CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs*

George C. Necula, Scott McPeak, Shree P. Rahul, and Westley Weimer

Department of Electrical Engineering and Computer Sciences
University of California, Berkeley,
{necula,smcpeak,sprahul,weimer}@cs.berkeley.edu

Abstract. This paper describes the **C I**ntermediate **L**anguage: a high-level representation along with a set of tools that permit easy analysis and source-to-source transformation of C programs.
Compared to C, CIL has fewer constructs. It breaks down certain complicated constructs of C into simpler ones, and thus it works at a lower level than abstract-syntax trees. But CIL is also more high-level than typical intermediate languages (e.g., three-address code) designed for compilation. As a result, what we have is a representation that makes it easy to analyze and manipulate C programs, and emit them in a form that resembles the original source. Moreover, it comes with a front-end that translates to CIL not only ANSI C programs but also those using Microsoft C or GNU C extensions.
We describe the structure of CIL with a focus on how it disambiguates those features of C that we found to be most confusing for program analysis and transformation. We also describe a whole-program merger based on structural type equality, allowing a complete project to be viewed as a single compilation unit. As a representative application of CIL, we show a transformation aimed at making code immune to stack-smashing attacks. We are currently using CIL as part of a system that analyzes and instruments C programs with run-time checks to ensure type safety. CIL has served us very well in this project, and we believe it can usefully be applied in other situations as well.

1 Introduction

The C programming language is well-known for its flexibility in dealing with low-level constructs. Unfortunately, it is also well-known for being difficult to understand and analyze, both by humans and by automated tools. When we embarked on our project to analyze and instrument C programs in order to bring out the existing safe usage of pointers or to enforce it when it was not

* This research was supported in part by the National Science Foundation Career Grant No. CCR-9875171, and ITR Grants No. CCR-0085949 and No. CCR-0081588, and gifts from AT&T Research and Microsoft Research. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

R. N. Horspool (Ed.): CC 2002, LNCS 2304, pp. 213–228, 2002.

apparent, we examined a number of existing C intermediate languages and front ends before deciding to create our own. None of the available toolkits met all of our requirements. Some (e.g., [3,9]) were too high-level to support detailed analyses; some were designed to be fed to a compiler and were thus too low level, and some (e.g., SUIF [14,8]) failed to handle GCC extensions, which prevented them from working on software that use these extensions, such as Linux device drivers and kernels.

```
1 struct { int *fld; } *str1;
2 struct { int fld[5]; } str2[4];
3 str1[1].fld[2];
4 str2[1].fld[2];
```

Fig. 1. A short C program fragment highlighting ambiguous syntax

Extracting the precise meaning of a C program often requires additional processing of the abstract syntax. For example, consider lines 3 and 4 in Fig. 1. They have the same syntax but different meanings: line 3 involves three memory references while line 4 involves only one. While low-level representations do not have such ambiguities, they typically lose structural information about types, loops and other high-level constructs. In addition, it is difficult to print out such a low-level representation in a way that is faithful to the original source. Our goal has been to find a compromise between the two approaches.

The applications we are targeting are systems that want to carry out analyses and source-to-source transformations on C programs. A good intermediate language for such a task should be simple to analyze, close to the source and able to handle real-world code. This paper describes CIL, a highly-structured "clean" subset of C that meets these requirements.

CIL features a reduced number of syntactic and conceptual forms; for example, all looping constructs are reduced to a single form, all function bodies are given explicit `return` statements and syntactic sugar like "`->`" is eliminated. CIL also separates type declarations from code, makes type promotions explicit and flattens scopes (with alpha renaming) within function bodies. These simplifications reduce the number of cases that must be considered when manipulating a C program, making it more amenable to analysis and transformation. Many of these steps are carried out at some stage by most C compilers, but CIL makes analysis easier by exposing more structure in the abstract syntax.

CIL's conceptual design tries to stay close to C, so that conclusions about a CIL program can be mapped back to statements about the source program. Additionally, translating from CIL to C is fairly easy, including reconstruction of common C syntactic idioms.

Finally, a key requirement for CIL is the ability to parse and represent the variety of constructs which occur in real-world systems code, such as compiler-

specific extensions and inline assembly. CIL supports all GCC and MSVC extensions except for nested functions, and it can handle the entire Linux kernel.

The rest of this paper describes our handling of C features and CIL applications. In Section 2 we describe the syntax, typing and semantics for our language of lvalues. We present expressions and instructions in Section 3 and control-flow information in Section 4. Section 5 details our treatment of types. We discuss source-level attributes in Section 6. Having described the features of CIL we move on to using it for analysis in Section 7 and applying it to existing multi-file programs in Section 8. In Section 9 we discuss related work and we conclude in Section 10.

