more likely as the code evolves. We also found that it is usually easy to change programs to comply with CMOD.

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

- We present CMOD, a set of four rules that makes it safe to treat header files as interfaces and source files as modules (Section 2). While this work focuses on C, we believe that CMOD may also be useful for languages that make use of the same modularity convention, such as C++, Objective C, and Cyclone [12].
- We specify our rules formally in the context of an operational semantics for the C preprocessor. We prove that our rules are sound, meaning that programs that obey them are type safe at link time and enforce information hiding (Section 3).
- We present an implementation of CMOD (Section 4) and describe the results of applying it to a small set of benchmarks (Section 5).

Thus CMOD brings the benefits of modular programming to C while supporting legacy systems.

2. Overview

Typically, a module M consists of two parts: an *interface* M_I that declares the names and types of the visible or *exported* values in the module, and an *implementation* M_S that defines all the values and types in M_I, plus possibly other, private definitions. Any component C that wishes to use values in M_S relies only on the information in M_I during compilation, and does not examine M_S directly. The compiler separately ensures that M_S *implements* M_I, meaning that it exports any types and symbols in the interface.

The key idea is that an interface *abstracts* a module's possible implementations, so that clients will not depend on one implementation in particular:

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 τ, no module other than M_S may use values of type τ 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 *any* module that implements that M_I. Moreover, this linkage will 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 will be type safe.*

The main goal of CMOD is to support modular programming in C that satisfies these two properties.

2.1 Basic Modules in C

The notions of interface and implementation are used by most C programmers, albeit informally. In C programming style, it is customary to treat `.h` header files as interface files, and `.c` source files as implementation files. For example, in Figure 1, `bitmap.h` is the interface to source file `bitmap.c`, whose functions are called by `main.c`. In order to use the types and values in `bitmap.h`, the file `main.c` imports it with the command `#include "bitmap.h"`, which the preprocessor recognizes and textually replaces with the contents of file `bitmap.h`.

This coding convention is *almost* enough to ensure Properties 2.1 and 2.2. First, `main.c` will fail to compile if `main` refers

bitmap.h

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

bitmap.c

```
1  #include "bitmap.h"
2
3  void init(BM *map) { ... }
4  void set(BM *map, int bit) { ... }
```

main.c

```
1  #include "bitmap.h"
2
3  int main(void) {
4    BM bitmap;
5    init(&bitmap);
6    set(&bitmap, 1);
7    ...
8  }
```

Figure 1: Basic C Modules

to symbols not present in `bitmap.h`, thus enforcing information hiding. Second, the normal C compilation rules ensure that `main.c` uses `BM`, `set`, and `init` at the same types given in `bitmap.h`. Likewise, because the file `bitmap.c` contains `#include "bitmap.h"` we know that it effectively implements interface `bitmap.h`, since the C compiler will complain if the types in `bitmap.h` and `bitmap.c` do not match. Thus we know that linking the two modules together will be type safe.

Unfortunately, as we show in the remainder of this section, this basic convention is neither sufficient nor necessary to ensure Properties 2.1 and 2.2. Therefore, we defined CMOD as a series of rules that tighten this convention so that the modularity properties hold. The rules aim to adhere as closely as possible to C programming practice while still enforcing a modular programming discipline. In this section we describe the rules and their design rationale, with examples. Section 3 presents a formal semantics and rule specification, and Section 4 describes our implementation.

2.2 Header Files as Interfaces

We begin by considering the core idea that `.h` files should be used as in Figure 1: clients should treat them as interfaces, and providers should use them to enforce that a module implements its interface. In C, this convention can be violated in two ways.

First, clients can ignore headers and use local `extern` declarations instead. For example, Figure 2 shows a modification of Figure 1 in which `main.c` does not include `bitmap.h`, but instead locally declares the imported functions, in this case getting the type on `init` wrong. Second, a module can fail to `#include` its own interface file, and thus fail to implement the interface correctly. In Figure 2, `bitmap.h` and `bitmap.c` do not agree on the type of `BM`, so even if `main.c` did include `bitmap.h` there would still be a mismatch. In both cases, because C does not check types at link time, a C compiler will happily compile and link `main.c` and `bitmap.c` together, causing strange errors at run-time.

We can prevent these problems by ensuring that an appropriate header file always mediates linking, so that the client and provider agree on a symbol's type. Thus we propose the first CMOD rule:

RULE 1. *Whenever one file links to a symbol defined by another file, both files must include a header that contains the type of that symbol.*

This rule enforces the convention from Figure 1 that each module has its own interface that both it and its clients include. But it is also more flexible. For example, programmers frequently use one header as an interface for several source files, e.g., the C standard library header `stdio.h` often covers many different `.c` files. To adhere to Rule 1, each `.c` file would `#include "stdio.h"`. Another convention is to have several header files for a single source file, to provide "public" and "private" views of the module. In this case the

bitmap.h

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

bitmap.c

```
1  typedef int* BM; /* ! */
2  void init (BM *map, int)
3  { ... }
4  void set(BM *map, int bit)
5  { ... }
```

main.c

```
1  typedef int BM;
2  extern void init (BM *); /* ! */
3  extern void set(BM *, int);
4  int main(void) {
5    BM bitmap;
6    init (&bitmap);
7    set(&bitmap, 1);
8    ...
9  }
```

Figure 2: Incorrect Uses of Header Files (Violates Rule 1)

bitmap.h

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

bitmap.c

```
1  #include "config.h"
2  #include "bitmap.h"
3
4  struct BM { int data; };
5  void set(struct BM *map, int bit) { ... }
6  void init (struct BM *map) { ... }
```

client.c

```
1  #include "config.h"
2  #include "bitmap.h"
3
4  /* type error: */
5  struct BM { int *data };
6  void foo(void) {c
7    struct BM map;
8    init (&map);
9    ... /* use of map */ ...
10 }
```

Figure 3: Information Hiding and Abstract Types

source file would include *both* headers, while clients would include one or the other.

All but one of the type errors we found in our experiments (Section 5) were due to violations of Rule 1.

2.3 Types in Interfaces

Rule 1 ensures that function and variable definitions can be hidden by a header, and when shared are always viewed at consistent types. We need to ensure the same thing for named type definitions.

Consider the code in Figure 3. In this case, `bitmap.h` does not reveal the representation of `struct BM` to its clients. The source file `bitmap.c` contains one definition, and the file `client.c` contains another definition. This illustrates two problems. Most egregiously, the two definitions disagree, which is a type error. However, even if they did agree, `struct BM` should really be treated abstractly by the client, so that `bitmap.c` is free to change its implementation. Thus `client.c` violates information hiding.

With terms, only one module can define a variable, and other modules link against that definition; if two modules were to define the same symbol, then linking would fail. With types, however, the linker will not complain if two modules define the same type, allowing the problems in Figure 3. To solve these problems, we treat types as the linker does terms: there should only be one definition of a type that all modules "link against."

For types, a definition can occur in one of two places. If the type is *transparent*, then many modules can know its definition. In this case, we can place the definition in a header file, and "linking" occurs by including the header, rather than via symbol resolution as for terms. As an example, the standard definition of `size_t` is in `stddef.h`. Conversely, if the type is *abstract*, then only the module that implements the type's functions should know its definition. In this case, we place the type definition in the module's source file, and not in any header file. Between these two policies is a middle ground: we can define two header files for a given module, an "internal" one that reveals a type's definition and an "external" one that does not. Modules that implement the type (and its functions) include both headers, and those that use it abstractly include only the external one.

bitmap.h

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

bitmap.c

```
1  #include "config.h"
2  #include "bitmap.h"
3
4  #ifdef COMPACT
5  void init (BM *map)
6  { ... }
7  void set(BM *map, int bit)
8  { ... }
9  #else
10 void set(BM *map, int bit)
11 { ... }
12 void init (BM *map)
13 { ... }
14 #endif
```

config.h

```
1  #ifndef _CONFIG_H
2  #define _CONFIG_H
3  #ifdef __BSD__
4   #undef COMPACT
5  #else
6   #define COMPACT
7  #endif
8  #endif
```

main.c

```
1  #include "config.h"
2  #include "bitmap.h"
3
4  int main(void) {
5    BM bitmap;
6    init (&bitmap);
7    set(&bitmap, 1);
8    ...
9  }
```

Figure 4: Using the Preprocessor for Configuration

We can view these approaches as *ownership*: for transparent types the header owns the type definition, while for abstract types the source file owns it. Thus, CMOD's second rule:

RULE 2. *Each type definition in the linked program must be owned by exactly one source or header file.*

This rule forbids the program in Figure 3, which contains two definitions of type `struct BM`. The other type error we found in our experiments was due to a violation of this Rule 2.

