CIL: Infrastructure for C Program Analysis and Transformation

February 14, 2017

1 Introduction

CIL has a Source Forge page: http://sourceforge.net/projects/cil.

CIL (**C I**ntermediate **L**anguage) is a high-level representation along with a set of tools that permit easy analysis and source-to-source transformation of C programs.

CIL is both lower-level than abstract-syntax trees, by clarifying ambiguous constructs and removing redundant ones, and also higher-level than typical intermediate languages designed for compilation, by maintaining types and a close relationship with the source program. The main advantage of CIL is that it compiles all valid C programs into a few core constructs with a very clean semantics. Also CIL has a syntax-directed type system that makes it easy to analyze and manipulate C programs. Furthermore, the CIL front-end is able to process not only ANSI-C programs but also those using Microsoft C or GNU C extensions. If you do not use CIL and want instead to use just a C parser and analyze programs expressed as abstract-syntax trees then your analysis will have to handle a lot of ugly corners of the language (let alone the fact that parsing C itself is not a trivial task). See Section 16 for some examples of such extreme programs that CIL simplifies for you.

In essence, CIL is a highly-structured, "clean" subset of C. 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, syntactic sugar like `"->"` is eliminated and function arguments with array types become pointers. (For an extensive list of how CIL simplifies C programs, see Section 4.) This reduces the number of cases that must be considered when manipulating a C program. CIL also separates type declarations from code and flattens scopes within function bodies. This structures the program in a manner more amenable to rapid analysis and transformation. CIL computes the types of all program expressions, and makes all type promotions and casts explicit. CIL supports all GCC and MSVC extensions except for nested functions and complex numbers. Finally, CIL organizes C's imperative features into expressions, instructions and statements based on the presence and absence of side-effects and control-flow. Every statement can be annotated with successor and predecessor information. Thus CIL provides an integrated program representation that can be used with routines that require an AST (e.g. type-based analyses and pretty-printers), as well as with routines that require a CFG (e.g., dataflow analyses). CIL also supports even lower-level representations (e.g., three-address code), see Section 8.

CIL comes accompanied by a number of Perl scripts that perform generally useful operations on code:

- A driver which behaves as either the `gcc` or Microsoft VC compiler and can invoke the preprocessor followed by the CIL application. The advantage of this script is that you can easily use CIL and the analyses written for CIL with existing make files.
- A whole-program merger that you can use as a replacement for your compiler and it learns all the files you compile when you make a project and merges all of the preprocessed source files into a single one. This makes it easy to do whole-program analysis.
- A patcher makes it easy to create modified copies of the system include files. The CIL driver can then be told to use these patched copies instead of the standard ones.

CIL has been tested very extensively. It is able to process the SPECINT95 benchmarks, the Linux kernel, GIMP and other open-source projects. All of these programs are compiled to the simple CIL and then passed

to `gcc` and they still run! We consider the compilation of Linux a major feat especially since Linux contains many of the ugly GCC extensions (see Section 16.2). This adds to about 1,000,000 lines of code that we tested it on. It is also able to process the few Microsoft NT device drivers that we have had access to. CIL was tested against GCC's c-torture testsuite and (except for the tests involving complex numbers and inner functions, which CIL does not currently implement) CIL passes most of the tests. Specifically CIL fails 23 tests out of the 904 c-torture tests that it should pass. GCC itself fails 19 tests. A total of 1400 regression test cases are run automatically on each change to the CIL sources.

CIL is relatively independent on the underlying machine and compiler. When you build it CIL will configure itself according to the underlying compiler. However, CIL has only been tested on Intel x86 using the gcc compiler on Linux and cygwin and using the MS Visual C compiler. (See below for specific versions of these compilers that we have used CIL for.)

The largest application we have used CIL for is CCured, a compiler that compiles C code into type-safe code by analyzing your pointer usage and inserting runtime checks in the places that cannot be guaranteed statically to be type safe.

You can also use CIL to "compile" code that uses GCC extensions (e.g. the Linux kernel) into standard C code.

CIL also comes accompanies by a growing library of extensions (see Section 8). You can use these for your projects or as examples of using CIL.

`PDF` versions of this manual and the CIL API are available. However, we recommend the `HTML` versions because the postprocessed code examples are easier to view.

If you use CIL in your project, we would appreciate letting us know. If you want to cite CIL in your research writings, please refer to the paper "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs" by George C. Necula, Scott McPeak, S.P. Rahul and Westley Weimer, in "Proceedings of Conference on Compilier Construction", 2002.

2 Installation

You need the following tools to build CIL:

- A Unix-like shell environment (with bash, perl, make, mv, cp, etc.). On Windows, you will need cygwin with those packages.
- An ocaml compiler. You will need OCaml release 3.08 or higher to build CIL. CIL has been tested on Linux and on Windows (where it can behave as either Microsoft Visual C or gcc). On Windows, you can build CIL both with the cygwin version of ocaml (preferred) and with the Win32 version of ocaml.
- An underlying C compiler, which can be either gcc or Microsoft Visual C.

1. Get the source code.
   - *Official distribution* (Recommended):
     (a) Download the CIL distribution (latest version is `http://sourceforge.net/projects/cil/files/cil/cil-1`. See the Section **??** for recent changes to the CIL distribution.
     (b) Unzip and untar the source distribution. This will create a directory called `cil` whose structure is explained below.
     ```
     tar xvfz cil-1.7.3.tar.gz
     ```
   - *Git Repository*:
     Alternately, you can download an up to the minute version of CIL from our Subversion repository at:
     ```
     git clone git://git.code.sf.net/p/cil/code cil-code
     ```
     There is also a Github mirror:
     ```
     git clone git://github.com/kerneis/cil.git
     ```

However, the Git version may be less stable than the released version.

2. Enter the `cil` directory and run the `configure` script and then GNU make to build the distribution. If you are on Windows, at least the `configure` step must be run from within `bash`.
```
cd cil
./configure
make
make quicktest
```

3. You should now find `cilly.asm.exe` in a subdirectory of `obj`. The name of the subdirectory is either `x86_WIN32` if you are using `cygwin` on Windows or `x86_LINUX` if you are using Linux (although you should be using instead the Perl wrapper `bin/cilly`). Note that we do not have an `install` make target and you should use Cil from the development directory.

The `configure` script tries to find appropriate defaults for your system. You can control its actions by passing the following arguments:

- `CC=foo` Specifies the path for the `gcc` executable. By default whichever version is in the PATH is used. If `CC` specifies the Microsoft `cl` compiler, then that compiler will be set as the default one. Otherwise, the `gcc` compiler will be the default.

CIL requires an underlying C compiler and preprocessor. CIL depends on the underlying compiler and machine for the sizes and alignment of types. The installation procedure for CIL queries the underlying compiler for architecture and compiler dependent configuration parameters, such as the size of a pointer or the particular alignment rules for structure fields. (This means, of course, that you should re-run `./configure` when you move CIL to another machine.)

We have tested CIL on the following compilers:

- On Windows, `cl` compiler version 12.00.8168 (MSVC 6), 13.00.9466 (MSVC .Net), and 13.10.3077 (MSVC .Net 2003). Run `cl` with no arguments to get the compiler version.
- On Windows, using `cygwin` and `gcc` version 2.95.3, 3.0, 3.2, 3.3, and 3.4.
- On Linux, using `gcc` version 2.95.3, 3.0, 3.2, 3.3, 4.0, and 4.1.

Others have successfully used CIL on x86 processors with Mac OS X, FreeBSD and OpenBSD; on amd64 processors with FreeBSD; on SPARC processors with Solaris; and on PowerPC processors with Mac OS X. If you make any changes to the build system in order to run CIL on your platform, please send us a patch.

2.1 Building CIL on Windows with Microsoft Visual C

Some users might want to build a standalone CIL executable on Windows (an executable that does not require cygwin.dll to run). You will need cygwin for the build process only. Here is how we do it

1. Start with a clean CIL directory
2. Start a command-line window setup with the environment variables for Microsoft Visual Studio. You can do this by choosing Programs/Microsoft Visual Studio/Tools/Command Prompt. Check that you can run `cl`.
3. Ensure that `ocamlc` refers to a Win32 version of ocaml. Run `ocamlc -v` and look at the path to the standard library. If you have several versions of ocaml, you must set the following variables:

```
set OCAMLWIN=C:/Programs/ocaml-win

set OCAMLLIB=%OCAMLWIN%/lib
```

```
set PATH=%OCAMLWIN%/bin;%PATH%
set INCLUDE=%INCLUDE%;%OCAMLWIN%/inc
set LIB=%LIB%;%OCAMLWIN%/lib;obj/x86_WIN32
```

4. Run `bash -c "./configure CC=cl"`.
5. Run `bash -c "make WIN32=1 quickbuild"`
6. Run `bash -c "make WIN32=1 NATIVECAML=1 cilly`
7. Run `bash -c "make WIN32=1 doc`
8. Run `bash -c "make WIN32=1 bindistrib-nocheck`

The above steps do not build the CIL library, but just the executable. The last step will create a subdirectory `TEMP_cil-bindistrib` that contains everything that you need to run CIL on another machine. You will have to edit manually some of the files in the `bin` directory to replace `CILHOME`. The resulting CIL can be run with ActiveState Perl also.

3 Distribution Contents

The file `distrib/cil-1.7.3.tar.gz` contains the complete source CIL distribution, consisting of the following files:

Filename	*Description*
`Makefile.in`	`configure` source for the Makefile that builds CIL/
`configure`	The configure script.
`configure.in`	The `autoconf` source for `configure`.
`config.guess`	Stuff required by `configure`.
`config.sub`	idem
`install-sh`	idem
`doc/`	HTML documentation of the CIL API.
`obj/`	Directory that will contain the compiled CIL modules and executables.
`bin/cilly`	A Perl script that can be invoked with the same arguments as either `gcc` or Microsoft Visual C and will convert the program to CIL, perform some simple transformations, emit it and compile it as usual.
`lib/patcher`	A Perl script that applies specified patches to standard include files.
`src/check.ml,mli`	Checks the well-formedness of a CIL file.
`src/cil.ml,mli`	Definition of CIL abstract syntax and utilities for manipulating it.
`src/clist.ml,mli`	Utilities for efficiently managing lists that need to be concatenated often.
`src/errormsg.ml,mli`	Utilities for error reporting.
`src/ext/heapify.ml`	A CIL transformation that moves array local variables from the stack to the heap.
`src/ext/logcalls.ml,mli`	A CIL transformation that logs every function call.
`src/ext/sfi.ml`	A CIL transformation that can log every memory read and write.
`src/frontc/clexer.mll`	The lexer.
`src/frontc/cparser.mly`	The parser.
`src/frontc/cabs.ml`	The abstract syntax.