2 Handling of Lvalues

An *lvalue* is an expression referring to a region of storage [7]. Only an lvalue can appear on the left-hand side of an assignment. Understanding lvalues in C requires more than a simple abstract syntax tree. As shown in Fig. 1, the C fragment `str1[1].fld[2]` may involve one, two or three memory references depending on the types involved. If `str1` and `fld` are both arrays, the fragment actually refers to an offset within a single contiguous object named `str1`. If `str1` is an array and `fld` is a pointer, the value at `str[1].fld` must be loaded and then an offset from that value must be referenced. The case when `str1` is a pointer and `fld` is an array is similar. Finally, if both are pointers, `str1`, `str1[1].fld` and `str1[1].fld[2]` must all be referenced. As a result, program analyses that care about these differences will find it hard to analyze lvalues in abstract-syntax tree form.

$$
\begin{array}{lll}
\mathsf{lvalue} & ::= & \langle \mathsf{lbase}, \mathsf{loffset} \rangle \\
\mathsf{lbase} & ::= & \mathsf{Var}(\mathsf{variable}) \mid \mathsf{Mem}(\mathsf{exp}) \\
\mathsf{loffset} & ::= & \mathsf{NoOffset} \mid \mathsf{Field}(\mathsf{field}, \mathsf{loffset}) \mid \mathsf{Index}(\mathsf{exp}, \mathsf{loffset})
\end{array}
$$

Fig. 2. The abstract syntax of CIL lvalues

As shown in Fig. 2, in CIL an lvalue is expressed as a pair of a base plus an offset. The base address can be either the starting address for the storage for a variable (local or global) or any pointer expression. We distinguish the two cases so that we can tell quickly whether we are accessing a component of a variable or a memory region through a pointer. An offset in the variable or memory region denoted by the base consists of a sequence of field or index designators.

The meaning of an lvalue is a memory address along with the type of the object stored there. Fig. 3 shows the definitions of two judgments that define the meaning. The meaning of a variable base is the address of the variable and its type. The judgment $\Gamma \vdash \mathsf{lbase} \Downarrow (a, \tau)$ says that the lvalue base lbase refers to an object of type τ at address a. Lvalue offsets are treated as functions

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash \mathsf{Var}(x) \Downarrow (\&x, \tau)} \qquad \frac{\Gamma \vdash e : \mathsf{Ptr}(\tau)}{\Gamma \vdash \mathsf{Mem}(e) \Downarrow (e, \tau)}$$

$$\frac{}{\Gamma \vdash (a, \tau)@\mathsf{NoOffset} \Downarrow (a, \tau)}$$

$$\frac{\tau_1 = \mathsf{Struct}(f : \tau_f, ...) \quad \Gamma \vdash (a_1 + \mathit{OffsetOf}(f, \tau_1), \tau_f)@\mathit{off} \Downarrow (a_2, \tau_2)}{\Gamma \vdash (a_1, \tau_1)@\mathsf{Field}(f, \mathit{off}) \Downarrow (a_2, \tau_2)}$$

$$\frac{\tau_1 = \mathsf{Array}(\tau) \quad \Gamma \vdash (a_1 + e * \mathit{SizeOf}(\tau), \tau)@\mathit{off} \Downarrow (a_2, \tau_2)}{\Gamma \vdash (a_1, \tau_1)@\mathsf{Index}(e, \mathit{off}) \Downarrow (a_2, \tau_2)}$$

Fig. 3. Typing and evaluation rules for CIL lvalues

that shift address-type pairs to new address-type pairs within the same object. The judgment $\Gamma \vdash (a_1, \tau_1)@o \Downarrow (a_2, \tau_2)$ means that the lvalue offset o, when applied an lvalue denoting (a_1, τ_1), yields an lvalue denoting an object of type τ_2 at address a_2. In this latter judgment a_2 is an address within the range $[a_1, a_1 + \texttt{sizeof}(\tau_1))$.

Considering again the example from Fig. 1, the two lvalues shown there have the following CIL representations in which it is obvious when we reference a variable or a pointer indirection.

$$\texttt{str1}[1].\texttt{fld}[2] = \langle \mathsf{Mem}(2 + \mathsf{Lvalue}\langle \mathsf{Mem}(1 + \mathsf{Lvalue}\langle \mathsf{Var}(\texttt{str1}), \mathsf{NoOffset}\rangle), \mathsf{Field}(\texttt{fld}, \mathsf{NoOffset})\rangle)\rangle$$

$$\texttt{str2}[1].\texttt{fld}[2] = \langle \mathsf{Var}(\texttt{str2}), \mathsf{Index}(1, \mathsf{Index}(2, \mathsf{NoOffset}))\rangle$$

This interpretation of lvalues upholds standard C equivalences like "`x == *&x`" and "`(*(&a.f)).g == a.f.g`", and makes tasks like instrumenting every memory access in the program much easier. As in other intermediate representations, all occurrences of the same variable share a variable declaration. This makes it easy to change variable properties (like the variable name or type) and allows for the use of pointer equality checks when comparing variables.