This notion of ownership assumes a global namespace, in which type names and variables have a single meaning throughout a program. For variables, this namespace is actually more relaxed: two modules *can* define the same symbol as long as they declare it as `static`, which ensures there are no linking conflicts. While we could define certain notions of `static` for types, we use this stronger rule because it is simple to implement, and we have found that programmers generally follow this practice.

2.4 Preprocessing and Header Files

The first two rules form the core of the modular programming convention enforced by CMOD. However, for these rules to work properly, we must properly account for the actions of the preprocessor.

When a header file is textually inserted into a source file using `#include`, the header's meaning could be affected by the program text that precedes it, while the included header could itself affect the meaning of subsequently-included headers. This situation arises due to the preprocessor's use of macros and conditional commands, such as `#ifdef` and `#ifndef`.

To see why, consider the code shown in Figure 4, which extends our example from Figure 1 to represent bitmaps in one of two ways,

depending on whether the `COMPACT` macro (set in `config.h`) has been previously defined.[1] In the figure, `config.h` affects the meaning of `bitmap.h`, because the latter branches on macro `COMPACT` which can be set or unset by the former. This means that the order the files are included matters: if we were to reverse lines 1–2 in `main.c`, then compiling `main.c` and `bitmap.c` under the same initial preprocessor environment would produce a type-incorrect program following linking.

When the behavior of a pair of headers depends on the order in which they are included during preprocessing, we call these headers *order dependent*. We say that two header files are *vertically dependent* if, as with `config.h` and `bitmap.h` in the example, the dependence arises based on the order the programmer `#includes` them in a source file. On the other hand, we say that two header files are *horizontally dependent* if one's behavior is affected when it includes the other. We could convert the vertical dependences in the example to be horizontal by moving line 1 of `bitmap.c` (that is, `#include "config.h"`) to just prior to line 1 in `bitmap.h`. Thus the contents of `bitmap.h` now depend on a file it includes.

Vertical dependences are generally bad coding style because they complicate local reasoning about interfaces and the modules they represent, and as shown can undermine type safety. Avoiding vertical dependence improves reuse because it allows the programmer to include a header file anywhere among the imports of an implementation and know that it is being interpreted consistently. On the other hand, horizontal dependence is generally good programming practice because it makes a header "self-contained" so that it can be understood in isolation. Thus we could swap lines 1–2 in `main.c` without affecting the meaning of the module. Notice, however, that with a horizontally-dependent `bitmap.h`, the file `config.h` is included twice in `main.c`, once directly and once via `bitmap.h`. The duplicate inclusion is harmless because of the `#ifndef _CONFIG_H` directive that causes any duplicate file inclusions to be completely ignored. Our formal semantics assumes this behavior, and we check that it holds for our benchmarks.

While vertical dependences are generally undesirable, the vertical dependence shown in the example is nearly ubiquitous in large C programs: source files often include a single `config.h` file to configure the software. We can allow such a file, as long as it is included *first* by every linked source file, ensuring its effects will be uniform. This is not a terrible burden on the programmer, as when interpreting the meaning of a header, he can always keep in mind the effects of the system `config.h`. Thus we propose the third CMOD rule:

RULE 3. *With the exception of a designated, initial* `config.h`*, header file inclusion must be order-independent.*

Finally, while macros are often set by `#define` and `#undef` commands in the code, they can also be set by the system or according to `-D` options passed to the compiler on the command line. Therefore, we must make sure that when linking together two files that they have been compiled in compatible macro environments. In our example, the data representation used by both `bitmap.c` and `main.c` in Figure 4 depends on whether `__BSD__` has been set, and thus depends on whether `-D__BSD__` has been passed as an argument to the compiler. If the programmer compiles one file with this option and one file without, both files will compile without warning and, though they adhere to the all of CMOD's rules so far, the linked program will be type-unsafe.

Thus we propose the final CMOD rule, which simply requires that the configuration options are the same:

[1] The value of `COMPACT` itself depends on the value of `__BSD__`, which is not set anywhere in the source code. We will return to this point for Rule 4.

program	$\mathcal{P}$	::=	$\cdot \mid f \circ \mathcal{P}$
fragment	f	:=	$\cdot \mid s, f$
statements	s	::=	$c \mid d$
preproc. commands	c	::=	$\mathsf{include}\ h$
		$\mid$	$\mathsf{def}\ m$
		$\mid$	$\mathsf{undef}\ m$
		$\mid$	$\mathsf{ifdef}\ m\ \mathsf{then}\ f\ \mathsf{else}\ f$
definitions	d	::=	$\mathsf{let}\ g : \tau = e$
		$\mid$	$\mathsf{extern}\ g : \tau$
		$\mid$	$\mathsf{let\ type}\ t = \tau$
		$\mid$	$\mathsf{type}\ t$
terms	e	::=	$n \mid \lambda y : \tau.\ e \mid e\ e \mid y \mid g$
types	τ	::=	$t \mid \mathsf{int} \mid \tau \rightarrow \tau$
	m	$\in$	macro names
	h	$\in$	file names
	g	$\in$	global variable names
	t	$\in$	type names

Figure 5: Source Language

RULE 4. *All files linked together must have been compiled in the same preprocessor environment.*

TODO: Here's where we handwave a bit about "compatible" environments rather than requiring them to be the same.

The core contribution of CMOD is these four rules. While C programmers understand C's basic modularity convention, many do not understand the subtleties of using it safely (as the experimental results of Section 5 illustrate). Ours is the first work to formalize rules, and to write a tool that checks them automatically, that ensure safe modular programming in C in keeping with accepted practice.

3. Formalism

This section presents the syntax of an idealized source language and the semantics of the preprocessor on this language. We define the CMOD rules in terms of the output of the preprocessor and show that they are sufficient to prove type safety and information hiding.

3.1 Syntax

Figure 5 shows our source language. A source program $\mathcal{P}$ consists of a list of fragments f, which represent modules. Fragments are 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 at the position of the include command and then preprocesses it. In our semantics we will assume we are given a mapping from include filenames to fragments. The commands $\mathsf{def}\ m$ and $\mathsf{undef}\ m$ define and undefine, respectively, the boolean preprocessor macro m from that point forward. 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 additional preprocessor commands.

The C preprocessor includes additional features not found in our language, including macro functions 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. We discuss these issues in Section 4.

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. The command $\mathsf{extern}\ g : \tau$ declares the existence of global g of type τ. This form is used in header files to

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

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

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

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

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

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

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

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

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

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

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

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

Figure 6: Operational Semantics for Preprocessor

import a symbol. The command let type $t = \tau$ defines a named type t to be an alias for τ, while type t merely declares that t may be used as a type name. We say that g and t are *defined* by let g and let type $t = \tau$, respectively, while g and t are *declared* by extern g and type t, respectively. Within a program we allow many declarations of a global variable but only one definition. Within a file, if a type name has been defined then it is *transparent*, and otherwise it is *abstract*.

3.2 Preprocessor Semantics

Following C, our source language has a two-level 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, as shown in Figure 6. 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 the rules described in Section 2. In particular, the preprocessor is defined as a relation between *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.