Filename	*Description*
`src/frontc/cprint.ml`	The pretty printer for CABS.
`src/frontc/cabs2cil.ml`	The elaborator to CIL.
`src/main.ml`	The `cilly` application.
`src/pretty.ml,mli`	Utilities for pretty printing.
`src/rmtmps.ml,mli`	A CIL tranformation that removes unused types, variables and inlined functions.
`src/stats.ml,mli`	Utilities for maintaining timing statistics.
`src/trace.ml,mli`	Utilities useful for printing debugging information.
`ocamlutil/`	Miscellaneous libraries that are not specific to CIL.
`ocamlutil/Makefile.ocaml`	A file that is included by `Makefile`.
`obj/feature_config.ml`	File generated by the Makefile describing which extra "features" to compile. See Section 5.
`obj/machdep.ml`	File generated by the Makefile containing information about your architecture, such as the size of a pointer.
`src/machdep-ml.c`	C program that generates `machdep.ml` files.

4 Compiling C to CIL

In this section we try to describe a few of the many transformations that are applied to a C program to convert it to CIL. The module that implements this conversion is about 5000 lines of OCaml code. In contrast a simple program transformation that instruments all functions to keep a shadow stack of the true return address (thus preventing stack smashing) is only 70 lines of code. This example shows that the analysis is so much simpler because it has to handle only a few simple C constructs and also because it can leverage on CIL infrastructure such as visitors and pretty-printers.

In no particular order these are a few of the most significant ways in which C programs are compiled into CIL:

1. CIL will eliminate all declarations for unused entities. This means that just because your hello world program includes `stdio.h` it does not mean that your analysis has to handle all the ugly stuff from `stdio.h`.

2. Type specifiers are interpreted and normalized:

   ```
   int long signed x;
   signed long extern x;
   long static int long y;

   // Some code that uses these declaration, so that CIL does not remove them
   int main() { return x + y; }
   ```

   See the CIL output for this code fragment

3. Anonymous structure and union declarations are given a name.

   ```
   struct { int x; } s;
   ```

   See the CIL output for this code fragment

4. Nested structure tag definitions are pulled apart. This means that all structure tag definitions can be found by a simple scan of the globals.

```
struct foo {
   struct bar {
      union baz {
          int x1;
          double x2;
      } u1;
      int y;
   } s1;
   int z;
} f;
```

See the CIL output for this code fragment

5. All structure, union, enumeration definitions and the type definitions from inners scopes are moved to global scope (with appropriate renaming). This facilitates moving around of the references to these entities.

```
int main() {
  struct foo {
        int x; } foo;
  {
     struct foo {
        double d;
     };
     return foo.x;
  }
}
```

See the CIL output for this code fragment

6. Prototypes are added for those functions that are called before being defined. Furthermore, if a prototype exists but does not specify the type of parameters that is fixed. But CIL will not be able to add prototypes for those functions that are neither declared nor defined (but are used!).

```
int f();  // Prototype without arguments
int f(double x) {
    return g(x);
}
int g(double x) {
   return x;
}
```

See the CIL output for this code fragment

7. Array lengths are computed based on the initializers or by constant folding.

```
int a1[] = {1,2,3};
int a2[sizeof(int) >= 4 ? 8 : 16];
```

See the CIL output for this code fragment

8. Enumeration tags are computed using constant folding:

```
int main() {
  enum {
     FIVE = 5,
     SIX, SEVEN,
     FOUR = FIVE - 1,
     EIGHT = sizeof(double)
  } x = FIVE;
 return x;
}
```

See the CIL output for this code fragment

9. Initializers are normalized to include specific initialization for the missing elements:

```
int a1[5] = {1,2,3};
struct foo { int x, y; } s1 = { 4 };
```

See the CIL output for this code fragment

10. Initializer designators are interpreted and eliminated. Subobjects are properly marked with braces. CIL implements the whole ISO C99 specification for initializer (neither GCC nor MSVC do) and a few GCC extensions.

```
struct foo {
   int x, y;
   int a[5];
   struct inner {
      int z;
   } inner;
} s = { 0, .inner.z = 3, .a[1 ... 2] = 5, 4, y : 8 };
```

See the CIL output for this code fragment

11. String initializers for arrays of characters are processed

```
char foo[] = "foo plus bar";
```

See the CIL output for this code fragment

12. String constants are concatenated

```
char *foo = "foo " " plus " " bar ";
```

See the CIL output for this code fragment

13. Initializers for local variables are turned into assignments. This is in order to separate completely the declarative part of a function body from the statements. This has the unfortunate effect that we have to drop the `const` qualifier from local variables !

```
int x = 5;
struct foo { int f1, f2; } a [] = {1, 2, 3, 4, 5 };
```

See the CIL output for this code fragment

14. Local variables in inner scopes are pulled to function scope (with appropriate renaming). Local scopes thus disappear. This makes it easy to find and operate on all local variables in a function.

```
int x = 5;
int main() {
  int x = 6;
  {
    int x = 7;
    return x;
  }
  return x;
}
```

See the CIL output for this code fragment

15. Global declarations in local scopes are moved to global scope:

```
int x = 5;
int main() {
  int x = 6;
  {
    static int x = 7;
    return x;
  }
  return x;
}
```

See the CIL output for this code fragment

16. Return statements are added for functions that are missing them. If the return type is not a base type then a `return` without a value is added. The guaranteed presence of return statements makes it easy to implement a transformation that inserts some code to be executed immediately before returning from a function.

```
int foo() {
  int x = 5;
}
```

See the CIL output for this code fragment

17. One of the most significant transformations is that expressions that contain side-effects are separated into statements.

```
 int x, f(int);
 return (x ++ + f(x));
```

See the CIL output for this code fragment

Internally, the `x ++` statement is turned into an assignment which the pretty-printer prints like the original. CIL has only three forms of basic statements: assignments, function calls and inline assembly.

18. Shortcut evaluation of boolean expressions and the `?:` operator are compiled into explicit conditionals:

```
int x;
int y = x ? 2 : 4;
int z = x || y;
// Here we duplicate the return statement
if(x && y) { return 0; } else { return 1; }
// To avoid excessive duplication, CIL uses goto's for
// statement that have more than 5 instructions
if(x && y || z) { x ++; y ++; z ++; x ++; y ++; return z; }
```

See the CIL output for this code fragment

19. GCC's conditional expression with missing operands are also compiled into conditionals:

```
int f();;
return f() ? : 4;
```

See the CIL output for this code fragment

20. All forms of loops (`while`, `for` and `do`) are compiled internally as a single `while(1)` looping construct with explicit `break` statement for termination. For simple `while` loops the pretty printer is able to print back the original:

```
int x, y;
for(int i = 0; i<5; i++) {
   if(i == 5) continue;
   if(i == 4) break;
   i += 2;
}
while(x < 5) {
  if(x == 3) continue;
  x ++;
}
```

See the CIL output for this code fragment

21. GCC's block expressions are compiled away. (That's right there is an infinite loop in this code.)

```
int x = 5, y = x;
int z = ({ x++; L: y -= x; y;});
return ({ goto L; 0; });
```

See the CIL output for this code fragment

22. CIL contains support for both MSVC and GCC inline assembly (both in one internal construct)

23. CIL compiles away the GCC extension that allows many kinds of constructs to be used as lvalues:

```
int x, y, z;
return &(x ? y : z) - & (x ++, x);
```

See the CIL output for this code fragment

24. All types are computed and explicit casts are inserted for all promotions and conversions that a compiler must insert:

25. CIL will turn old-style function definition (without prototype) into new-style definitions. This will make the compiler less forgiving when checking function calls, and will catch for example cases when a function is called with too few arguments. This happens in old-style code for the purpose of implementing variable argument functions.

26. Since CIL sees the source after preprocessing the code after CIL does not contain the comments and the preprocessing directives.

27. CIL will remove from the source file those type declarations, local variables and inline functions that are not used in the file. This means that your analysis does not have to see all the ugly stuff that comes from the header files:

```
#include <stdio.h>

typedef int unused_type;

static char unused_static (void) { return 0; }

int main() {
  int unused_local;
  printf("Hello world\n"); // Only printf will be kept from stdio.h
}
```

See the CIL output for this code fragment

5 How to Use CIL

There are two predominant ways to use CIL to write a program analysis or transformation. The first is to phrase your analysis as a module that is called by our existing driver. The second is to use CIL as a stand-alone library. We highly recommend that you use `cilly`, our driver.

5.1 Using `cilly`, the CIL driver

The most common way to use CIL is to write an Ocaml module containing your analysis and transformation, which you then link into our boilerplate driver application called `cilly`. `cilly` is a Perl script that processes and mimics `GCC` and `MSVC` command-line arguments and then calls `cilly.byte.exe` or `cilly.asm.exe` (CIL's Ocaml executable).

An example of such module is `logwrites.ml`, a transformation that is distributed with CIL and whose purpose is to instrument code to print the addresses of memory locations being written. (We plan to release a C-language interface to CIL so that you can write your analyses in C instead of Ocaml.) See Section 8 for a survey of other example modules.

Assuming that you have written `/home/necula/logwrites.ml`, here is how you use it:

1. Modify `logwrites.ml` so that it includes a CIL "feature descriptor" like this:

```
let feature : featureDescr =
  { fd_name = "logwrites";
    fd_enabled = ref false;
    fd_description = "generation of code to log memory writes";
    fd_extraopt = [];
    fd_doit =
    (function (f: file) ->
      let lwVisitor = new logWriteVisitor in
      visitCilFileSameGlobals lwVisitor f)
  }
```

The `fd_name` field names the feature and its associated command-line arguments. The `fd_enabled` field is a `bool ref`. "`fd_doit`" will be invoked if `!fd_enabled` is true after argument parsing, so initialize the ref cell to true if you want this feature to be enabled by default.

When the user passes the `--dologwrites` command-line option to `cilly`, the variable associated with the `fd_enabled` flag is set and the `fd_doit` function is called on the `Cil.file` that represents the merger (see Section 13) of all C files listed as arguments.

2. Invoke `configure` with the arguments

```
./configure EXTRASRCDIRS=/home/necula EXTRAFEATURES=logwrites
```

This step works if each feature is packaged into its own ML file, and the name of the entry point in the file is `feature`.

An alternative way to specify the new features is to change the build files yourself, as explained below. You'll need to use this method if a single feature is split across multiple files.