3 Expressions and Instructions

CIL syntax has three basic concepts: expressions, instructions, and statements. Expressions represent functional computation, without side-effects or control flow. Instructions express side effects, including function calls, but have no local (intraprocedural) control flow. Statements capture local control flow.

The abstract syntax for CIL expressions is given in Fig. 4. Constants are fully typed and their original textual representation is maintained in addition to their value. SizeOf and AlignOf expressions are preserved both because computing them is dependent on compiler and compilation options, and also because a transformation may wish to change types. Casts are inserted explicitly to make the program conform to our type system, which has no implicit coercion rules.

The StartOf expression has no explicit C syntax but is used to represent the implicit coercion from an array to the address of its first element. Without such

a rule a typing judgment for `*exp` must do a case analysis based on the type of `exp`, leading to two distinct typing rules for `*exp`. The addition of StartOf allows for syntax-directed type checking, by making the coercion explicit in the source. The StartOf operator is not printed, and has the following type rule (it is the only way to convert an array to a pointer to the first element):

$$\frac{\mathsf{lvalue} \Downarrow (a, \mathsf{Array}(\tau))}{\mathsf{StartOf}(\mathsf{lvalue}) \Downarrow (a, \mathsf{Ptr}(\tau))}$$

The other C expressions (such as the "`? :`" operator or expressions that can have side-effects) are converted to CIL instructions or statements, which are discussed next.

```
exp ::= Constant(const)    | Lvalue(lvalue)            | SizeOfExp(exp)
      | SizeOfType(type)   | AlignOfExp(exp)           | AlignOfType(type)
      | UnOp(unop, exp)    | BinOp(binop, exp, exp)    | Cast(type, exp)
      | AddressOf(lvalue)  | StartOf(lvalue)

instr ::= Set(lvalue, exp)
        | Call(lvalue option, exp, exp list)
        | Asm(raw strings, lvalue list, exp list)
```

Fig. 4. The syntax of CIL expressions and instructions

Each instruction contains a single assignment or function call. The Set instruction updates the value of an lvalue. The Call instruction has an optional lvalue into which the return value of the function is stored. The function component of the Call instruction must be of function type; CIL removes redundant `&` and `*` operators applied to functions or function pointers. The arguments to functions are expressions (without side-effects or embedded control flow). Finally, the Asm instruction is used to capture the common occurrence of inline assembly in systems programs. CIL understands Microsoft- and GNU-style assembly directives and reports the inputs (as a list of expressions) and the outputs (as a list of lvalues) of the assembly block. Other information (volatility, raw assembly template strings) is stored, but not interpreted. CIL also stores location information with all statements and can take advantage of this information to insert `#line` directives when emitting output. This allows error messages in a heavily-transformed program to line up with the correct source line in the original program.

4 Integrating a CFG into the Intermediate Language

On top of the lvalues, expressions and instructions, CIL provides both high-level program structure and low-level control-flow information. The program structure is captured by a recursive structure of statements, with every statement

annotated with successor and predecessor control-flow information. This single program representation can be used with routines that require an AST (e.g., type-based analyses or pretty-printers), as well as with routines that require a CFG (e.g., dataflow analyses).

```
stmt ::= Instr(instr list)                    | Return(exp option)
       | Goto(stmt)                           | Break
       | Continue                             | If(exp, stmt list, stmt list)
       | Switch(exp, stmt list, stmt list)    | Loop(stmt list)
```

Fig. 5. The syntax of CIL statements

Fig. 5 shows the syntax of CIL statements. In addition to the information we show, each statement also contains labels, source location information and a list of successor and predecessor statements. Assignments and function calls are grouped under Instr and do not have any control flow embedded within them. CIL can resolve Break and Continue to Gotos if desired, but leaving them as they are makes code-motion transformations (e.g., loop unrolling) easier. A Return statement optionally records the return value. Every function in CIL has at least one Return statement. An If statement records the condition, which is an expression, together with the two branches, which are lists of statements. CIL has only a loop-forever looping construct and we always use a Break statement to exit from such a loop. In many cases the pretty printer is able to print out a nicer-looking `while` loop. Notice that Fig. 5 does not have any syntax for case, which is used in switch statements. The reason is we implement case as an optional label that can be associated with any statement. A switch statement then consists of an expression, a list of statements which represent the entire body of the switch (with the case labels indicating the starting point of the various cases). To provide faster access to the individual cases, we also store the starting points of the cases as a separate list in the switch statement.