The program starts with an initial (possibly empty) set of defined macros, Δ, which in practice is supplied on the command line when the compiler is invoked. Δ may change as preprocessing progresses. 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 τ are annotated with either the header file h in which the type was defined, or $\cdot$ 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 that have been declared Z; 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 include file names to their corresponding fragments. Preprocessing begins with an accumulator whose components are all $\emptyset$, which we write $\mathcal{A}_\emptyset$. We also start with the h state component set to $\cdot$, and assume we are given the file system mapping $\mathcal{F}$ and an initial set of defines Δ.

In the rules in Figure 6, 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] is standard. We abuse notation and write f', f as the concatenation of fragments f' and f (where $\cdot, f' = f'$ and $(s, f'), f'' = s, (f', f'')$). [INCLUDE] looks up file name h in the file system and reduces to the corresponding fragment. It also inserts a special command pop $h'; \mathcal{A}$ 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 [INCLUDE] cannot make progress. While nonstandard, this semantics simplifies the soundness proof. In practice, programmers use preprocessor directives to make duplicate file inclusion a no-op, which provides the same result as if it never occurred. Our implementation checks to be sure this practice is followed.

[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, described in Section 3.4, ensures that the same variable is always given the same type within a fragment. [LET] adds g to the set of exported global variables, adds g's type to N, and adds any global variables mentioned in e (written $\mathrm{fg}(e)$) to the imports. Note that to keep the rules simpler, we do not allow `static` definitions, which in C introduce file-scoped names that can be reused across different files. Instead we simply assume names that are not part of interfaces are alpha-converted to be different across different fragments. Finally, [TYPE-DECL] declares a type, which the preprocessor notes 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 $\cdot$ if the current file is not a header.

3.3 CMOD Rules

Given the operational semantics, we can now formally specify the rules presented in Section 2. In order 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* $\Delta; \mathcal{F} \vdash f_0 \rightsquigarrow \langle \mathcal{A}; f_1 \rangle$ *as shorthand for* $\mathcal{F} \vdash \langle \cdot; \mathcal{A}_\emptyset; \Delta; f_0 \rangle \xrightarrow{*} \langle h; \mathcal{A}; \Delta'; f_1 \rangle$, *where* $\xrightarrow{*}$ *is the reflexive, transitive closure of the rules in Figure 6.*

DEFINITION 3.2 (Complete Preprocessing). *We write* $\Delta; \mathcal{F} \vdash f \rightsquigarrow \mathcal{A}$ *as shorthand for* $\Delta; \mathcal{F} \vdash f_0 \rightsquigarrow \langle \mathcal{A}; \cdot \rangle$.

The formal CMOD rules are shown in Figure 7. Figure 7(a) defines the judgment $\Delta; \mathcal{F} \vdash \mathcal{R}_1(f_0, f_1)$, 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 $\Delta; \mathcal{F} \vdash g \xleftarrow{\text{decl}} \mathcal{I}$, defined by [SYM-DECL]. This judgment 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 from the initial empty accumulator. Then $\Delta; \mathcal{F} \vdash \mathcal{R}_1(f_0, f_1)$ holds if for any global variable name g in N, which contains any global variable names imported by one fragment and exported by the other, it is 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_0 and f_1 include.

Figure 7(b) defines the judgment $\Delta; \mathcal{F} \vdash \mathcal{R}_2(f)$, which enforces Rule 2: each named type must have exactly one owner, either a source file or a header file. This rule examines pairs of fragments, preprocessing each one and comparing the resulting type definition maps T_1 and T_2 using [NAMED-TYPES-OK]. The first premise of [NAMED-TYPES-OK] states that, whenever some type τ in T_1 is defined within the fragment f_1 itself (i.e., its annotation is $\cdot$), τ's definition is not known to the other fragment. In other words, f_1 *owns* the type, which is abstract in f_2, ensuring information hiding. 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. The second premise of [NAMED-TYPES-OK] states that those types defined during preprocessing of both f_1 and f_2 have the same definition. Recall that T_i maps type names to types τ^h that include the file name h, so this premise ensures that both definitions came from the h, i.e., h owns the definition.

Figure 7(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 order-independent. (We will consider Rule 4 below.) [RULE 3] uses auxiliary judgment $\Delta; \mathcal{F} \vdash f \overrightarrow{\otimes} h$,

$$\text{[SYM-DECL]}\quad \frac{h \in \mathcal{I} \qquad \Delta; \mathcal{F} \vdash \mathcal{F}(h) \rightsquigarrow \mathcal{A} \qquad g \in \mathcal{A}^D}{\Delta; \mathcal{F} \vdash g \xleftarrow{\text{decl}} \mathcal{I}}$$

$$\text{[RULE 1]}\quad \frac{\begin{array}{c}\Delta; \mathcal{F} \vdash f_1 \rightsquigarrow \mathcal{A}_1 \qquad \Delta; \mathcal{F} \vdash f_2 \rightsquigarrow \mathcal{A}_2 \\ N = \left(\mathcal{A}_1^I \cap \mathcal{A}_2^E\right) \cup \left(\mathcal{A}_1^E \cap \mathcal{A}_2^I\right) \\ \forall g \in N \,.\, \Delta; \mathcal{F} \vdash g \xleftarrow{\text{decl}} \mathcal{A}_1^{\mathcal{I}} \cap \mathcal{A}_2^{\mathcal{I}}\end{array}}{\Delta; \mathcal{F} \vdash \mathcal{R}_1(f_0, f_1)}$$

(a) Rule 1: Shared Headers

$$\text{[NAMED-TYPES-OK]}\quad \frac{\begin{array}{c}\forall (t \mapsto \tau^{\cdot}) \in T_1 \,.\, t \notin \mathrm{dom}(T_2) \\ \forall t \in \mathrm{dom}(T_1) \cap \mathrm{dom}(T_2) \,.\, T_1(t) = \tau_1^{h_1} \;\wedge\; T_2(t) = \tau_2^{h_2} \Rightarrow h_1 = h_2\end{array}}{\vdash_\tau T_1, T_2}$$

$$\text{[RULE 2]}\quad \frac{\begin{array}{c}\Delta; \mathcal{F} \vdash f_0 \rightsquigarrow \mathcal{A}_0 \qquad \Delta; \mathcal{F} \vdash f_1 \rightsquigarrow \mathcal{A}_1 \qquad f_0 \neq f_1 \\ \vdash_\tau \mathcal{A}_0^T, \mathcal{A}_1^T \qquad \vdash_\tau \mathcal{A}_1^T, \mathcal{A}_0^T\end{array}}{\Delta; \mathcal{F} \vdash \mathcal{R}_2(f_0, f_1)}$$

(b) Rule 2: Consistent Named Types

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

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

(c) Rule 3: Order Independence

$$\text{[ALL]}\quad \frac{\begin{array}{c}\forall f_0, f_1 \in \mathcal{P} \,.\, \Delta; \mathcal{F} \vdash \mathcal{R}_1(f_0, f_1) \\ \forall f_0, f_1 \in \mathcal{P} \,.\, \Delta; \mathcal{F} \vdash \mathcal{R}_2(f_0, f_1) \\ \forall f \in \mathcal{P} \,.\, \Delta; \mathcal{F} \vdash \mathcal{R}_3(f)\end{array}}{\Delta; \mathcal{F} \vdash \mathcal{R}(\mathcal{P})}$$

(e) All Rules

Figure 7: CMOD Rules

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 $\mathcal{A}_1$. None of the macros that were changed in $\mathcal{A}_1$ (described by $\mathcal{A}_1^{\mathcal{C}}$) are used by h (described by $\mathcal{A}_2^{\mathcal{U}}$). Likewise, the macros changed by h ($\mathcal{A}_2^{\mathcal{C}}$) should not be used by files that came earlier ($\mathcal{A}_1^{\mathcal{U}}$). Put together, these conditions ensure that h is order-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, and so the equivalent in our formal system is to start with an accumulator and initial Δ from preprocessing `config.h`.