(a) Put `logwrites.ml` in the `src` or `src/ext` directory. This will make sure that `make` can find it. If you want to put it in some other directory, modify `Makefile.in` and add to `SOURCEDIRS` your directory. Alternately, you can create a symlink from `src` or `src/ext` to your file.

(b) Modify the `Makefile.in` and add your module to the `CILLY_MODULES` or `CILLY_LIBRARY_MODULES` variables. The order of the modules matters. Add your modules somewhere after `cil` and before `main`.

(c) If you have any helper files for your module, add those to the makefile in the same way. e.g.:

```
CILLY_MODULES = $(CILLY_LIBRARY_MODULES) \
                myutilities1 myutilities2 logwrites \
                main
```

Again, order is important: `myutilities2.ml` will be able to refer to Myutilities1 but not Logwrites. If you have any ocamllex or ocamlyacc files, add them to both `CILLY_MODULES` and either `MLLS` or `MLYS`.

(d) Modify `main.ml` so that your new feature descriptor appears in the global list of CIL features.

```
let features : C.featureDescr list =
  [ Logcalls.feature;
    Oneret.feature;
    Heapify.feature1;
    Heapify.feature2;
    makeCFGFeature;
    Partial.feature;
    Simplemem.feature;
    Logwrites.feature;  (* add this line to include the logwrites feature! *)
  ]
  @ Feature_config.features
```

Features are processed in the order they appear on this list. Put your feature last on the list if you plan to run any of CIL's built-in features (such as makeCFGfeature) before your own.

Standard code in `cilly` takes care of adding command-line arguments, printing the description, and calling your function automatically. Note: do not worry about introducing new bugs into CIL by adding a single line to the feature list.

3. Now you can invoke the `cilly` application on a preprocessed file, or instead use the `cilly` driver which provides a convenient compiler-like interface to `cilly`. See Section 7 for details using `cilly`. Remember to enable your analysis by passing the right argument (e.g., `--dologwrites`).

5.2 Using CIL as a library

CIL can also be built as a library that is called from your stand-alone application. Add `cil/src`, `cil/src/frontc`, `cil/obj/x86_LINUX` (or `cil/obj/x86_WIN32`) to your Ocaml project `-I` include paths. Building CIL will also build the library `cil/obj/*/cil.cma` (or `cil/obj/*/cil.cmxa`). You can then link your application against that library.

You can call the `Frontc.parse: string -> unit -> Cil.file` function with the name of a file containing the output of the C preprocessor. The `Mergecil.merge: Cil.file list -> string -> Cil.file` function merges multiple files. You can then invoke your analysis function on the resulting `Cil.file` data structure. You might want to call `Rmtmps.removeUnusedTemps` first to clean up the prototypes and variables that are not used. Then you can call the function `Cil.dumpFile: cilPrinter -> out_channel -> Cil.file -> unit` to print the file to a given output channel. A good `cilPrinter` to use is `defaultCilPrinter`.

Check out `src/main.ml` and `bin/cilly` for other good ideas about high-level file processing. Again, we highly recommend that you just our `cilly` driver so that you can avoid spending time re-inventing the wheel to provide drop-in support for standard `makefile`s.

Here is a concrete example of compiling and linking your project against CIL. Imagine that your program analysis or transformation is contained in the single file `main.ml`.

```
$ ocamlopt -c -I $(CIL)/obj/x86_LINUX/ main.ml
$ ocamlopt -ccopt -L$(CIL)/obj/x86_LINUX/ -o main unix.cmxa str.cmxa \
        $(CIL)/obj/x86_LINUX/cil.cmxa main.cmx
```

The first line compiles your analysis, the second line links it against CIL (as a library) and the Ocaml Unix library. For more information about compiling and linking Ocaml programs, see the Ocaml home page at http://caml.inria.fr/ocaml/.

In the next section we give an overview of the API that you can use to write your analysis and transformation.

6 CIL API Documentation

The CIL API is documented in the file `src/cil.mli`. We also have an online documentation extracted from `cil.mli` and other useful modules. We index below the main types that are used to represent C programs in CIL:

- An index of all types
- An index of all values
- Cil.file is the representation of a file.
- Cil.global is the representation of a global declaration or definitions. Values for operating on globals.
- Cil.typ is the representation of a type. Values for operating on types.
- Cil.compinfo is the representation of a structure or a union type
- Cil.fieldinfo is the representation of a field in a structure or a union
- Cil.enuminfo is the representation of an enumeration type.
- Cil.varinfo is the representation of a variable
- Cil.fundec is the representation of a function
- Cil.lval is the representation of an lvalue. Values for operating on lvalues.
- Cil.exp is the representation of an expression without side-effects. Values for operating on expressions.

- Cil.instr is the representation of an instruction (with side-effects but without control-flow)
- Cil.stmt is the representation of a control-flow statements. Values for operating on statements.
- Cil.attribute is the representation of attributes. Values for operating on attributes.

6.1 Using the visitor

One of the most useful tools exported by the CIL API is an implementation of the visitor pattern for CIL programs. The visiting engine scans depth-first the structure of a CIL program and at each node is queries a user-provided visitor structure whether it should do one of the following operations:

- Ignore this node and all its descendants
- Descend into all of the children and when done rebuild the node if any of the children have changed.
- Replace the subtree rooted at the node with another tree.
- Replace the subtree with another tree, then descend into the children and rebuild the node if necessary and then invoke a user-specified function.
- In addition to all of the above actions then visitor can specify that some instructions should be queued to be inserted before the current instruction or statement being visited.

By writing visitors you can customize the program traversal and transformation. One major limitation of the visiting engine is that it does not propagate information from one node to another. Each visitor must use its own private data to achieve this effect if necessary.

Each visitor is an object that is an instance of a class of type Cil.cilVisitor.. The most convenient way to obtain such classes is to specialize the Cil.nopCilVisitor.class (which just traverses the tree doing nothing). Any given specialization typically overrides only a few of the methods. Take a look for example at the visitor defined in the module `logwrites.ml`. Another, more elaborate example of a visitor is the [copyFunctionVisitor] defined in `cil.ml`.

Once you have defined a visitor you can invoke it with one of the functions:

- Cil.visitCilFile or Cil.visitCilFileSameGlobals - visit a file
- Cil.visitCilGlobal - visit a global
- Cil.visitCilFunction - visit a function definition
- Cil.visitCilExp - visit an expression
- Cil.visitCilLval - visit an lvalue
- Cil.visitCilInstr - visit an instruction
- Cil.visitCilStmt - visit a statement
- Cil.visitCilType - visit a type. Note that this does not visit the files of a composite type. use visitGlobal to visit the [GCompTag] that defines the fields.

Some transformations may want to use visitors to insert additional instructions before statements and instructions. To do so, pass a list of instructions to the Cil.queueInstr method of the specialized object. The instructions will automatically be inserted before that instruction in the transformed code. The Cil.unqueueInstr method should not normally be called by the user.

6.2 Interpreted Constructors and Deconstructors

Interpreted constructors and deconstructors are a facility for constructing and deconstructing CIL constructs using a pattern with holes that can be filled with a variety of kinds of elements. The pattern is a string that uses the C syntax to represent C language elements. For example, the following code:

```
Formatcil.cType "void * const (*)(int x)"
```

is an alternative way to construct the internal representation of the type of pointer to function with an integer argument and a void * const as result:

```
TPtr(TFun(TVoid [Attr("const", [])],
          [ ("x", TInt(IInt, []), []) ], false, []), [])
```

The advantage of the interpreted constructors is that you can use familiar C syntax to construct CIL abstract-syntax trees.

You can construct this way types, lvalues, expressions, instructions and statements. The pattern string can also contain a number of placeholders that are replaced during construction with CIL items passed as additional argument to the construction function. For example, the `%e:id` placeholder means that the argument labeled "id" (expected to be of form `Fe exp`) will supply the expression to replace the placeholder. For example, the following code constructs an increment instruction at location `loc`:

```
Formatcil.cInstr "%v:x = %v:x + %e:something"
        loc
        [ ("something", Fe some_exp);
          ("x", Fv some_varinfo) ]
```

An alternative way to construct the same CIL instruction is:

```
Set((Var some_varinfo, NoOffset),
    BinOp(PlusA, Lval (Var some_varinfo, NoOffset),
          some_exp, intType),
    loc)
```

See Cil.formatArg for a definition of the placeholders that are understood.

A dual feature is the interpreted deconstructors. This can be used to test whether a CIL construct has a certain form:

```
Formatcil.dType "void * const (*)(int x)" t
```

will test whether the actual argument `t` is indeed a function pointer of the required type. If it is then the result is `Some []` otherwise it is `None`. Furthermore, for the purpose of the interpreted deconstructors placeholders in patterns match anything of the right type. For example,

```
Formatcil.dType "void * (*)(%F:t)" t
```

will match any function pointer type, independent of the type and number of the formals. If the match succeeds the result is `Some [ FF forms ]` where `forms` is a list of names and types of the formals. Note that each member in the resulting list corresponds positionally to a placeholder in the pattern.

The interpreted constructors and deconstructors do not support the complete C syntax, but only a substantial fragment chosen to simplify the parsing. The following is the syntax that is supported:

```
Expressions:
  E ::= %e:ID | %d:ID | %g:ID | n | L | ( E ) | Unop E | E Binop E
        | sizeof E | sizeof ( T ) | alignof E  | alignof ( T )
        | & L | ( T ) E

Unary operators:
  Unop ::= + | - | ~ | %u:ID
```

```
Binary operators:
  Binop ::= + | - | * | / | << | >> | & | ``|'' | ^
          | == | != | < | > | <= | >= | %b:ID

Lvalues:
  L ::= %l:ID | %v:ID Offset | * E | (* E) Offset | E -> ident Offset

Offsets:
  Offset ::= empty | %o:ID | . ident Offset | [ E ] Offset

Types:
  T ::= Type_spec Attrs Decl

Type specifiers:
  Type_spec ::= void | char | unsigned char | short | unsigned short
             | int | unsigned int | long | unsigned long | %k:ID | float
             | double | struct %c:ID | union %c:ID


Declarators:
  Decl ::= * Attrs Decl | Direct_decl


Direct declarators:
  Direct_decl ::= empty | ident | ( Attrs Decl )
                 | Direct_decl [ Exp_opt ]
                 | ( Attrs Decl )( Parameters )

Optional expressions
  Exp_opt ::= empty | E | %eo:ID

Formal parameters
  Parameters ::= empty | ... | %va:ID | %f:ID | T | T , Parameters

List of attributes
  Attrs ::= empty | %A:ID | Attrib Attrs

Attributes
  Attrib ::= const | restrict | volatile | __attribute__ ( ( GAttr ) )

GCC Attributes
  GAttr ::= ident | ident ( AttrArg_List )

Lists of GCC Attribute arguments:
  AttrArg_List ::= AttrArg | %P:ID | AttrArg , AttrArg_List

GCC Attribute arguments
  AttrArg ::= %p:ID | ident | ident ( AttrArg_List )

Instructions
  Instr ::= %i:ID ; | L = E ; | L Binop= E | Callres L ( Args )
```

```
Actual arguments
   Args ::= empty | %E:ID | E | E , Args

Call destination
   Callres ::= empty | L = | %lo:ID

Statements
  Stmt ::= %s:ID | if ( E ) then Stmt ; | if ( E ) then Stmt else Stmt ;
       | return Exp_opt | break ; | continue ; | { Stmt_list }
       | while (E ) Stmt | Instr_list

Lists of statements
   Stmt_list ::= empty | %S:ID | Stmt Stmt_list
                | Type_spec Attrs Decl ; Stmt_list
                | Type_spec Attrs Decl = E ; Stmt_list
                | Type_spec Attrs Decl = L (Args) ; Stmt_list

List of instructions
   Instr_list ::= Instr | %I:ID | Instr Instr_list
```