5 Handling of Types

Fig. 6 describes the representation of C types in CIL. The `Named` type arises from uses of type names defined with `typedef`. The other types have their usual counterparts in C.

The notable features of CIL with respect to type handling have to do with composite types, i.e. `struct`s and `union`s. C programs can declare named and anonymous composite types at the file scope or in local scopes. This makes it hard to move expressions that involve locally defined types and also forces one to scan the entire AST to find declarations of such types. To simplify these tasks CIL moves all type declarations to the beginning of the program and gives them global scope. All anonymous composite types are given unique names in CIL and every composite types has its own declaration at the top-level. All references to a

type::=Void | Int(intKind) | Float(floatKind) | Ptr(type)
| Array(type, exp) | Fun(type, variable list) | Enum(enumInfo)
| Named(string, type) | Struct(compInfo) | Union(compInfo)

enumInfo ::= (string, item list)
compInfo ::= (string, field list)

Fig. 6. The abstract syntax of CIL types

composite type share the same instance of the `compInfo` structure, which makes it easy to change the definition of a composite type and also provides a common place to watch for recursive type definitions (all such definitions must involve at least one `compInfo`).

As far as types are concerned, CIL is similar to SUIF except that SUIF eliminates all user-defined `typedefs` and introduces extraneous ones, while CIL is careful to maintain the `typedef` structure present in the source.

6 Handling of Attributes

It is often useful to have a mechanism for the programmer to communicate additional information to the program analysis. We decided to use and extend for this purpose the GNU C notation for pragmas and attributes. Pragmas can appear only at top-level while attributes can be associated with identifiers and with their types. The advantage of this method is that `gcc` will still be able to process the annotated file (since it ignores attributes and pragmas that it does not recognize).

In GNU C a declaration can contain a number of attributes of the form `__attribute__((a))` where `a` is the attribute. For example, here is the prototype for the `printk` function found in the Linux kernel:

```
int printk(const char *fmt, ...)
               __attribute__ ((format (printf, 1, 2)));
```

The attribute above is associated with the name being declared, and it indicates that `printk` is a `printf`-like function, whose first argument is a format string, and arguments starting from the second are to be matched with the format specifiers. One difficulty in using the GNU C notation for attributes is the apparent lack of a formal specification for attribute placement and attribute association with types and identifiers. We have worked out a specification that seems to extend both that of GNU C and the placement of type qualifiers in ANSI C [6].

Attributes and pragmas can use the sub-language of C expressions excluding the comma expression and side-effecting expressions but including a constructed attribute such as the `format` attribute in the example above.

The following is the syntax of C declarations that our front-end supports:

$$
\begin{array}{lcl}
\mathsf{declaration} & ::= & \mathsf{base\_type}\ \mathsf{attributes}_{\mathsf{opt}}\ \mathsf{declarator}\ \mathsf{attributes}_{\mathsf{opt}}\ \mathsf{init}_{\mathsf{opt}} \\
\mathsf{declarator} & ::= & \mathsf{identifier} \\
& | & \mathsf{declarator}\ [\ \mathsf{exp}_{\mathsf{opt}}\ ] \\
& | & *\ \mathsf{attributes}_{\mathsf{opt}}\ \mathsf{declarator} \\
& | & \mathsf{declarator}\ (\ \mathsf{parameters}_{\mathsf{opt}}\ ) \\
& | & (\ \mathsf{attributes}_{\mathsf{opt}}\ \mathsf{declarator}\ )
\end{array}
$$

The attributes that appear at the end of the declaration are associated with the declared identifier. All other attributes are associated with types. In particular, attributes appearing after a base type are associated with that type, those appearing after the pointer type constructor `*` are associated with the pointer type. Finally, the attributes appearing before the declarator in a parenthesized declarator are associated with the type of the declarator.

For example in the declaration below we declare an array `a` of 8 pointers to functions with no arguments and returning pointers to integers:

```
int A1 * A2 (A3 * (A4 a)[8])(void) A5;
```

The attribute `A1` belongs to the type `int` and `A2` to the pointer type `int A1 *`. The attribute `A3` belongs to the function type and `A4` to the array type (the type of `a`). The attribute `A5` applies to the declared name `a`.

The `gcc` compiler accepts most of this attribute language but does not accept all of it in all contexts in which declarations occur. For example the name attributes are accepted in function prototypes but not in function definitions. This suggests that the placement of attributes has not been carefully designed in `gcc` but rather added in an ad-hoc manner.