Finally, Figure 7(d) defines the judgment $\Delta; \mathcal{F} \vdash \mathcal{R}(\mathcal{P})$, which holds if a program $\mathcal{P}$ satisfies Rules 1, 3, and 2 in a common preprocessing environment Δ. This latter restriction enforces Rule 4. Thus if $\Delta; \mathcal{F} \vdash \mathcal{R}(\mathcal{P})$ holds, then every pair of fragments in

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

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

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

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

Figure 8: The C Compiler

$\mathcal{P}$ must use shared headers for global variables, must use order-independent header files, and must have a single owner for each type definition.

3.4 Compilation and Linking

In order to prove that the rules in Figure 7 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 [10]. 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 8, [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 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.

3.5 Soundness

We can now formally specify and prove the information hiding and link-time type safety properties of CMOD. 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 3.3 (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 show that any global variable name imported by an object file is declared in a header file.

LEMMA 3.4. *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 $f \in \mathcal{P}$ we have $\Delta; \mathcal{F} \vdash f \rightsquigarrow \mathcal{A}$ and $\Delta; \mathcal{F} \vdash f \xrightarrow{comp} [\Psi_I \Rightarrow H : \Psi_E]$. Then for any $g \in \text{dom}\,(\Psi_I)$, there exists some header file $h \in \mathcal{A}^{\mathcal{I}}$ such that $\Delta; \mathcal{F} \vdash \mathcal{F}(h) \rightsquigarrow \mathcal{A}_h$ with $g \in \mathcal{A}_h^D$.*

PROOF SKETCH Since $g \in \text{dom}\,(\Psi_I)$, by [COMPILE] it must be the case that $g \in (I - E)$, where I and E are from the accumulator in the compilation of f. Since the exports of the compiled program $\mathcal{P}$ are empty, by [LINK] there must be some $f_2 \in \mathcal{P}$ such that $\Delta; \mathcal{F} \vdash f_2 \xrightarrow{\text{comp}} [\Psi_{I2} \Rightarrow H_2 : \Psi_{E2}]$ and $g \in \text{dom}\,(\Psi_{E2})$. Then by [COMPILE], we must have $g \in E_2$, where E_2 is from the accumulator from f_2. But then since $\Delta; \mathcal{F} \vdash \mathcal{R}_1(f, f_2)$, by [RULE 1] there exists $h \in \mathcal{A}^{\mathcal{I}}$ such that $\Delta; \mathcal{F} \vdash \mathcal{F}(h) \rightsquigarrow \mathcal{A}_h$ and $g \in \mathcal{A}_h^D$.

As a corollary to this lemma, we know that any symbol not in a header file is never imported, and thus is private.

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 $h_j \in \bigcup_i \mathcal{A}_{fi}^{\mathcal{I}}$ that $\Delta; \mathcal{F} \vdash \mathcal{F}(h_j) \rightsquigarrow \mathcal{A}_{hj}$. Then for all $f_i \in \mathcal{P}$, $g \notin \bigcup_j \mathcal{A}_{hj}^D$ implies $g \notin \Psi_{Ii}$ where $\Delta; \mathcal{F} \vdash f_i \xrightarrow{comp} [\Psi_{Ii} \Rightarrow H_i : \Psi_{Ei}]$.*

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 $f_1 \in \mathcal{P}$ we have $\Delta; \mathcal{F} \vdash f_1 \rightsquigarrow \mathcal{A}_1$. Further suppose that $(t \mapsto \tau^{\cdot}) \in \mathcal{A}_1^T$. Then for any fragment $f_2 \in \mathcal{P}$ such that $f_2 \neq f_1$ and $\Delta; \mathcal{F} \vdash f_2 \rightsquigarrow \mathcal{A}_2$, we have $t \notin \text{dom}\,(\mathcal{A}_2^T)$.*

PROOF SKETCH Suppose $\Delta; \mathcal{F} \vdash f_2 \rightsquigarrow \mathcal{A}_2$ with $f_2 \neq f_1$. Then by $\Delta; \mathcal{F} \vdash \mathcal{R}_2(f, f_2)$ and [RULE 2], the first premise of [NAMED-TYPES-OK] holds, and we trivially have $t \notin \text{dom}\,(\mathcal{A}_2^T)$. (Note that we do not need the second hypothesis of [NAMED-TYPES-OK] for information hiding purposes; we use that for link-time type safety.)

Put together, Theorems 3.5 and 3.6 give us Property 2.1 for CMOD

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 [10]. Figure 9

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

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

$$\text{[MTAL0-WF-OBJ]}\ \frac{\vdash \Psi_I \qquad \vdash \Psi_A \leq \Psi_E \qquad \Psi_I \cup \Psi_A \vdash H : \Psi_A \qquad \mathrm{dom}\,(\Psi_I) \cap \mathrm{dom}\,(\Psi_A) = \emptyset}{\vdash [\Psi_I \Rightarrow H : \Psi_E]}$$

$$\text{[MTAL0-COMPAT]}\ \frac{\forall g \in \mathrm{dom}\,(\Psi_1) \cap \mathrm{dom}\,(\Psi_2)\ .\ \Psi_1(g) = \Psi_2(g)}{\vdash \Psi_1 \sim \Psi_2}$$

$$\text{[MTAL0-LC]}\ \frac{\vdash \Psi_{I1} \sim \Psi_{I2} \qquad \vdash \Psi_{I1} \sim \Psi_{E2} \qquad \vdash \Psi_{I2} \sim \Psi_{E1} \qquad \mathrm{dom}\,(\Psi_{E1}) \cap \mathrm{dom}\,(\Psi_{E2}) = \emptyset}{\vdash [\Psi_{I1} \Rightarrow H_1 : \Psi_{E1}] \overset{\text{lc}}{\leftrightarrow} [\Psi_{I2} \Rightarrow H_2 : \Psi_{E2}]}$$

$$\text{[MTAL0-LINK]}\ \frac{\begin{array}{c}\vdash [\Psi_{I1} \Rightarrow H_1 : \Psi_{E1}] \\ \vdash [\Psi_{I2} \Rightarrow H_2 : \Psi_{E2}] \\ \vdash [\Psi_{I1} \Rightarrow H_1 : \Psi_{E1}] \overset{\text{lc}}{\leftrightarrow} [\Psi_{I2} \Rightarrow H_2 : \Psi_{E2}] \\ \mathrm{dom}\,(H_1) \cap \mathrm{dom}\,(H_2) = \emptyset\end{array}}{\begin{array}{c}\vdash [\Psi_{I1} \Rightarrow H_1 : \Psi_{E1}]\ \text{link}\ [\Psi_{I2} \Rightarrow H_2 : \Psi_{E2}] \rightsquigarrow \\ [(\Psi_{I1} \cup \Psi_{I2}) \setminus (\Psi_{E1} \cup \Psi_{E2}) \Rightarrow H_1 \cup H_2 : \Psi_{E1} \cup \Psi_{E2}]\end{array}}$$

Figure 9: MTAL$_0$ [10]

gives the rules for MTAL$_0$, which we discuss briefly from bottom to top. [MTAL0-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. [MTAL0-LC] defines link-compatibility, which simply requires that the exports and imports of the object files assign the same types to shared symbols (using [MTAL0-COMPAT]) and that the same symbol is not exported twice. Finally, [MTAL0-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 [10] does allow type abstraction, 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 [19]. However, notice that Rule 2 requires that a type name has the same definition everywhere (because the C compiler requires that a type name can only be given one definition in a header file). 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, then, we need to prove that any pair of object files linked together satisfy [MTAL0-LINK]. Our strategy is to first prove that order-independence ensures that, if one file imports a symbol and one file exports a symbol, then the CMOD rules force the types to match. Then we show that as a consequence, programs that pass the CMOD rules satisfy [MTAL0-LINK]. Due to lack of space, we only sketch the proof; full details can be found in the Appendix of our companion technical report.