Notes regarding the syntax:

- In the grammar description above non-terminals are written with uppercase initial
- All of the patterns consist of the `%` character followed by one or two letters, followed by ":" and an indentifier. For each such pattern there is a corresponding constructor of the Cil.formatArg type, whose name is the letter 'F' followed by the same one or two letters as in the pattern. That constructor is used by the user code to pass a Cil.formatArg actual argument to the interpreted constructor and by the interpreted deconstructor to return what was matched for a pattern.
- If the pattern name is uppercase, it designates a list of the elements designated by the corresponding lowercase pattern. E.g. %E designated lists of expressions (as in the actual arguments of a call).
- The two-letter patterns whose second letter is "o" designate an optional element. E.g. %eo designates an optional expression (as in the length of an array).
- Unlike in calls to `printf`, the pattern %g is used for strings.
- The usual precedence and associativity rules as in C apply
- The pattern string can contain newlines and comments, using both the `/* ... */` style as well as the `//` one.
- When matching a "cast" pattern of the form `( T ) E`, the deconstructor will match even expressions that do not have the actual cast but in that case the type is matched against the type of the expression. E.g. the patters `"(int)%e"` will match any expression of type `int` whether it has an explicit cast or not.
- The %k pattern is used to construct and deconstruct an integer type of any kind.
- Notice that the syntax of types and declaration are the same (in order to simplify the parser). This means that technically you can write a whole declaration instead of a type in the cast. In this case the name that you declare is ignored.
- In lists of formal parameters and lists of attributes, an empty list in the pattern matches any formal parameters or attributes.
- When matching types, uses of named types are unrolled to expose a real type before matching.

- The order of the attributes is ignored during matching. The the pattern for a list of attributes contains %A then the resulting `formatArg` will be bound to **all** attributes in the list. For example, the pattern `"const %A"` matches any list of attributes that contains `const` and binds the corresponding placeholder to the entire list of attributes, including `const`.
- All instruction-patterns must be terminated by semicolon
- The autoincrement and autodecrement instructions are not supported. Also not supported are complex expressions, the `&&` and `||` shortcut operators, and a number of other more complex instructions or statements. In general, the patterns support only constructs that can be represented directly in CIL.
- The pattern argument identifiers are not used during deconstruction. Instead, the result contains a sequence of values in the same order as the appearance of pattern arguments in the pattern.
- You can mix statements with declarations. For each declaration a new temporary will be constructed (using a function you provive). You can then refer to that temporary by name in the rest of the pattern.
- The `%v:` pattern specifier is optional.

The following function are defined in the `Formatcil` module for constructing and deconstructing:

- Formatcil.cExp constructs Cil.exp.
- Formatcil.cType constructs Cil.typ.
- Formatcil.cLval constructs Cil.lval.
- Formatcil.cInstr constructs Cil.instr.
- Formatcil.cStmt and Formatcil.cStmts construct Cil.stmt.
- Formatcil.dExp deconstructs Cil.exp.
- Formatcil.dType deconstructs Cil.typ.
- Formatcil.dLval deconstructs Cil.lval.
- Formatcil.dInstr deconstructs Cil.lval.

Below is an example using interpreted constructors. This example generates the CIL representation of code that scans an array backwards and initializes every even-index element with an expression:

```
Formatcil.cStmts
  loc
  "int idx = sizeof(array) / sizeof(array[0]) - 1;
   while(idx >= 0) {
     // Some statements to be run for all the elements of the array
     %S:init
     if(! (idx & 1))
       array[idx] = %e:init_even;
     /* Do not forget to decrement the index variable */
     idx = idx - 1;
   }"
  (fun n t -> makeTempVar myfunc ~name:n t)
  [ ("array", Fv myarray);
    ("init", FS [stmt1; stmt2; stmt3]);
    ("init_even", Fe init_expr_for_even_elements) ]
```

To write the same CIL statement directly in CIL would take much more effort. Note that the pattern is parsed only once and the result (a function that takes the arguments and constructs the statement) is memoized.

6.2.1 Performance considerations for interpreted constructors

Parsing the patterns is done with a LALR parser and it takes some time. To improve performance the constructors and deconstructors memoize the parsed patterns and will only compile a pattern once. Also all construction and deconstruction functions can be applied partially to the pattern string to produce a function that can be later used directly to construct or deconstruct. This function appears to be about two times slower than if the construction is done using the CIL constructors (without memoization the process would be one order of magnitude slower.) However, the convenience of interpreted constructor might make them a viable choice in many situations when performance is not paramount (e.g. prototyping).

6.3 Printing and Debugging support

The Modules Pretty and Errormsg contain respectively utilities for pretty printing and reporting errors and provide a convenient `printf`-like interface.

Additionally, CIL defines for each major type a pretty-printing function that you can use in conjunction with the Pretty interface. The following are some of the pretty-printing functions:

- Cil.d_exp - print an expression
- Cil.d_type - print a type
- Cil.d_lval - print an lvalue
- Cil.d_global - print a global
- Cil.d_stmt - print a statment
- Cil.d_instr - print an instruction
- Cil.d_init - print an initializer
- Cil.d_attr - print an attribute
- Cil.d_attrlist - print a set of attributes
- Cil.d_loc - print a location
- Cil.d_ikind - print an integer kind
- Cil.d_fkind - print a floating point kind
- Cil.d_const - print a constant
- Cil.d_storage - print a storage specifier

You can even customize the pretty-printer by creating instances of Cil.cilPrinter.. Typically such an instance extends Cil.defaultCilPrinter. Once you have a customized pretty-printer you can use the following printing functions:

- Cil.printExp - print an expression
- Cil.printType - print a type
- Cil.printLval - print an lvalue
- Cil.printGlobal - print a global
- Cil.printStmt - print a statment
- Cil.printInstr - print an instruction
- Cil.printInit - print an initializer

- Cil.printAttr - print an attribute
- Cil.printAttrs - print a set of attributes

CIL has certain internal consistency invariants. For example, all references to a global variable must point to the same `varinfo` structure. This ensures that one can rename the variable by changing the name in the `varinfo`. These constraints are mentioned in the API documentation. There is also a consistency checker in file `src/check.ml`. If you suspect that your transformation is breaking these constraints then you can pass the `--check` option to cilly and this will ensure that the consistency checker is run after each transformation.

6.4 Attributes

In CIL you can attach attributes to types and to names (variables, functions and fields). Attributes are represented using the type Cil.attribute. An attribute consists of a name and a number of arguments (represented using the type Cil.attrparam). Almost any expression can be used as an attribute argument. Attributes are stored in lists sorted by the name of the attribute. To maintain list ordering, use the functions Cil.typeAttrs to retrieve the attributes of a type and the functions Cil.addAttribute and Cil.addAttributes to add attributes. Alternatively you can use Cil.typeAddAttributes to add an attribute to a type (and return the new type).

GCC already has extensive support for attributes, and CIL extends this support to user-defined attributes. A GCC attribute has the syntax:

```
gccattribute ::= __attribute__((attribute))    (Note the double parentheses)
```

Since GCC and MSVC both support various flavors of each attribute (with or without leading or trailing _) we first strip ALL leading and trailing _ from the attribute name (but not the identified in [ACons] parameters in Cil.attrparam). When we print attributes, for GCC we add two leading and two trailing _; for MSVC we add just two leading _.

There is support in CIL so that you can control the printing of attributes (see Cil.setCustomPrintAttribute and Cil.setCustomPrintAttributeScope). This custom-printing support is now used to print the "const" qualifier as "`const`" and not as "`__attribute__((const))`".

The attributes are specified in declarations. This is unfortunate since the C syntax for declarations is already quite complicated and after writing the parser and elaborator for declarations I am convinced that few C programmers understand it completely. Anyway, this seems to be the easiest way to support attributes.

Name attributes must be specified at the very end of the declaration, just before the `=` for the initializer or before the `,` that separates a declaration in a group of declarations or just before the `;` that terminates the declaration. A name attribute for a function being defined can be specified just before the brace that starts the function body.

For example (in the following examples `A1`,...,`An` are type attributes and `N` is a name attribute (each of these uses the `__attribute__` syntax):

```
int x N;
int x N, * y N = 0, z[] N;
extern void exit() N;
int fact(int x) N { ... }
```

Type attributes can be specified along with the type using the following rules:

1. The type attributes for a base type (int, float, named type, reference to struct or union or enum) must be specified immediately following the type (actually it is Ok to mix attributes with the specification of the type, in between unsigned and int for example).

   For example:

   ```
   int A1 x N;  /* A1 applies to the type int. An example is an attribute
                   "even" restricting the type int to even values. */
   struct foo A1 A2 x; // Both A1 and A2 apply to the struct foo type
   ```

2. The type attributes for a pointer type must be specified immediately after the * symbol.

```
/* A pointer (A1) to an int (A2) */
int A2 * A1 x;
/* A pointer (A1) to a pointer (A2) to a float (A3) */
float A3 * A2 * A1 x;
```

   Note: The attributes for base types and for pointer types are a strict extension of the ANSI C type qualifiers (const, volatile and restrict). In fact CIL treats these qualifiers as attributes.