7 Using CIL for Analyses and Source-to-Source Transformations

This section describes two concrete uses of CIL. The first is a small example that demonstrates the ease with which CIL can be used to encode simple program transformations. The second shows how CIL can be used to support serious program analysis and transformation tasks.

7.1 Preventing Buffer Overruns

To demonstrate the use of CIL for source-to-source transformations we present the CIL encoding of a refinement for the `StackGuard` [4] buffer overrun defense. `StackGuard` is a `gcc` patch that places a special "canary" word next to the return address on the stack and checks the validity of the canary word before returning from a function. It is likely that any buffer overrun that rewrites the return address will also modify the canary and thus be detected. As presented, however, the algorithm still has a slight chance of failure (e.g., if the attacker

```
exception NeedsGuarding              (* should we guard this function? *)

class containsArray = object       (* does this type contain an array? *)
  inherit nopCilVisitor                              (* only visit types *)
  method vtype t = match t with                      (* inspect the type *)
    TArray _ -> raise NeedsGuarding          (* found an array, guard it *)
    TPtr _ -> SkipChildren                     (* do not follow pointers *)
  | _ (* not array *) -> DoChildren      (* no array yet, keep looking *)
end

class sgFixupReturn restore_ra_stmt = object     (* rewrite all returns *)
  inherit nopCilVisitor                           (* only look for returns *)
  method vstmt s = match s.skind with          (* check each statement *)
    Return _ -> let new_block = mkBlock              (* restore the ra *)
      [restore_ra_stmt ; s] in ChangeTo(mkStmt new_block)
  | _ (* not Return *) -> DoChildren   (* descend in other statements *)
end

class sgAnalyzeVisitor f get_and_push_ra restore_ra = object
  inherit nopCilVisitor                          (* consider each function *)
  method vfunc fundec =              (* do we need to guard this one? *)
    try                    (* raise an exception if we need to guard it *)
      List.iter (fun vi ->               (* inspect each local variable *)
        visitCilType (new containsArray) vi.vtype ;   (* find arrays *)
      ) fundec.slocals ;
      SkipChildren                    (* no local arrays found, return *)
    with NeedsGuarding ->          (* local arrays present, guard this *)
      fundec.sbody.bstmts <- get_and_push_ra :: fundec.sbody.bstmts ;
      let modify = new sgFixupReturn restore_ra in
      fundec.sbody <- visitCilBlock modify fundec.sbody ;
      ChangeTo(fundec)                 (* now this function saves the *)
end                (* return address on entry and restores it on exit *)

let stackguard (f : file) =                (* apply the transformation *)
  let make_stmt fundec = mkStmt (Instr
    [Call(None, Lval(Var(fundec.svar),NoOffset), [], locUnknown)]) in
             (* get_and_push_ra and restore_ra are external functions *)
                (* build up CIL statements that call those functions *)
  let get_and_push_ra = make_stmt (emptyFunction "get_and_push_ra") in
  let restore_ra = make_stmt (emptyFunction "restore_ra") in
  visitCilFile (new sgAnalyzeVisitor get_and_push_ra restore_ra) f
```

Fig. 7. Complete OCaml source for a refined `StackGuard` transformation using CIL

guesses the canary value) and incurs overhead even for functions that do not have local array variables.

Fig. 7 shows a refined implementation of `StackGuard`. This transformation pushes the current return address on a private stack when a function is entered (line 28) and pops the saved value before returning (line 15). We assume that there are two external functions `get_and_push_ra` and `restore_ra` for this purpose. Only functions with local variables that contain arrays are modified (the code in lines 3–9 implements the check). This transformation is simplified by the fact that all CIL functions have explicit `return`s (checked for on line 14). The code makes use of CIL library routines (like visitors). After applying this transformation, all that remains is to provide (at link-time) the implementation for the functions that save and restore the return address. This transformation would be significantly more complicated when performed on an AST. In fact the transformation would have to perform first some of the elaboration that CIL performs.

7.2 Ensuring Memory Safety of C Programs

CCured [12] is a system that combines type inference and run-time checking to make existing C programs memory-safe. It carries out a whole-program analysis of the structure and use of the types in the program. It uses the results of the analysis to change the type definitions and memory accesses in the program. When the safety of a memory reference cannot be statically verified, an appropriate run-time check is inserted.

The analysis involves iterating over all the types in the program and comparing those that are involved in casts using a form of structural equality. CIL's simpler type language, in which recursion is limited to composite types, makes this easier. As a result of this analysis, some pointers are transformed into multi-word structures that carry extra run-time information (for example array-bounds information). Memory reads and writes involving such pointers are instrumented to contain run-time checks. These transformations are quite extensive and require detailed modifications of types, lvalues, variables and declarations. Without the clear disambiguation of these features provided by CIL, it would be difficult to determine which syntactic constructs represent accesses to memory and how to change them.