We first show that if order-independence holds, then any symbol that appears in a header file shared by two fragments has the same type in both fragments.

LEMMA 3.7 (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:*

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

Then if

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

all hold, then $\mathcal{A}_h^N(g) = \mathcal{A}_e^N(g) = \mathcal{A}_i^N(g)$, i.e., g maps to the same type in each fragment.

PROOF SKETCH First observe that each file includes h. By order-independence ([RULE 3]), the behavior of preprocessing h is unaffected by where it is included—in particular, its conditional compilation choices will not change, and so we can show $\mathcal{A}_h^N \subseteq \mathcal{A}_e^N$ and $\mathcal{A}_h^N \subseteq \mathcal{A}_i^N$. But then $(g \mapsto \mathcal{A}_h^N)$ appears in $\mathcal{A}_e^N$ and $\mathcal{A}_i^N$. Since [COMPILE] ensures this is the only type given to g in f_e and f_i, we know that both fragments give g the same type.

Given this lemma, it is now straightforward to show that a program will link safely if it compiles and passes the CMOD checks.

THEOREM 3.8 (Type-Safe Linking). *Suppose $\Delta; \mathcal{F} \vdash \mathcal{R}(\mathcal{P})$, and suppose $\Delta; \mathcal{F} \vdash \mathcal{P} \overset{comp}{\longrightarrow} [\emptyset \Rightarrow H_{\mathcal{P}} : \Psi_{E\mathcal{P}}]$. Also suppose that for any $f_i, f_j \in \mathcal{P}$ that are distinct ($i \neq j$), it is the case that*

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

Then

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

PROOF SKETCH By assumption [LINK] holds, and since the linked file form (O_{ij}) in [LINK] is the same as in [MTAL0-LINK], we just need to show the hypotheses of [MTAL0-LINK]. First we need to show that each object file is well-formed, which can be shown as a consequence of [COMPILE]. The last premise of [MTAL0-LINK], disjointness of the domains H_i and H_j, follows directly from [LINK].

Then to show [MTAL0-LC] holds, we first claim that $\mathrm{dom}\,(\Psi_{Ei}) \cap \mathrm{dom}\,(\Psi_{Ej}) = \emptyset$, which holds because the compiler forbids duplicate symbols in H_i and H_j, and $\mathrm{dom}\,(\Psi_{Ei}) \subseteq \mathrm{dom}\,(H_i)$ by [LET], and similarly $\mathrm{dom}\,(\Psi_{Ej}) \subseteq \mathrm{dom}\,(H_j)$.

To show $\vdash \Psi_{Ii} \sim \Psi_{Ej}$, we have $\Delta; \mathcal{F} \vdash \mathcal{R}_1(f_i, f_j)$, which by [RULE 1] implies that for any $g \in \mathrm{dom}\,(\Psi_{Ii}) \cap \mathrm{dom}\,(\Psi_{Ij})$ there is a common header than declares g. That, along with $\Delta; \mathcal{F} \vdash \mathcal{R}_3(f_i)$ and $\Delta; \mathcal{F} \vdash \mathcal{R}_3(f_j)$ by [ALL] (notice the Δs are the same, which is Rule 4), satisfy the hypotheses of Lemma 3.7, and thus g is given the same type in Ψ_{Ii} and Ψ_{Ej}. The case for $\vdash \Psi_{Ij} \sim \Psi_{Ei}$ is similar, and to show $\vdash \Psi_{Ii} \sim \Psi_{Ij}$, we observe that since the fully-compiled program has no free symbols, there must be some file that exports any symbol imported by f_i and f_j. By transitivity, those fragments both use that symbol at the same type.

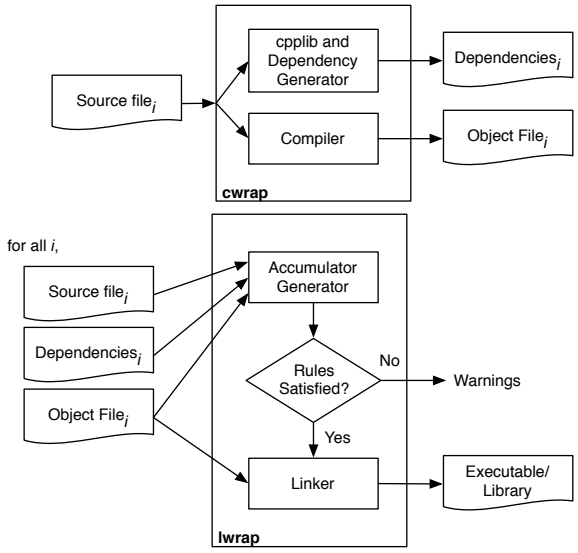

Figure 10: CMOD Architecture

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.

4. Implementation

We have implemented CMOD for the full C language.[2] Our formal language lacks several features present in the full preprocessed C, including certain conditional commands (e.g., `#if`, `#ifndef`), token concatenation `##`, and macro substitution (e.g., constants as in `#define FOO 1` and "functions" as in `#define FOO(x) (x+1)`). For that matter, CPP allows preprocessor commands to appear at arbitrary syntactic positions in the program. Put together, these additional features would be extremely difficult to model faithfully. In this section we argue why they do not affect the soundness of CMOD as applied to full C, and provide some details about our implementation.

We can think of each header as a *parameterized signature*; that is, a function from a list of macro definitions to preprocessed C program text and a modified list of macro definitions. From this point of view, we do not care *how* the function is implemented, only *which* macros it is a function of—the effects of macro substitution, syntax fragments, etc. are immaterial assuming the resulting macro environment is known (a function of `#define` and `#undef` commands) and the resulting C code is well-formed (easily checked by the compiler). Therefore, the question is, given a header file, which macro definitions could affect its processing?

In our formalism, a macro is "used" whenever it is defined or branched upon (rules [DEF], [UNDEF], [IFDEF+], and [IFDEF-]). Such uses clearly affect header processing, since one branch or the other is included in the final result. We can extend this idea to the full preprocessor's other conditional commands straightforwardly. The only other preprocessor effect due directly to macro definitions is *macro expansion*—meaning the substitution of each occurrence of a macro's name with its definition, potentially iteratively—so we must keep track of this as well.

Thus, despite the complexity of the full CPP, tracking macro uses is sufficient to know the required "inputs" for a header file, and tracking macro changes (from [DEF] and [UNDEF]) is sufficient to know its macro "outputs." Thus [RULE 4] still ensures the effects of including a header are uniform and [RULE 3] ensures they are order-independent. Moreover, [RULE 1] and [RULE 2] check for shared headers and consistent named types on *preprocessed* files (in particular, [RULE 1] preprocesses each fragment and the header file that contains the declaration). Therefore again, no matter how complex the preprocessing behavior, if the output is well-formed C (checked by the compiler) extracting the necessary declarations will be straightforward and the results consistent because the preprocessor has been applied to all fragments uniformly.

As shown in Figure 10, our implementation consists of two programs, `cwrap` and `lwrap`, which are drop-in wrappers that replace the C compiler and linker. During the build process, each source file is separately compiled with `cwrap`, which preprocesses the code using `cpplib` (part of `gcc`) and tracks the headers being preprocessed. All headers included by a source are stored as dependencies in a `.D` file along with the compile flags passed in to `cwrap`, namely preprocessor definitions (`-D`) and search path information (`-I`). Then the compiler is invoked to generate the object file.