3. The attributes for a function type or for an array type can be specified using parenthesized declarators. For example:

```
/* A function (A1) from int (A2) to float (A3) */
float A3 (A1 f)(int A2);

/* A pointer (A1) to a function (A2) that returns an int (A3) */
int A3 (A2 * A1 pfun)(void);

/* An array (A1) of int (A2) */
int A2 (A1 x0)[]

/* Array (A1) of pointers (A2) to functions (A3) that take an int (A4) and
 * return a pointer (A5) to int (A6)  */
int A6 * A5 (A3 * A2 (A1 x1)[5])(int A4);


/* A function (A4) that takes a float (A5) and returns a pointer (A6) to an
 * int (A7) */
extern int A7 * A6 (A4 x2)(float A5 x);

/* A function (A1) that takes a int (A2) and that returns a pointer (A3) to
 * a function (A4) that takes a float (A5) and returns a pointer (A6) to an
 * int (A7) */
int A7 * A6 (A4 * A3 (A1 x3)(int A2 x))(float A5) {
   return & x2;
}
```

Note: ANSI C does not allow the specification of type qualifiers for function and array types, although it allows for the parenthesized declarator. With just a bit of thought (looking at the first few examples above) I hope that the placement of attributes for function and array types will seem intuitive.

This extension is not without problems however. If you want to refer just to a type (in a cast for example) then you leave the name out. But this leads to strange conflicts due to the parentheses that we introduce to scope the attributes. Take for example the type of x0 from above. It should be written as:

```
int A2 (A1 )[]
```

But this will lead most C parsers into deep confusion because the parentheses around A1 will be confused for parentheses of a function designator. To push this problem around (I don't know a solution) whenever we are about to print a parenthesized declarator with no name but with attributes, we comment out the attributes so you can see them (for whatever is worth) without confusing the compiler. For example, here is how we would print the above type:

```
int A2 /*(A1 )*/[]
```

Handling of predefined GCC attributes GCC already supports attributes in a lot of places in declarations. The only place where we support attributes and GCC does not is right before the { that starts a function body.

GCC classifies its attributes in attributes for functions, for variables and for types, although the latter category is only usable in definition of struct or union types and is not nearly as powerful as the CIL type attributes. We have made an effort to reclassify GCC attributes as name and type attributes (they only apply for function types). Here is what we came up with:

- GCC name attributes:

  section, constructor, destructor, unused, weak, no_instrument_function, noreturn, alias, no_check_memory_usage, dllinport, dllexport, exception, model

  Note: the "noreturn" attribute would be more appropriately qualified as a function type attribute. But we classify it as a name attribute to make it easier to support a similarly named MSVC attribute.

- GCC function type attributes:

  fconst (printed as "const"), format, regparm, stdcall, cdecl, longcall

  I was not able to completely decipher the position in which these attributes must go. So, the CIL elaborator knows these names and applies the following rules:

  - All of the name attributes that appear in the specifier part (i.e. at the beginning) of a declaration are associated with all declared names.
  - All of the name attributes that appear at the end of a declarator are associated with the particular name being declared.
  - More complicated is the handling of the function type attributes, since there can be more than one function in a single declaration (a function returning a pointer to a function). Lacking any real understanding of how GCC handles this, I attach the function type attribute to the "nearest" function. This means that if a pointer to a function is "nearby" the attribute will be correctly associated with the function. In truth I pray that nobody uses declarations as that of x3 above.

Handling of predefined MSVC attributes MSVC has two kinds of attributes, declaration modifiers to be printed before the storage specifier using the notation "`__declspec(...)`" and a few function type attributes, printed almost as our CIL function type attributes.

The following are the name attributes that are printed using `__declspec` right before the storage designator of the declaration: thread, naked, dllimport, dllexport, noreturn

The following are the function type attributes supported by MSVC: fastcall, cdecl, stdcall

It is not worth going into the obscure details of where MSVC accepts these type attributes. The parser thinks it knows these details and it pulls these attributes from wherever they might be placed. The important thing is that MSVC will accept if we print them according to the rules of the CIL attributes !

7 The CIL Driver

We have packaged CIL as an application `cilly` that contains certain example modules, such as `logwrites.ml` (a module that instruments code to print the addresses of memory locations being written). Normally, you write another module like that, add command-line options and an invocation of your module in `src/main.ml`. Once you compile CIL you will obtain the file `obj/cilly.asm.exe`.

We wrote a driver for this executable that makes it easy to invoke your analysis on existing C code with very little manual intervention. This driver is `bin/cilly` and is quite powerful. Note that the `cilly` script is configured during installation with the path where CIL resides. This means that you can move it to any place you want.

A simple use of the driver is:

```
bin/cilly --save-temps -D HAPPY_MOOD -I myincludes hello.c -o hello
```

--save-temps tells CIL to save the resulting output files in the current directory. Otherwise, they'll be put in `/tmp` and deleted automatically. Not that this is the only CIL-specific flag in the list – the other flags use `gcc`'s syntax.

This performs the following actions:

- preprocessing using the -D and -I arguments with the resulting file left in `hello.i`,
- the invocation of the `cilly.asm` application which parses `hello.i` converts it to CIL and the pretty-prints it to `hello.cil.c`
- another round of preprocessing with the result placed in `hello.cil.i`
- the true compilation with the result in `hello.cil.o`
- a linking phase with the result in `hello`

Note that `cilly` behaves like the `gcc` compiler. This makes it easy to use it with existing `Makefiles`:

```
make CC="bin/cilly" LD="bin/cilly"
```

`cilly` can also behave as the Microsoft Visual C compiler, if the first argument is `--mode=MSVC`:

```
bin/cilly --mode=MSVC /D HAPPY_MOOD /I myincludes hello.c /Fe hello.exe
```

(This in turn will pass a `--MSVC` flag to the underlying `cilly.asm` process which will make it understand the Microsoft Visual C extensions)

`cilly` can also behave as the archiver `ar`, if it is passed an argument `--mode=AR`. Note that only the `cr` mode is supported (create a new archive and replace all files in there). Therefore the previous version of the archive is lost. You will also need to remove any other commands that operate on the generated library (e.g. `ranlib`, `lorder`), as the .a file is no longer an actual binary library.

Furthermore, `cilly` allows you to pass some arguments on to the underlying `cilly.asm` process. As a general rule all arguments that start with `--` and that `cilly` itself does not process, are passed on. For example,

```
bin/cilly --dologwrites -D HAPPY_MOOD -I myincludes hello.c -o hello.exe
```

will produce a file `hello.cil.c` that prints all the memory addresses written by the application.

The most powerful feature of `cilly` is that it can collect all the sources in your project, merge them into one file and then apply CIL. This makes it a breeze to do whole-program analysis and transformation. All you have to do is to pass the `--merge` flag to `cilly`:

```
make CC="bin/cilly --save-temps --dologwrites --merge"
```

You can even leave some files untouched:

```
make CC="bin/cilly --save-temps --dologwrites --merge --leavealone=foo --leavealone=bar"
```

This will merge all the files except those with the basename `foo` and `bar`. Those files will be compiled as usual and then linked in at the very end.

The sequence of actions performed by `cilly` depends on whether merging is turned on or not:

- If merging is off
  1. For every file `file.c` to compile
     (a) Preprocess the file with the given arguments to produce `file.i`
     (b) Invoke `cilly.asm` to produce a `file.cil.c`
     (c) Preprocess to `file.cil.i`
     (d) Invoke the underlying compiler to produce `file.cil.o`
  2. Link the resulting objects

- If merging is on
  1. For every file `file.c` to compile
     (a) Preprocess the file with the given arguments to produce `file.i`
     (b) Save the preprocessed source as `file.o`
  2. When linking executable `hello.exe`, look at every object file that must be linked and see if it actually contains preprocessed source. Pass all those files to a special merging application (described in Section 13) to produce `hello.exe_comb.c`
  3. Invoke `cilly.asm` to produce a `hello.exe_comb.cil.c`
  4. Preprocess to `hello.exe_comb.cil.i`
  5. Invoke the underlying compiler to produce `hello.exe_comb.cil.o`
  6. Invoke the actual linker to produce `hello.exe`

Note that files that you specify with `--leavealone` are not merged and never presented to CIL. They are compiled as usual and then are linked in at the end.

And a final feature of `cilly` is that it can substitute copies of the system's include files:

```
make CC="bin/cilly --includedir=myinclude"
```

This will force the preprocessor to use the file `myinclude/xxx/stdio.h` (if it exists) whenever it encounters `#include <stdio.h>`. The `xxx` is a string that identifies the compiler version you are using. This modified include files should be produced with the patcher script (see Section 14).

7.1 `cilly` Options

Among the options for the `cilly` you can put anything that can normally go in the command line of the compiler that `cilly` is impersonating. `cilly` will do its best to pass those options along to the appropriate subprocess. In addition, the following options are supported (a complete and up-to-date list can always be obtained by running `cilly --help`):

- `--gcc=command` Tell `cilly` to use `command` to invoke gcc, e.g. `--gcc=arm-elf-gcc` to use a cross-compiler. See also the `--envmachine` option below that tells CIL to assume a different machine model.
- `--mode=mode` This must be the first argument if present. It makes `cilly` behave as a given compiled. The following modes are recognized:
  - GNUCC - the GNU C Compiler. This is the default.
  - MSVC - the Microsoft Visual C compiler. Of course, you should pass only MSVC valid options in this case.
  - AR - the archiver `ar`. Only the mode `cr` is supported and the original version of the archive is lost.
- `--help` Prints a list of the options supported.
- `--verbose` Prints lots of messages about what is going on.
- `--stages` Less than `--verbose` but lets you see what `cilly` is doing.
- `--merge` This tells `cilly` to first attempt to collect into one source file all of the sources that make your application, and then to apply `cilly.asm` on the resulting source. The sequence of actions in this case is described above and the merger itself is described in Section 13.
- `--leavealone=xxx`. Do not merge and do not present to CIL the files whose basename is "xxx". These files are compiled as usual and linked in at the end.

- `--includedir=xxx`. Override the include files with those in the given directory. The given directory is the same name that was given an an argument to the patcher (see Section 14). In particular this means that that directory contains subdirectories named based on the current compiler version. The patcher creates those directories.
- `--usecabs`. Do not CIL, but instead just parse the source and print its AST out. This should looked like the preprocessed file. This is useful when you suspect that the conversion to CIL phase changes the meaning of the program.
- `--save-temps=xxx`. Temporary files are preserved in the xxx directory. For example, the output of CIL will be put in a file named `*.cil.c`.
- `--save-temps`. Temporay files are preserved in the current directory.

7.2 `cilly.asm` Options

All of the options that start with `--` and are not understood by `cilly` are passed on to `cilly.asm`. `cilly` also passes along to `cilly.asm` flags such as `--MSVC` that both need to know about. The following options are supported. Many of these flags also have corresponding "`--no*`" versions if you need to go back to the default, as in "`--nowarnall`".