The transformed program is available for user inspection and compiler consumption. CIL's high-level structural information means that the resulting output is quite faithful to the original source, allowing the two to be compared more easily than is possible with conventional intermediate representations. CCured makes use of the whole-program merger, described in Section 8, to handle entire software projects. It also uses attributes, described in Section 6, to communicate detailed information about pointer structure.

8 A Whole-Program Merger

We have described so far an intermediate language that makes both program analysis and source-to-source transformation easy. However, many analyses are most effective when applied to the whole program. Therefore we designed and implemented a tool that merges all of a program's compilation units into a single compilation unit, with proper renaming to preserve semantics.

We designed the merger application to impersonate a compiler (it works with both the GNU C and Microsoft Visual C compilers) and to keep track of all the files that are compiled to build the whole program, along with the specific compiler options that were used for each file. When the compiler is invoked for compilation only (no linking) our tool creates the expected object file but stores in it only the preprocessed source file. The actual compilation is delayed until link time. When the compiler is invoked to link the program, it learns the names of all the object files that constitute the project. All of the associated preprocessed source files can then be loaded and merged. This setup has the benefit that it can be used with `make`-based projects by simply changing of the name of the invoked compiler and linker.

The actual merging of compilation units turned out to be surprisingly tricky. First, file-scope identifiers must be renamed properly to avoid clashes with globals and with similar identifiers in different files. In C these are the identifiers of variables and functions declared `static`, the names of types introduced with `typedef`, and the tags of union, structure and enumeration types. Unfortunately this is not sufficient because file-scope type identifiers declared in header files will result in multiple copies with different names at each inclusion point. Since C uses name equivalence for types, such copies will no longer be compatible, leading to numerous type errors in the merged program. As a result we need to do a more careful renaming of file-scope identifiers. To illustrate the problem consider the two file fragments below. For clarity, we add a "2" suffix to the file-scope names from the second file; in reality the names might be identical in the two files, especially if they originate from the same header file.

File 1:

```
struct list { int data; struct list * next; };
extern struct list *head;
struct tree { struct stuff *data; struct tree *l, *r;};
struct stuff { int elem; };
...
```

File 2:

```
struct list2 { int data; struct list2 * next; };
extern struct list2 *head;
struct tree2 { struct stuff2 *data; struct tree2 *l, *r;};
struct stuff2 { int start; int elem; };
```

Note that the tags `list` and `list2` could use the same name. In fact they must use the same name: if we give them different names then the merged program will have conflicting declarations of the global `head`. Because of the extra `start` field in `stuff2`, however, the tags `stuff` and `tree` must have names dif-

ferent from `stuff2` and `tree2` respectively. In this case if we fail to rename the `tree` tag then the program will misbehave in a very strange way. Such situations do actually occur in practice (e.g. `vortex` and `gcc` among the SPECINT95 benchmarks [13]). Such renaming errors can be very hard to find in a large program. This motivated us to try to describe precisely the problem and the merging algorithm involved in such a way that we can argue that we do not change the behavior of the program.

Our naming problem arises from the fact that C uses name equivalence for types yet different compilation units are free to use different names even for types that are intended to interoperate with other units. In essence this means that the linked program cares only about structural type equivalence. Thus when we try to merge different modules together we have to go beyond name equivalence and use structural type equivalence. A similar problem occurs in distributed systems via remote-procedure call or remote storage where different components might use different type names for types that are structurally equivalent and thus compatible [11]. This is in fact a common argument in favor of using structural type equivalence [1,2].

Our merging algorithm makes one pass over all the compilation units, incrementally accumulating a merged program. For each file there are two merging phases. In the first phase we merge the types and tags (since they do not depend on variable names). Then in the second stage we rewrite the variable declarations and function bodies. In order to merge the types we first expand all of the `typedef` definitions. This is possible because in C the body of a `typedef` cannot refer to the name being defined or to type names not already defined. This leaves us with a set of tag definitions, which can be recursive as shown above. Without loss of generality we can model the tag definitions as follows:

Tag definition	$d ::= \texttt{struct}\ t\ \{T_1; T_2\}$
Type	$T ::= \texttt{Int} \mid \texttt{Ptr}(\texttt{struct}\ t)$

Note that the constructor is always applied to a tag. The case when a pointer or array constructor would be applied to a base type is modeled as a base type and the case when the constructor would be applied to another constructed type is treated itself as a constructor application.