When the program is linked using `lwrap`, we compute the accumulator information for each of the input files and check for rule violations. Unlike in the formalism, in practice it is difficult to parse C syntax and CPP directives simultaneously, and so we use a variety of tools to generate the accumulators. We add callbacks to `cpplib` to capture the include file names and the macro definitions, tests, and expansions, giving us $\mathcal{A}^{\mathcal{I}}$, $\mathcal{A}^{\mathcal{U}}$, and $\mathcal{A}^{\mathcal{C}}$, which are required for [RULE 1] and [RULE 3]. We use `ctags` [6] to extract information about types and declarations from source code, giving us $\mathcal{A}^{T}$ and $\mathcal{A}^{D}$, as required for [RULE 1] and [RULE 2]. Finally, we directly read symbol table information from the compiled ELF object files to give us $\mathcal{A}^{I}$ and $\mathcal{A}^{E}$ for checking [RULE 1]. Note that we do not compute $\mathcal{A}^{H}$, $\mathcal{A}^{N}$, or $\mathcal{A}^{Z}$, since these are needed for the soundness proof but not to actually check the rules.

If the program uses a `config.h`, the name of that file is passed to `lwrap` in an environment variable, and `lwrap` then checks that it is included consistently at the beginning of each file. This check, along with ensuring that the command line macro definitions are the same across all files, enforces Rule 4. Rules 1, 2, and 3 can then be checked using the generated accumulators, where in Rule 3 we ignore any order dependencies on `config.h`. Then after `lwrap` has checked all the CMOD rules, any violations are reported to the user. If there are no violations, then the object files are linked together as usual to build the final executable or library.

Note that because our tool has only limited reliance on direct source code information (via `ctags`), we believe that the basic CMOD approach can be easily adapted to any language that uses the same preprocessor conventions and compiles to standard object files, e.g., Cyclone and other C variants.

5. Experiments

TODO: Fix the table to swap the columns: right now rule 4 is second and rule 2 is fourth.

We evaluated CMOD on a number of publically available projects. Our goal was to measure how well the programs conform to our module system in practice, and to determine whether violations of the rules indeed are problematic.

The left portion of Figure 11 describes our benchmarks. For each benchmark, we indicate whether it has a `config.h` file, list the number of *build targets* (executables or libraries) within the project, the number of non-comment, non-blank lines of code, the number of `.c` files, and the number of `.h` files. Our tool requires programs to be compiled and linked separately, and so we do not consider targets that are compiled and linked via a single invocation

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

Program	Tgts	LoC	.c/.h	Warnings (Type Errors) Rule 1	Rule 4	Rule 3	Rule 2	Changes Required Rule 1	Rule 4	Rule 3	Rule 2	Build Time Stock	CMOD
rcs-5.7*	9	12k	25/4	0	0	0	1	0	0	0	+6/-6	3.0s	8.8s
bc-1.06*	3	10k	19/12	0	0	4	2fp	0	0	+29/-29	0	2.3s	3.4s
gawk-3.1.5*	4	30k	21/20	7	0	24	0	+1F+2/0	0	+5F+19/-5F-12	0	11.0s	15.1s
m4-1.4.4*	2	10k	19/7	2(1)	0	0	1(1)	+2/-1	0	0	1F+2/-2	3.3s	4.1s
retawq-0.2.6c*	1	21k	5/8	0	0	12	0	0	0	+1f/0	0	5.5s	6.5s
flex-2.5.4	2	16k	22/10	0	0	2	6	0	0	+1/0	+1/-15	4.7s	7.4s
xinetd-2.3.14*	8	16k	60/68	7	0	0	4	+5/-1	0	0	+1f+7/-10	6.2	12.3
gzip-1.2.4	1	5k	15/6	2	0	0	0	×	0	0	0	0.9s	1.7s
openssh-4.2p1*	13	52k	157/119	42+39fp	0	25	6	+7/-6	0	+1/-22	+57/-57	27.7s	72.7s
zebra-0.94*	8	107k	111/118	82(4)	0	37	0	+39/-6	0	0	0	32.6s	52.6s
gnuplot-4.0.0*	4	80k	49/100	7	0	353	0	+4/0	0	×	0	28.9s	41.2s
mt-daapd-0.2.4	1	18k	23/26	14(2)	0	0	1fp	+13/-2	0	0	0	6.3s	8.6s
bison-2.3*	3	21k	57/94	0	0	6	7+3fp	0	0	+2/-3	2F+6/-140	9.5s	14.1s
vsftpd-2.0.3	1	12k	34/41	4	0	9	0	+1/0	0	+3/-13	0	2.7s	4.0s

*Has `config.h` file

Figure 11: Experimental Results

of `gcc`. In the numerical totals, we count each file only once, even if it occurs in multiple targets.

We chose projects of varying sizes (from **TODO:** lowest LOC to highest LOC), varying usage and stages of development (e.g., `xinetd`, `flex`, `gawk`, and `bison` are quite mature and widely used, while `zebra`, `mtdaapd`, and `retawq` are newer and less used), and varying reuse of modules between 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 Xeon machine with 3GB RAM running the Linux 2.4.21-40.ELsmp kernel. We used `gcc` version 3.2.3, GNU `ld/ar` version 2.14.90.0.4, and `ctags` version 5.4.

5.1 Overview of Results

The right portion of figure 11 summarizes the results of applying CMOD to our benchmarks. The first group of columns count the rule violations. Note that we have not pruned duplicate warnings for the same source in different build targets. One Rule 1 warning is issued for each symbol name and pair of files such that the files import and export the symbol without a mediating heading. One Rule 2 warning is issued per definition of multiply-defined types. One Rule 3 warning is issued per pair of header files and macro such that a change and use of the macro cause an order dependence between the headers. **TODO:** Fix next sentence! The previous description of what a Rule 4 warning corresponded to didn't say anything. One per mismatched file? Just one overall? There were no Rule 4 warnings. The numbers in parentheses indicate actual type errors we found that we attribute to violating the associated rule. 7 of the 8 type errors we found were due to Rule 1 violations; the other was due to Rule 2.

Warning counts provide a gross estimate of non-conformance, so for a more precise measure we quantified the effort required to make the program CMOD-compliant. The next three columns count the number of additions (+) and deletions (-) of files (F) and lines of code (no unit) required to remove 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. Most violations required changing only a few lines of code. We did not fix two programs: `gnuplot` makes heavy use of order-dependency among include files, which would require a complete rewrite to change, and `gzip` includes assembler sources, which we cannot process. Finally, the last two columns in Figure 11 measure the time taken to build the program without and with CMOD.

We discuss the warnings in more detail below. Overall, our results demonstrate that (a) CMOD is useful at finding both type errors and "errors waiting to happen" in real software systems,

if_netlink.c:
```
25  /* Extern from rt_netlink.c */
26  void interface_lookup_netlink ();
```

rtread_netlink.c:
```
25  /* Extern from rt_netlink.c */
26  void netlink_route_read ();
```

rt_netlink.c:
```
320 int
321 netlink_interface () { ...
896 int
897 netlink_route_read () { ...
```

(a) Local, incorrect type declarations in zebra

m4.h:
```
??  typedef char boolean;
117 typedef enum { FALSE = 0, TRUE = 1 } boolean;
```

(b) Multiple, inconsistent type definitions in m4

Figure 12: Illustration of cases with suspect coding practices.

and (b) that programmers already adhere to CMOD's modularity convention for the most part, with violations easy to fix, so using CMOD is neither onerous for legacy systems nor contrary to C programming style. **TODO:** summary about performance

In the remainder of this section, we examine the warnings generated by CMOD and discuss its performance in more detail.

5.2 Warnings

We now summarize the most frequently occurring patterns that violate CMOD's rules, listed in decreasing order of severity (in terms of providing the potential for errors).