General Options:

- `--version` output version information and exit
- `--verbose` Print lots of random stuff. This is passed on from cilly
- `--warnall` Show all warnings.
- `--debug=xxx` turns on debugging flag xxx
- `--nodebug=xxx` turns off debugging flag xxx
- `--flush` Flush the output streams often (aids debugging).
- `--check` Run a consistency check over the CIL after every operation.
- `--strictcheck` Same as `--check`, but it treats consistency problems as errors instead of warnings.
- `--nocheck` turns off consistency checking of CIL.
- `--noPrintLn` Don't output #line directives in the output.
- `--commPrintLn` Print #line directives in the output, but put them in comments.
- `--commPrintLnSparse` Like `--commPrintLn` but print only new line numbers.
- `--log=xxx` Set the name of the log file. By default stderr is used
- `--MSVC` Enable MSVC compatibility. Default is GNU.
- `--ignore-merge-conflicts` ignore merging conflicts.
- `--extrafiles=filename`: the name of a file that contains a list of additional files to process, separated by whitespace.
- `--stats` Print statistics about the running time of the parser, conversion to CIL, etc. Also prints memory-usage statistics. You can time parts of your own code as well. Calling (`Stats.time ``label'' func arg`) will evaluate `(func arg)` and remember how long this takes. If you call `Stats.time` repeatedly with the same label, CIL will report the aggregate time.

  If available, CIL uses the x86 performance counters for these stats. This is very precise, but results in "wall-clock time." To report only user-mode time, find the call to `Stats.reset` in `main.ml`, and change it to `Stats.reset Stats.SoftwareTimer`.

- `--envmachine`. Use machine model specified in `CIL_MACHINE` environment variable, rather than the one compiled into CIL. Note that you should not pass gcc's 32/64-bit `-m32` and `-m64` options to `cilly` if you use `--envmachine` (they use `--envmachine` under the hood). See Section 7.4 for a description of `CIL_MACHINE`'s format.

  Lowering Options

- `--noLowerConstants` do not lower constant expressions.
- `--noInsertImplicitCasts` do not insert implicit casts.
- `--forceRLArgEval` Forces right to left evaluation of function arguments.
- `--disallowDuplication` Prevent small chunks of code from being duplicated.
- `--keepunused` Do not remove the unused variables and types.
- `--rmUnusedInlines` Delete any unused inline functions. This is the default in MSVC mode.

  Output Options:

- `--printCilAsIs` Do not try to simplify the CIL when printing. Without this flag, CIL will attempt to produce prettier output by e.g. changing `while(1)` into more meaningful loops.
- `--noWrap` do not wrap long lines when printing
- `--out=xxx` the name of the output CIL file. `cilly` sets this for you.
- `--mergedout=xxx` specify the name of the merged file
- `--cabsonly=xxx` CABS output file name

  Selected features. See Section 8 for more information.

- `--dologcalls`. Insert code in the processed source to print the name of functions as are called. Implemented in `src/ext/logcalls.ml`.
- `--dologwrites`. Insert code in the processed source to print the address of all memory writes. Implemented in `src/ext/logwrites.ml`.
- `--dooneRet`. Make each function have at most one 'return'. Implemented in `src/ext/oneret.ml`.
- `--dostackGuard`. Instrument function calls and returns to maintain a separate stack for return addresses. Implemeted in `src/ext/heapify.ml`.
- `--domakeCFG`. Make the program look more like a CFG. Implemented in `src/cil.ml`.
- `--dopartial`. Do interprocedural partial evaluation and constant folding. Implemented in `src/ext/partial.ml`.
- `--dosimpleMem`. Simplify all memory expressions. Implemented in `src/ext/simplemem.ml`.

  For an up-to-date list of available options, run `cilly.asm --help`.

7.3 Internal Options

All of the `cilly.asm` options described above can be set programmatically – see `src/ciloptions.ml` or the individual extensions to see how. Some options should be set before parsing to be effective.

Additionally, a few CIL options have no command-line flag and can only be set programmatically. These options may be useful for certain analyses:

- `Cabs2cil.doCollapseCallCast`:This is false by default. Set to true to replicate the behavior of CIL 1.3.5 and earlier.

  When false, all casts in the program are made explicit using the `CastE` expression. Accordingly, the destination of a Call instruction will always have the same type as the function's return type.

  If true, the destination type of a Call may differ from the return type, so there is an implicit cast. This is useful for analyses involving `malloc`. Without this option, CIL converts "`T* x = malloc(n);`" into "`void* tmp = malloc(n); T* x = (T*)tmp;`". If you don't need all casts to be made explicit, you can set `Cabs2cil.doCollapseCallCast` to true so that CIL won't insert a temporary and you can more easily determine the allocation type from calls to `malloc`.

7.4 Specifying a machine model

The `--envmachine` option tells CIL to get its machine model from the `CIL_MACHINE` environment variable, rather than use the model compiled in to CIL itself. This is necessary when using CIL as part of a cross-compilation setup, and to handle gcc's `-m32` and `-m64` which select between a 32-bit and 64-bit machine model.

`CIL_MACHINE` is a space-separated list of key=value pairs. Unknown keys are ignored. The following keys must be defined:

Key	Value
bool	sizeof(_Bool),alignof(_Bool)
short	sizeof(short),alignof(short)
int	sizeof(int),alignof(int)
long	sizeof(long),alignof(long)
long_long	sizeof(long long),alignof(long long)
float	sizeof(float),alignof(float)
double	sizeof(double),alignof(double)
long_double	sizeof(long double),alignof(long double)
pointer	sizeof(all pointers),alignof(all pointers)
enum	sizeof(all enums),alignof(all enums)
fun	sizeof(all fn ptrs),alignof(all fn ptrs)
alignof_string	alignof(all string constants)
max_alignment	maximum alignment for any type
size_t	the definition of size_t
wchar_t	the definition of wchar_t
char_signed	true if char is signed
big_endian	true for big-endian machine models
const_string_literals	true if string constants are const
__thread_is_keyword	true if `__thread` is a keyword
__builtin_va_list	true if `__builtin_va_list` is a builtin type
underscore_name	true if generated symbols preceded by _

Some notes:

- Values cannot contain spaces as spaces separate key/value pairs.
- The value of size_t is the text for the type defined as size_t, e.g. `unsigned long`. As spaces are not allowed in values, CIL will replace underscores by spaces: `size_t=unsigned_long`.
- sizeof(t) and alignof(t) are respectively the size and alignment of type t.
- The boolean-valued keys expect `true` for true, and `false` for false.
- The `src/machdep-ml.c` program will print a `CIL_MACHINE` value when run with the `--env` option.

As an example, here's the `CIL_MACHINE` value for 64-bit targets on an x86-based Mac OS X machine:

```
short=2,2 int=4,4 long=8,8 long_long=8,8 pointer=8,8 enum=4,4
float=4,4 double=8,8 long_double=16,16 void=1 bool=1,1 fun=1,1
alignof_string=1 max_alignment=16 size_t=unsigned_long
wchar_t=int char_signed=true const_string_literals=true
big_endian=false __thread_is_keyword=true
__builtin_va_list=true underscore_name=true
```

8 Library of CIL Modules

We are developing a suite of modules that use CIL for program analyses and transformations that we have found useful. You can use these modules directly on your code, or generally as inspiration for writing similar modules. A particularly big and complex application written on top of CIL is CCured (`../ccured/index.html`).

8.1 Control-Flow Graphs

The Cil.stmt datatype includes fields for intraprocedural control-flow information: the predecessor and successor statements of the current statement. This information is not computed by default. If you want to use the control-flow graph, or any of the extensions in this section that require it, you have to explicitly ask CIL to compute the CFG using one of these two methods:

8.1.1 The CFG module (new in CIL 1.3.5)

The best way to compute the CFG is with the CFG module. Just invoke Cfg.computeFileCFG on your file. The Cfg API describes the rest of actions you can take with this module, including computing the CFG for one function at a time, or printing the CFG in `dot` form.

8.1.2 Simplified control flow

CIL can reduce high-level C control-flow constructs like `switch` and `continue` to lower-level `goto`s. This completely eliminates some possible classes of statements from the program and may make the result easier to analyze (e.g., it simplifies data-flow analysis).

You can invoke this transformation on the command line with `--domakeCFG` or programatically with Cil.prepareCFG. After calling Cil.prepareCFG, you can use Cil.computeCFGInfo to compute the CFG information and find the successor and predecessor of each statement.

For a concrete example, you can see how `cilly --domakeCFG` transforms the following code (note the fall-through in case 1):

```
int foo (int predicate) {
  int x = 0;
  switch (predicate) {
    case 0: return 111;
    case 1: x = x + 1;
    case 2: return (x+3);
    case 3: break;
    default: return 222;
  }
  return 333;
}
```

See the CIL output for this code fragment

8.2 Data flow analysis framework

The Dataflow module (click for the ocamldoc) contains a parameterized framework for forward and backward data flow analyses. You provide the transfer functions and this module does the analysis. You must compute control-flow information (Section 8.1) before invoking the Dataflow module.

8.3 Inliner

The file ext/inliner.ml contains a function inliner.

8.4 Dominators

The module Dominators contains the computation of immediate dominators. It uses the Dataflow module.

8.5 Points-to Analysis

The module `ptranal.ml` contains two interprocedural points-to analyses for CIL: `Olf` and `Golf`. `Olf` is the default. (Switching from `olf.ml` to `golf.ml` requires a change in `Ptranal` and a recompiling `cilly`.)

The analyses have the following characteristics:

- Not based on C types (inferred pointer relationships are sound despite most kinds of C casts)
- One level of subtyping
- One level of context sensitivity (Golf only)
- Monomorphic type structures
- Field insensitive (fields of structs are conflated)
- Demand-driven (points-to queries are solved on demand)
- Handle function pointers

The analysis itself is factored into two components: `Ptranal`, which walks over the CIL file and generates constraints, and `Olf` or `Golf`, which solve the constraints. The analysis is invoked with the function `Ptranal.analyze_file: Cil.file -> unit`. This function builds the points-to graph for the CIL file and stores it internally. There is currently no facility for clearing internal state, so `Ptranal.analyze_file` should only be called once.

The constructed points-to graph supports several kinds of queries, including alias queries (may two expressions be aliased?) and points-to queries (to what set of locations may an expression point?).

The main interface with the alias analysis is as follows:

- `Ptranal.may_alias: Cil.exp -> Cil.exp -> bool`. If `true`, the two expressions may have the same value.
- `Ptranal.resolve_lval: Cil.lval -> (Cil.varinfo list)`. Returns the list of variables to which the given left-hand value may point.
- `Ptranal.resolve_exp: Cil.exp -> (Cil.varinfo list)`. Returns the list of variables to which the given expression may point.
- `Ptranal.resolve_funptr: Cil.exp -> (Cil.fundec list)`. Returns the list of functions to which the given expression may point.