Given two sets of tag definitions, one from the already merged program `M` and one from the file being merged `F`, we must find which of the latter set of tags can share names with already defined tags. For the language of tag definitions considered above this is precisely structural type equivalence for recursive types. For each pair of tags $\texttt{struct}\ t\ \{T_1; T_2\}$ from `M` and $\texttt{struct}\ t'\ \{T_1'; T_2'\}$ from `F` we scan the bodies of the definitions and we find either that they always match, or that they cannot possibly match under any renaming, or that they match provided some other tags are renamed to the same name. Notice that we consider only renaming of tags with other tags. Thus exactly one of the following two kinds of constraints will be generated for each pair of tag definitions (the second kind of constraint can have zero, one or two equalities on the right of the equivalence):

$$t \neq t'$$
$$t = t' \Longleftrightarrow t_1 = t'_1 \wedge t_2 = t'_2$$

Once we decide on the names for the tags in a file we process the variable and function definitions. Among variable declarations we can share only static and `inline` function definitions. We also remove duplicate global function prototypes and `extern` variable declarations. The whole implementation of the merger algorithm is about 600 lines of OCaml code.

We have tried the merger on various programs. The largest were those from the SPECINT95 and SPECINT00 benchmark suites. We have found it to work reasonably fast, with the biggest cost being that of saving the preprocessed source files instead of the object files. For example, to merge the sources of the `gcc` compiler on a machine using an Intel Pentium 400MHz, it took 90 seconds to preprocess and save all of the sources, then 9 seconds to parse the preprocessed sources with another 9 seconds to merge them. `gcc` consists of 116 source and header files, totaling about 100,000 lines. The result of preprocessing them has two million tokens while the result of the merging is a file with only 600,000 tokens (two-thirds of all tokens are shared between modules). We have found similar results for other programs.

As side benefits from using the merger we have observed that both the `gcc` and the Microsoft C compiler parse faster and sometimes produce slightly faster executables from the merged files, supposedly due to increased ability to optimize the program. However, the increased opportunity for inlining can also make the optimization phase substantially slower when full optimization is turned on.

9 Related Work

A variety of intermediate languages have been developed for use by compilers. Most of them are too low-level to extract recognizable source after transformation. Some intermediate representations have been designed specifically to aid high-level analyses, but they do not do sufficient elaboration of the source (as CIL does for lvalues, for example) to enable detailed and trouble-free analysis or transformation.

Microsoft's AST Toolkit [3] supports all of ANSI C and C++, along with Microsoft's extensions, but it does not support GNU extensions. It is tightly integrated with the MSVC compiler, works on any program the compiler works on, and offers hooks into various compilation stages. Its high-level program representation is harder to use in source-to-source transformations. For example, it does not provide expressions without side-effects as CIL does.

Ckit [9] is a C front end written in Standard ML. It uses abstract syntax trees and does not come with built-in support for control flow graphs. Although it does full ANSI C type checking, it does not annotate the code with explicit casts and type promotions.

Edison Design Group's front end [5] features a high quality parser for the full ANSI C and C++ languages. Its emphasis is on thorough syntax analysis and error checking. It uses a high-level intermediate language, and it leaves the

task of elucidating complicated C constructs to an appropriate back end. It also works on one source file at a time.

C-Breeze [10] is an infrastructure for building C compilers. It initially parses a program into an abstract syntax tree. Although it comes with a library of routines that can construct control flow graphs and carry out various analyses, these routines work on a much lower representation of the program, which is derived from the abstract syntax tree. No built-in support is provided for analyzing programs spanning several files.

The system that meets our requirements most closely is SUIF [14,8]. SUIF is an infrastructure for compiler research, consisting of an intermediate language, front ends for C and C++ (based on Edison's front end), and a library of routines to manipulate the intermediate representation. The intermediate language has an object-oriented design and supports program representation at various levels. The library includes transformers that can ensure most of the properties that are part of CIL's design. Although SUIF handles the full ANSI C language, it does not support many of the GCC extensions that appear in programs such as the Apache web server or the Linux kernel. For example, it cannot handle GNU-style assembly instructions or attributes. As a result, we have not been able to use SUIF to process large open-source projects like the Linux kernel or the SPECINT95 `gcc` benchmark. In addition, compared to SUIF's C output, CIL's external representation is usually closer to the original source. In many cases (e.g. `typedefs`) SUIF does not retain user-supplied names, and it introduces many extraneous casts that can confuse certain kinds of analyses, such as CCured. For example, line 3 of the example on page 214 is emitted by SUIF as:

```
(((((int *) (*(((int **) (((char *) &((((struct type_1 *)
        (str1))))[1]) + 0U)))))))[2]);
```

CIL output makes the memory accesses in this statement more apparent (as described in Section 2), and at the same time its output stays close to the source:

```
*((str1 + 1)->fld + 2);
```

Finally, although SUIF comes with some support for merging multiple source files, in some cases it fails to do it correctly. For example, SUIF (version 2.2) does not correctly handle the example described in Section 8 (although some earlier versions appear to).