Declaration in source code rather than in headers: We found that the most common way to violate Rule 1 was to locally declare a prototype within a `.c` file, rather than include the appropriate header. Roughly **TODO:** XXX out of the R1-TOTAL (more than half) of the warnings for Rule 1 were due to this error, including all of the actual type errors. **TODO:** Is this true?? In `mt-daapd`, a client guessed incorrectly the types of `yyparse` and `yyerror`, giving them extra arguments. For `m4` the client and provider disagreed on the argument type for a function; this function was not listed in any header file. For `zebra`, clients incorrectly define prototypes for four functions, in two cases using the wrong return type and in two cases listing too few arguments; no header was defined to include

prototypes for these functions. Figure 12(a) shows the two incorrect return type declarations along with the actual definitions. Ironically, the client code even included a comment describing where the original definition was from—yet the types in the local declaration were still wrong. All type errors due to Rule 1 violations were easily fixed by moving prototypes to headers, creating headers where required.

The implementation does not include its own header: We also found that Rule 1 was sometimes violated (**TODO:** YYY out of R1-TOTAL total warnings) when a `.c` file that defined several global symbols neglected to `#include` the corresponding `.h` file that declared those symbols. As an example, in `vsftpd`, the file `oneprocess.c` fails to include its own header `oneprocess.h` (this accounts for the four Rule 1 violations in `vsftpd`).

Order dependence between a source and its header: Violations of Rule 3 were the most common (in number) across our benchmarks. One violating pattern was to parameterize a header based on a given macro, and then `#define` that macro just prior to inclusion of the code. These warnings formed close to half, **TODO:** specifically XXX of the R3-TOTAL, of the Rule 3 violations. We were able to eliminate many such violations by moving the `#define` into a common `config.h` file. **TODO:** Saurabh: Is the above sentence true? Can you quantify how much of the time this worked?

As mentioned earlier, `gnuplot` had many order dependencies that we could not easily eliminate. `gnuplut` uses files extended with `.trm` suffix to serve as both being interfaces (for prototypes) *and* sources (for actual definitions), depending on CPP directives. The project so significantly depends on this structure that it would have been an enormous task to change it.

Vertical dependencies between two headers: Dependencies between headers included adjacent to one another in a source file (so-called vertical dependencies) were the other significant source of Rule 3 violations (**TODO:** YYY out of R3-TOTAL). Vertical dependencies were typically easy to fix by converting them into horizontal dependencies. In particular, if a pair of dependent headers always occur together in the consecutive order then it is easy to move the `#include` of the first header into the second header to remove this dependency.

Duplicate type definitions in multiple locations: Programmers often (**TODO:** say how much, precisely) copy and paste type definitions to various locations in the code, rather than creating a common header. For `m4`, one instance of this practice resulted in a type error, as shown in Figure 12(b). In this case, the type `boolean` was declared with two different types that do not match. **TODO:** Saurabh: I looked in m4, and I found the numbered line, but I couldn't find the boolean typedef; m4.h disables the boolean typedef to char if the other is enabled, and vice-versa. So where is it from? Please adjust. **TODO:** Comment more on other violations. E.g., there are a LOT in xinetd. What's the source of them? Also, we claimed earlier that we have not seen spurious clashes of type names in practice; is this true? We found that in **TODO:** How many? cases, multiple definitions of the same type name in fact corresponded correctly to different types. In these cases we were able to alpha-rename the types; a notion of "static" typing would also have eliminated the warnings. **TODO:** Is the previous accurate?

Miscellaneous warnings: We found some other CMOD violations outside these categories. For `gzip`, one object file was written in assembler and so could not include its own header file; this is outside the scope of what CMOD can check but we nonetheless list it as a potential violation. Another interesting example is that in `vstftpd`, the programmer was overly concerned about multiple file inclusion. The programmer used not only the `#ifndef` pattern shown for `a.h` in Figure **??**(b), but also had `#ifndef` directives around the `#include` in the source file. This is redundant and creates a dependency between the source and the header, accounting for all of `vsftpd`'s Rule 3 violations.

Inaccuracies: In the case of `openssh`, 39 of the 81 reported warnings for Rule 1 were due to problems with `ctags`, which could not properly parse a system header. `ctags` uses heuristics to parse C code embedded with preprocessor directives. When these heuristics fail, CMOD is not made aware of some header declarations and will thus misreport Rule 1 violations.

5.3 Performance Overhead

TODO: Discuss performance results

6. Related Work

Modular programming constructs are supported by a variety of modern programming languages, such as ML [3, 17], Haskell [15], Ada [1], and Modula-3 [11]. A nice overview of module programming mechanisms appears in Pierce [22]. Most of these languages have fairly sophisticated module systems in which interfaces can be built in various ways, e.g., by providing type parameters [14, 15], or overriding abstract type declarations [17, 3]. Ramsey et al [23] propose to extend ML with support for constructing interfaces from smaller pieces, to better support software evolution. In C programming practice, `cpp` directives are the de-facto module programming language. Its computations can be sophisticated but dangerous; CMOD's rules attempt to find the sweet spot in this space.

There are a number of existing proposals for C or C++ module systems, most of which focus on organizing code into *components*. Examples include Knit [24], Koala [27], and Click [18]. In each setting, components are specified as having explicit *exports* and *imports*, and the linking process is not implicit and by name, but may be directed by the programmer. For example, one can define a Koala component by referring to two existing components, and then using the `connect` primitive to link together the exports of one to the imports of the other. This is similar to *functor application* in ML-like languages [22, 17, 3]. Explicit linking allows a component to be included potentially many times within a single program. Microsoft's Component Object Technologies (COM) model [4] is another component-based system that can be used for organizing C modules within dynamically linked libraries (DLLs). While useful, component-oriented programming is the exception rather than the norm, and these systems do nothing the enforce the programming practice of treating files as modules and header files as interfaces.

Cyclone [12], a type-safe dialect of C, originally contained a module system following earlier work for Modular Typed Assembly Language (MTAL) [10]. Like C, Cyclone files could `#include` header files, but these files were not treated as interfaces by the language. Instead, a separate interface file also had to be present for each object file, and was based on *post-processed* Cyclone code. Interface files were checked for consistency during compilation and linking. This approach was ultimately abandoned because of the disconnect between the C-style convention of using `.h` files as interface files and Cyclone's native interface files—programmers felt it was unnatural to use both. Because CMOD has low dependence on the source language and uses `cpp` as a subroutine, it could easily be applied to Cyclone programs to support modular programming.

Ernst et al [7] study preprocessor usage in a variety of programs, categorizing and explaining various idioms. They observe that C preprocessor use, particularly macro expansions, can be quite complicated. For this reason, Ernst and others suggest that preprocessor use be greatly curtailed (others suggest eliminating `cpp` altogether [16]). Some suggestions concern the use of `cpp` for modular programming. For example, an adaptation of the Indian Hill C Style and Coding Standards [2] suggests that `#ifdef` should only appear in header files, and that header files should not be nested. This is more restrictive, but similar in spirit, to our requirement that head-

ers should be order-independent. Objective-C [5] has an `#import` directive, which behaves identically to `#include` except it prevents the same file from being included twice (without using `#ifndef` convention). Spencer and Collyer [25] suggest that uses of `#ifdef` be eliminated altogether in favor of search path adjustments and scripts for finding different files. CMOD relies on the preprocessor only to include imported definitions so they can be checked for consistency by the C compiler; reduction of other `cpp` features would simplify and speed up its operation. Favre [9] describes a denotational semantics for CPP by translating it into an abstract language APP. His goal is to describe the behavior of CPP and develop ways to reason about complex CPP programs [8], whereas the goal of CMOD is to enforce a safe module system that forbids undesirable uses of the preprocessor.

A number of tools automatically check for erroneous or questionable uses of `cpp` directives, including `lint` [13], PC-lint [21], and Check [26]. However, the detected bug patterns are fairly localized and generally concern problematic macro expansions; none of these tools enforces a module system.