The precision of the analysis can be customized by changing the values of several flags:

- `Ptranal.no_sub: bool ref`. If `true`, subtyping is disabled. Associated commandline option: **--ptr_unify**.

- `Ptranal.analyze_mono: bool ref`. (Golf only) If `true`, context sensitivity is disabled and the analysis is effectively monomorphic. Commandline option: **--ptr_mono**.
- `Ptranal.smart_aliases: bool ref`. (Golf only) If `true`, "smart" disambiguation of aliases is enabled. Otherwise, aliases are computed by intersecting points-to sets. This is an experimental feature.
- `Ptranal.model_strings: bool ref`. Make the alias analysis model string constants by treating them as pointers to chars. Commandline option: **--ptr_model_strings**
- `Ptranal.conservative_undefineds: bool ref`. Make the most pessimistic assumptions about globals if an undefined function is present. Such a function can write to every global variable. Commandline option: **--ptr_conservative**

In practice, the best precision/efficiency tradeoff is achieved by setting

```
Ptranal.no_sub = false
Ptranal.analyze_mono = true
Ptranal.smart_aliases = false
```

These are the default values of the flags.

There are also a few flags that can be used to inspect or serialize the results of the analysis.

- `Ptranal.debug_may_aliases`. Print the may-alias relationship of each pair of expressions in the program. Commandline option: **--ptr_may_aliases**.
- `Ptranal.print_constraints: bool ref`. If `true`, the analysis will print each constraint as it is generated.
- `Ptranal.print_types: bool ref`. If `true`, the analysis will print the inferred type of each variable in the program.

  If `Ptranal.analyze_mono` and `Ptranal.no_sub` are both `true`, this output is sufficient to reconstruct the points-to graph. One nice feature is that there is a pretty printer for recursive types, so the print routine does not loop.
- `Ptranal.compute_results: bool ref`. If `true`, the analysis will print out the points-to set of each variable in the program. This will essentially serialize the points-to graph.

8.6 StackGuard

The module `heapify.ml` contains a transformation similar to the one described in "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks", *Proceedings of the 7th USENIX Security Conference*. In essence it modifies the program to maintain a separate stack for return addresses. Even if a buffer overrun attack occurs the actual correct return address will be taken from the special stack.

Although it does work, this CIL module is provided mainly as an example of how to perform a simple source-to-source program analysis and transformation. As an optimization only functions that contain a dangerous local array make use of the special return address stack.

For a concrete example, you can see how `cilly --dostackGuard` transforms the following dangerous code:

```
int dangerous() {
  char array[10];
  scanf("%s",array); // possible buffer overrun!
}

int main () {
  return dangerous();
}
```

See the CIL output for this code fragment

8.7 Heapify

The module `heapify.ml` also contains a transformation that moves all dangerous local arrays to the heap. This also prevents a number of buffer overruns.

For a concrete example, you can see how `cilly --doheapify` transforms the following dangerous code:

```
int dangerous() {
  char array[10];
  scanf("%s",array); // possible buffer overrun!
}

int main () {
  return dangerous();
}
```

See the CIL output for this code fragment

8.8 One Return

The module `oneret.ml` contains a transformation the ensures that all function bodies have at most one return statement. This simplifies a number of analyses by providing a canonical exit-point.

For a concrete example, you can see how `cilly --dooneRet` transforms the following code:

```
int foo (int predicate) {
  if (predicate <= 0) {
    return 1;
  } else {
    if (predicate > 5)
      return 2;
    return 3;
  }
}
```

See the CIL output for this code fragment

8.9 Partial Evaluation and Constant Folding

The `partial.ml` module provides a simple interprocedural partial evaluation and constant folding data-flow analysis and transformation. This transformation always requires the `--domakeCFG` option. It performs:

- Constant folding even of compiler-dependent constants as, for example `sizeof(T)`.
- `if`-statement simplification for conditional expressions that evaluate to a constant. The `if`-statement gets replaced with the taken branch.
- Call elimination for
  1. empty functions and
  2. functions that return a constant.

  In case 1 the call disappears completely and in case 2 it is replaced by the constant the function returns.

Several commandline options control the behavior of the feature.

- `--partial_no_global_const`: Treat global constants as unknown values. This is the default.
- `--partial_global_const`: Treat global constants as initialized. Let global constants participate in the partial evaluation.

- `--partial_root_function` ıfunction-name: Name of the function where the simplification starts. Default: `main`.
- `--partial_use_easy_alias` Use Partial's built-in easy alias to analyze pointers. This is the default.
- `--partial_use_ptranal_alias` Use feature Ptranal to analyze pointers. Setting this option requires `--doptranal`.

For a concrete example, you can see how `cilly --domakeCFG --dopartial` transforms the following code (note the eliminated `if`-branch and the partial optimization of `foo`):

```
int foo(int x, int y) {
  int unknown;
  if (unknown)
    return y + 2;
  return x + 3;
}

int bar(void) {
  return -1;
}

int main(void) {
  int a, b, c;
  a = foo(5, 7) + foo(6, 7) + bar();
  b = 4;
  c = b * b;
  if (b > c)
    return b - c;
  else
    return b + c;
}
```

See the CIL output for this code fragment

8.10 Reaching Definitions

The `reachingdefs.ml` module uses the dataflow framework and CFG information to calculate the definitions that reach each statement. After computing the CFG (Section 8.1) and calling `computeRDs` on a function declaration, `ReachingDef.stmtStartData` will contain a mapping from statement IDs to data about which definitions reach each statement. In particular, it is a mapping from statement IDs to a triple the first two members of which are used internally. The third member is a mapping from variable IDs to Sets of integer options. If the set contains `Some(i)`, then the definition of that variable with ID `i` reaches that statement. If the set contains `None`, then there is a path to that statement on which there is no definition of that variable. Also, if the variable ID is unmapped at a statement, then no definition of that variable reaches that statement.

To summarize, reachingdefs.ml has the following interface:

- `computeRDs` – Computes reaching definitions. Requires that CFG information has already been computed for each statement.
- `ReachingDef.stmtStartData` – contains reaching definition data after `computeRDs` is called.
- `ReachingDef.defIdStmtHash` – Contains a mapping from definition IDs to the ID of the statement in which the definition occurs.
- `getRDs` – Takes a statement ID and returns reaching definition data for that statement.

- `instrRDs` – Takes a list of instructions and the definitions that reach the first instruction, and for each instruction calculates the definitions that reach either into or out of that instruction.
- `rdVisitorClass` – A subclass of nopCilVisitor that can be extended such that the current reaching definition data is available when expressions are visited through the `get_cur_iosh` method of the class.

8.11 Available Expressions

The `availexps.ml` module uses the dataflow framework and CFG information to calculate something similar to a traditional available expressions analysis. After `computeAEs` is called following a CFG calculation (Section 8.1), `AvailableExps.stmtStartData` will contain a mapping from statement IDs to data about what expressions are available at that statement. The data for each statement is a mapping for each variable ID to the whole expression available at that point(in the traditional sense) which the variable was last defined to be. So, this differs from a traditional available expressions analysis in that only whole expressions from a variable definition are considered rather than all expressions.

The interface is as follows:

- `computeAEs` – Computes available expressions. Requires that CFG information has already been comptued for each statement.
- `AvailableExps.stmtStartData` – Contains available expressions data for each statement after `computeAEs` has been called.
- `getAEs` – Takes a statement ID and returns available expression data for that statement.
- `instrAEs` – Takes a list of instructions and the availalbe expressions at the first instruction, and for each instruction calculates the expressions available on entering or exiting each instruction.
- `aeVisitorClass` – A subclass of nopCilVisitor that can be extended such that the current available expressions data is available when expressions are visited through the `get_cur_eh` method of the class.

8.12 Liveness Analysis

The `liveness.ml` module uses the dataflow framework and CFG information to calculate which variables are live at each program point. After `computeLiveness` is called following a CFG calculation (Section 8.1), `LiveFlow.stmtStartData` will contain a mapping for each statement ID to a set of `varinfo`s for varialbes live at that program point.

The interface is as follows:

- `computeLiveness` – Computes live variables. Requires that CFG information has already been computed for each statement.
- `LiveFlow.stmtStartData` – Contains live variable data for each statement after `computeLiveness` has been called.

Also included in this module is a command line interface that will cause liveness data to be printed to standard out for a particular function or label.

- `--doliveness` – Instructs cilly to comptue liveness information and to print on standard out the variables live at the points specified by `--live_func` and `live_label`. If both are ommitted, then nothing is printed.
- `--live_func` – The name of the function whose liveness data is of interest. If `--live_label` is ommitted, then data for each statement is printed.
- `--live_label` – The name of the label at which the liveness data will be printed.

8.13 Dead Code Elimination

The module `deadcodeelim.ml` uses the reaching definitions analysis to eliminate assignment instructions whose results are not used. The interface is as follows:

- `elim_dead_code` – Performs dead code elimination on a function. Requires that CFG information has already been computed (Section 8.1).
- `dce` – Performs dead code elimination on an entire file. Requires that CFG information has already been computed.

8.14 Simple Memory Operations

The `simplemem.ml` module allows CIL lvalues that contain memory accesses to be even futher simplified via the introduction of well-typed temporaries. After this transformation all lvalues involve at most one memory reference.

For a concrete example, you can see how `cilly --dosimpleMem` transforms the following code:

```
int main () {
  int ***three;
  int **two;
  ***three = **two;
}
```

See the CIL output for this code fragment

8.15 Simple Three-Address Code

The `simplify.ml` module further reduces the complexity of program expressions and gives you a form of three-address code. After this transformation all expressions will adhere to the following grammar:

```
basic::=
   Const _
   Addrof(Var v, NoOffset)
   StartOf(Var v, NoOffset)
   Lval(Var v, off), where v is a variable whose address is not taken
                     and off contains only "basic"

exp::=
   basic
   Lval(Mem basic, NoOffset)
   BinOp(bop, basic, basic)
   UnOp(uop, basic)
   CastE(t, basic)

lval ::=
   Mem basic, NoOffset
   Var v, off, where v is a variable whose address is not taken and off
               contains only "basic"
```

In addition, all `sizeof` and `alignof` forms are turned into constants. Accesses to arrays and variables whose address is taken are turned into "Mem" accesses. All field and index computations are turned into address arithmetic.