10 Conclusion and Future Work

The C programming language supports a number of features that make it attractive for systems programming. Unfortunately, many of these features are difficult to reason about. And even though there is abundant expertise on interpreting the constructs of the C programming language there are very few tools that make program analysis and especially source-to-source transformation easy.

CIL is a minimal design that attempts both to distill the C language constructs into a few ones with precise interpretation, and also to stay fairly close to the high-level structure of the code so that the results of source-to-source transformations bear sufficient resemblance to the source code. We have used CIL successfully both for simple analyses and transformations and also for a pervasive transformation that instruments C programs with code to ensure its memory safety. We thus believe that CIL indeed comes close to what we desire of an analysis and transformation infrastructure.

All of the CIL features came about in the context of one task or another that we used CIL for. It was surprisingly difficult in the beginning to handle lvalues and types correctly with most of the difficulties being generated by the implicit conversions in C between an array and a pointer to its first element and between a function and a pointer to it. We found that the most satisfactory solution to the first of these problems was to introduce the `StartOf` construct that does not exist in C. The only feature of CIL that we have not exercised as much as the others is the embedded control-flow graph. CCured includes a simple data flow analysis in support of array bounds checking elimination and we are starting to use CIL in yet another project where data flow analysis will be preeminent. We expect that out of these experiences we'll either gain more confidence in this part of the design or change it to better suit the needs of such analyses.

We have also found that it is extremely useful to have a whole-program merger that can act like a compiler and can be used transparently with `make`-based project. Merging errors that manage to get past the compiler and linker can be a nightmare to find in a large program, thus it is important to specify carefully how the merging algorithm works. We found that a restricted version of structural type equivalence for recursive types is both simple and sufficient for most purposes.

CIL currently handles all of ANSI C and almost all of GCC and MSVC extensions. The exception is GCC's trampoline extension for nested functions, which we have yet to encounter in practice. The next step is to extend the system to handle C++. The source code for CIL and the associated tools are available at `http://www.cs.berkeley.edu/~necula/cil`.

Acknowledgments

We wish to thank Aman Bhargava and Raymond To for help with the implementation of the CIL infrastructure, Mihai Budiu for assistance with the SUIF experiments, and the anonymous referees for their suggestions in improving this paper.

References

1. Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993. 224
2. Luca Cardelli, James Donahue, Mick Jordan, Bill Kalsow, and Greg Nelson. The Modula-3 type system. In *Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 202–212, January 1989. 224
3. Microsoft Corporation. The AST Toolkit. http://research.microsoft.com/sbt/asttoolkit/ast.asp. 214, 225
4. Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Conference*, pages 63–78, January 1998. 220
5. Edison Design Group. The C++ Front End. http://www.edg.com/cpp.html. 225
6. ISO/IEC. ISO/IEC 9899:1999(E) Programming Languages – C. 219
7. Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language (second edition)*. Prentice-Hall, Englewood Cliffs, N. J., 1988. 215
8. Holger Kienle and Urs Hölzle. Introduction to the SUIF 2.0 compiler system. Technical Report TRCS97-22, University of California, Santa Barbara. Computer Science Dept., December 10, 1997. 214, 226
9. Bell Labs. ckit: A Front End for C in SML. http://cm.bell-labs.com/cm/cs/what/smlnj/doc/ckit/overview.html. 214, 225
10. Calvin Lin, Samuel Guyer, Daniel Jimenez, and Teck Bok Tok. C-Breeze. http://www.cs.utexas.edu/users/c-breeze/. 226
11. Paul McJones and Andy Hisgen. The Topaz system: Distributed multiprocessor personal computing. In *Proceedings of the IEEE Workshop on Workstation Operating Systems*, November 1987. 224
12. George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th Annual ACM Symposium on Principles of Programming Languages*, January 2002. 222
13. Standard Performance Evaluation Corportation. SPEC 95 Benchmarks. July 1995. http://www.spec.org/osg/cpu95/CINT95. 224
14. Robert Wilson, Robert French, Christopher Wilson, Saman Amarasinghe, Jennifer Anderson, Steve Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary Hall, Monica Lam, and John Hennessy. The SUIF compiler system: a parallelizing and optimizing research compiler. Technical Report CSL-TR-94-620, Stanford University, Computer Systems Laboratory, May 1994. 214, 226