7. Conclusions

We have described CMOD, a module system for C that ensures type-safe separate compilation and information hiding. Our goal was to construct a sound module system that maintains compatibility with existing practice. We were able to achieve this goal by enforcing a set of four rules. At a high level, Rule 1 makes header files equivalent to regular modular interfaces; Rules 2 and 3 control preprocessor interactions; and Rule 2 checks for consistent use of type names and type abstraction. We showed formally that these rules in combination with the C compiler form a sound module system. Our experimental results show that in practice, C programs generally follow our rules, and programs that do not can be brought into compliance fairly easily. In conclusion, we believe that 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. Preprocessors from an Abstract Point of View. In *ICSM*, 1996.

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

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

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

[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] M. P. Jones. Using parameterized signatures to express modular structure. In *POPL*, 1996.

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

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

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

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

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

[20] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12), 1972.

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

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

[23] N. Ramsey, K. Fisher, and P. Govereau. An expressive language of signatures. In *ICFP*, 2005.

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

[25] H. Spencer and G. Collyer. #ifdef considered harmful, or portability experience with C News. In *USENIX Summer Technical Conference*, 1992.

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

[27] 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 7 are sound for MTAL_0, assuming no type abstraction or type naming is present.

We begin by stating some lemmas about MTAL_0 (Figure 9).

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* $\forall i, j, 1 \leq i, j \leq n, i \neq j \;.\; \vdash O_i \text{ link } O_j \rightsquigarrow O_{ij}$ *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 $\vdash O_{1\ldots n}$.

Next we describe a basic property of [COMPILE] from Figure 8.

LEMMA A.5. *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 9.

LEMMA A.6 (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 thus preprocessing f produces some result accumulator $(N, H, T, E, I, D, Z, \mathcal{C}, \mathcal{U}, \mathcal{I})$. To show that [MTAL0-WF-OBJ] holds, we need to identify a heap typing Ψ_A that satisfies the rule. We chose $\Psi_A = N|_{\neg\mathrm{dom}(\Psi_I)}$ from the result accumulator, where $\neg\mathrm{dom}\,(\Psi_I)$ is any symbol not in the domain of Ψ_I. We now can show that each of the premises of [MTAL0-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 $\Psi_A = N|_{\neg\mathrm{dom}(\Psi_I)}$. But then any symbol in $\mathrm{dom}\,(\Psi_E)$ must be in $\mathrm{dom}\,(\Psi_A)$. 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 $\mathrm{dom}\,(\Psi_I) \cup \mathrm{dom}\,(\Psi_A) = \mathrm{dom}\,(N)$ by construction, and since both are projections of N onto smaller domains, we have $\Psi_I \cup \Psi_A = N$. Thus for every $g \in \mathrm{dom}\,(N)$, we have $\Psi_I \cup \Psi_A \vdash e : N(g)$. Then since $\mathrm{dom}\,(\Psi_A) \subseteq \mathrm{dom}\,(N)$, 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. $\mathrm{dom}\,(\Psi_I) \cap \mathrm{dom}\,(\Psi_A) = \emptyset$. 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.7 (Preprocessing chains). *If* $\mathcal{F} \vdash \langle h; \mathcal{A}_\emptyset; \Delta; f_1\rangle \xrightarrow{*} \langle h_1; \mathcal{A}_1; \Delta_1; \cdot\rangle$ *then* $\mathcal{F} \vdash \langle h_1; \mathcal{A}_1; \Delta_1; f_2\rangle \xrightarrow{*} \langle h_2; \mathcal{A}_2; \Delta_2; \cdot\rangle$ *if and only if* $\mathcal{F} \vdash \langle h; \mathcal{A}_\emptyset; \Delta; (f_1; f_2)\rangle \xrightarrow{*} \langle h_2; \mathcal{A}_2; \Delta_2; \cdot\rangle$

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.8. *If* $\mathcal{F} \vdash \langle h; \mathcal{A}; \Delta; f\rangle \xrightarrow{*} \langle h'; \mathcal{A}'; \Delta'; \cdot\rangle$ *then* $h' = h$.

We use Lemmas A.7 and A.8 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.9. *Suppose that* $\mathcal{F} \vdash \langle h_0; \mathcal{A}_0; \Delta_0; f_0\rangle \xrightarrow{*} \langle h_1; \mathcal{A}_1; \Delta_1; f_1\rangle$. *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.10 (Passthrough Property). *Suppose we have*

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

where $\longrightarrow^k$ *is* k *steps of reduction by the rules in Figure 6. 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

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

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:

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

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 $\mathcal{A}'' = (\mathcal{A} \cup \mathcal{A}_{k-1})[\mathcal{U} \leftarrow^+ m] = \mathcal{A} \cup \left(\mathcal{A}_{k-1}[\mathcal{U} \leftarrow^+ m]\right) = \mathcal{A} \cup \mathcal{A}_k$. 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 $m \notin \mathcal{A}_{k-1}^{\mathcal{C}}$, by Lemma A.9 we have $\Delta_{k-1}(m) = \Delta(m)$. Then since $m \in \mathcal{A}_k^{\mathcal{U}}$, 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.9 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 $\mathcal{A}_k^{\mathcal{U}} = \mathcal{A}_{k-1}^{\mathcal{U}} \cup \{m\}$, and by induction $\Delta'_{k-1}(m) = \Delta_{k-1}(m)$ for $m \in \left(\mathcal{A}_{k-1}^{\mathcal{U}} \cup \mathcal{A}_{k-1}^{\mathcal{C}}\right)$, and $\Delta'_{k-1}(m) = \Delta'(m)$ otherwise. But we have just argued above that $\Delta'_{k-1}(m) = \Delta_{k-1}(m)$, and $m \in \mathcal{A}_k^{\mathcal{U}}$ 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 $m \in \left(\mathcal{A}_k^{\mathcal{U}} \cup \mathcal{A}_k^{\mathcal{C}}\right)$, 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.11 (Separate Reduction). *Suppose preprocessing f evaluates the statement $s = \mathsf{include}\ h$:*

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

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

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

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

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

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

Then because $\Delta; \mathcal{F} \vdash \mathcal{R}_3(f)$ and $h \in \mathcal{A}^{\mathcal{I}}$, we have $\Delta; \mathcal{F} \vdash f \overrightarrow{\otimes} h$. Therefore by [PARTIAL-INDEP], we have $\left(\mathcal{A}_1[\mathcal{I} \leftarrow^+ h]\right)^{\mathcal{C}} \cap \mathcal{A}^{\mathcal{U}}_h = \emptyset$. Then by Lemma A.9, 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

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

Notice that by (A.3), this reduction cannot get stuck. Then we can apply the Passthrough Property (Lemma A.10) on (A.5) and (A.4):

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

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 6), 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.12 (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:

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

Then by Lemma A.11, 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 $\left(g \mapsto \mathcal{A}^N_h(g)\right) \in \mathcal{A}^N_e$. 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.13 (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

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

Proof By assumption [LINK] holds. Observe that the linked file form (O_{ij}) in [LINK] is the same as in [MTAL0-OBJ], so we just need to show the hypotheses of this rule. To show [MTAL0-OBJ], we first need to show each object file is well-formed, which follows by Lemma A.6. The last premise of [MTAL0-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 [MTAL0-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 6 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

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

  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 $g \in \mathcal{A}^E_j$. Further, by [RULE 1] we have $\Delta; \mathcal{F} \vdash g \xleftarrow{\text{decl}} \mathcal{A}^{\mathcal{I}}_i \cap \mathcal{A}^{\mathcal{I}}_j$. Then by [SYM-DECL] there exists some $h \in \mathcal{A}^{\mathcal{I}}_i \cap \mathcal{A}^{\mathcal{I}}_j$ such that $\Delta; \mathcal{F} \vdash \mathcal{F}(h) \rightsquigarrow \mathcal{A}$ and $g \in \mathcal{A}^D$. Then we can apply Lemma A.12 to yield $\mathcal{A}^N_i(g) = \mathcal{A}^N_j(g)$. 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 $\Psi_{Ii}(g) = \Psi_{Ij}(g)$.

□