For a concrete example, you can see how `cilly --dosimplify` transforms the following code:

```
int main() {
  struct mystruct {
    int a;
    int b;
  } m;
  int local;
  int arr[3];
  int *ptr;

  ptr = &local;
  m.a = local + sizeof(m) + arr[2];
  return m.a;
}
```

See the CIL output for this code fragment

8.16 Converting C to C++

The module canonicalize.ml performs several transformations to correct differences between C and C++, so that the output is (hopefully) valid C++ code. This may be incomplete — certain fixes which are necessary for some programs are not yet implemented.

Using the `--doCanonicalize` option with CIL will perform the following changes to your program:

1. Any variables that use C++ keywords as identifiers are renamed.
2. C allows global variables to have multiple declarations and multiple (equivalent) definitions. This transformation removes all but one declaration and all but one definition.
3. `__inline` is #defined to `inline`, and `__restrict` is #defined to nothing.
4. C allows function pointers with no specified arguments to be used on any argument list. To make C++ accept this code, we insert a cast from the function pointer to a type that matches the arguments. Of course, this does nothing to guarantee that the pointer actually has that type.
5. Makes casts from int to enum types explicit. (CIL changes enum constants to int constants, but doesn't use a cast.)

8.17 Generating LLVM code (new in 1.3.7)

The llvm.ml module generates LLVM ("Low Level Virtual Machine", http://llvm.org) code from a CIL file. The current version only targets 32-bit x86 code (as of version 2.5, LLVM's 64-bit x86 support is still incomplete), and has a few significant limitations:

- No support for bitfields.
- No support for inline assembly.
- Ignores gcc pragmas and attributes (except those explicitly handled by CIL).
- No support for variable-sized types.

LLVM support is enabled by running `configure` with the `--with-llvm` option before compiling CIL. You can then convert C code to LLVM assembly with `cilly --dollvm somefile.c`. Or you can call `Llvm.generate` on a CIL file to get LLVM assembly as a `doc` string.

9 Controlling CIL

In the process of converting a C file to CIL we drop the unused prototypes and even inline function definitions. This results in much smaller files. If you do not want this behavior then you must pass the `--keepunused` argument to the CIL application.

Alternatively you can put the following pragma in the code (instructing CIL to specifically keep the declarations and definitions of the function `func1` and variable `var2`, the definition of type `foo` and of structure `bar`):

```
#pragma cilnoremove("func1", "var2", "type foo", "struct bar")
```

10 GCC Extensions

The CIL parser handles most of the `gcc` extensions and compiles them to CIL. The following extensions are not handled (note that we are able to compile a large number of programs, including the Linux kernel, without encountering these):

1. Nested function definitions.
2. Constructing function calls.
3. Naming an expression's type.
4. Complex numbers
5. Hex floats
6. Subscripts on non-lvalue arrays.
7. Forward function parameter declarations

The following extensions are handled, typically by compiling them away:

1. Attributes for functions, variables and types. In fact, we have a clear specification (see Section 6.4) of how attributes are interpreted. The specification extends that of `gcc`.
2. Old-style function definitions and prototypes. These are translated to new-style.
3. Locally-declared labels. As part of the translation to CIL, we generate new labels as needed.
4. Labels as values and computed goto. This allows a program to take the address of a label and to manipulate it as any value and also to perform a computed goto. We compile this by assigning each label whose address is taken a small integer that acts as its address. Every computed `goto` in the body of the function is replaced with a `switch` statement. If you want to invoke the label from another function, you are on your own (the `gcc` documentation says the same.)
5. Generalized lvalues. You can write code like `(a, b) += 5` and it gets translated to CIL.
6. Conditionals with omitted operands. Things like `x ? : y` are translated to CIL.
7. Double word integers. The type `long long` and the `LL` suffix on constants is understood. This is currently interpreted as 64-bit integers.
8. Local arrays of variable length. These are converted to uses of `alloca`, the array variable is replaced with a pointer to the allocated array and the instances of `sizeof(a)` are adjusted to return the size of the array and not the size of the pointer.
9. Non-constant local initializers. Like all local initializers these are compiled into assignments.
10. Compound literals. These are also turned into assignments.

11. Designated initializers. The CIL parser actually supports the full ISO syntax for initializers, which is more than both `gcc` and `MSVC`. I (George) think that this is the most complicated part of the C language and whoever designed it should be banned from ever designing languages again.

12. Case ranges. These are compiled into separate cases. There is no code duplication, just a larger number of `case` statements.

13. Transparent unions. This is a strange feature that allows you to define a function whose formal argument has a (tranparent) union type, but the argument is called as if it were the first element of the union. This is compiled away by saying that the type of the formal argument is that of the first field, and the first thing in the function body we copy the formal into a union.

14. Inline assembly-language. The full syntax is supported and it is carried as such in CIL.

15. Function names as strings. The identifiers `__FUNCTION__` and `__PRETTY_FUNCTION__` are replaced with string literals.

16. Keywords `typeof`, `alignof`, `inline` are supported.

11 CIL Limitations

There are several implementation details of CIL that might make it unusable or less than ideal for certain tasks:

- CIL operates after preprocessing. If you need to see comments, for example, you cannot use CIL. But you can use attributes and pragmas instead. And there is some support to help you patch the include files before they are seen by the preprocessor. For example, this is how we turn some `#define`s that we don't like into function calls.

- CIL does transform the code in a non-trivial way. This is done in order to make most analyses easier. But if you want to see the code `e1, e2++` exactly as it appears in the code, then you should not use CIL.

- CIL removes all local scopes and moves all variables to function scope. It also separates a declaration with an initializer into a declaration plus an assignment. The unfortunate effect of this transformation is that local variables cannot have the `const` qualifier.

12 Known Bugs and Limitations

12.1 Code that CIL won't compile

- We do not support tri-graph sequences (ISO 5.2.1.1).

- CIL cannot parse arbitrary `#pragma` directives. Their syntax must follow gcc's attribute syntax to be understood. If you need a pragma that does not follow gcc syntax, add that pragma's name to `no_parse_pragma` in `src/frontc/clexer.mll` to indicate that CIL should treat that pragma as a monolithic string rather than try to parse its arguments.

  CIL cannot parse a line containing an empty `#pragma`.

- CIL only parses `#pragma` directives at the "top level", this is, outside of any enum, structure, union, or function definitions.

  If your compiler uses pragmas in places other than the top-level, you may have to preprocess the sources in a special way (sed, perl, etc.) to remove pragmas from these locations.

- CIL cannot parse the following code (fixing this problem would require extensive hacking of the LALR grammar):

```
int bar(int ()); // This prototype cannot be parsed
int bar(int x()); // If you add a name to the function, it works
int bar(int (*)()); // This also works (and it is more appropriate)
```

- CIL also cannot parse certain K&R old-style prototypes with missing return type:

  ```
  g(); // This cannot be parsed
  int g(); // This is Ok
  ```

- CIL does not understand some obscure combinations of type specifiers ("signed" and "unsigned" applied to typedefs that themselves contain a sign specification; you could argue that this should not be allowed anyway):

  ```
  typedef signed char __s8;
  __s8 unsigned uchartest; // This is unsigned char for gcc
  ```

- CIL does not support constant-folding of floating-point values, because it is difficult to simulate the behavior of various C floating-point implementations in Ocaml. Therefore, code such as this will not compile:

  ```
  int globalArray[(1.0 < 2.0) ? 5 : 50]
  ```

- CIL uses Ocaml ints to represent the size of an object. Therefore, it can't compute the size of any object that is larger than 2^{30} bits (134 MB) on 32-bit computers, or 2^{62} bits on 64-bit computers.

12.2 Code that behaves differently under CIL

- GCC has a strange feature called "extern inline". Such a function can be defined twice: first with the "extern inline" specifier and the second time without it. If optimizations are turned off then the "extern inline" definition is considered a prototype (its body is ignored).

  With old versions of gcc, if optimizations are turned on then the extern inline function is inlined at all of its occurrences from the point of its definition all the way to the point where the (optional) second definition appears. No body is generated for an extern inline function. A body is generated for the real definition and that one is used in the rest of the file.

  With new versions of gcc, the extern inline function is used only if there is no actual (non-extern) second definition (i.e. the second definition is used in the whole file, even for calls between the extern inline definition and the second definition).

  By default, CIL follows the current gcc behavior for extern inline. However, if you set `oldstyleExternInline` to true, you will get an emulation of gcc's old behaviour (under the assumption that optimizations are enabled): CIL will rename your extern inline function (and its uses) with the suffix `__extinline`. This means that if you have two such definition, that do different things and the optimizations are not on, then the CIL version might compute a different answer! Also, if you have multiple extern inline declarations then CIL will ignore but the first one. This is not so bad because GCC itself would not like it.

- The implementation of `bitsSizeOf` does not take into account the packing pragmas. However it was tested to be accurate on cygwin/gcc-2.95.3, Linux/gcc-2.95.3 and on Windows/MSVC.

- `-malign-double` is ignored.

- The statement `x = 3 + x ++` will perform the increment of `x` before the assignment, while `gcc` delays the increment after the assignment. It turned out that this behavior is much easier to implement than gcc's one, and either way is correct (since the behavior is unspecified in this case). Similarly, if you write `x = x ++;` then CIL will perform the increment before the assignment, whereas GCC and MSVC will perform it after the assignment.

- Because CIL uses 64-bit floating point numbers in its internal representation of floating point numbers, `long double` constants are parsed as if they were `double` constants.

12.3 Effects of the CIL translation

- CIL cleans up C code in various ways that may suppress compiler warnings. For example, CIL will add casts where they are needed while `gcc` might print a warning for the missing cast. It is not a goal of CIL to emit such warnings — we support several versions of several different compilers, and mimicking the warnings of each is simply not possible. If you want to see compiler warnings, compile your program with your favorite compiler before using CIL.

- When you use variable-length arrays, CIL turns them into calls to `alloca`. This means that they are deallocated when the function returns and not when the local scope ends.

  Variable-length arrays are not supported as fields of a struct or union.

- In the new versions of `glibc` there is a function `__builtin_va_arg` that takes a type as its second argument. CIL handles that through a slight trick. As it parses the function it changes a call like:

  ```
  mytype x = __builtin_va_arg(marker, mytype)
  ```

  into

  ```
  mytype x;
  __builtin_va_arg(marker, sizeof(mytype), &x);
  ```

  The latter form is used internally in CIL. However, the CIL pretty printer will try to emit the original code.

  Similarly, `__builtin_types_compatible_p(t1, t2)`, which takes types as arguments, is represented internally as `__builtin_types_compatible_p(sizeof t1, sizeof t2)`, but the sizeofs are removed when printing.

13 Using the merger

There are many program analyses that are more effective when done on the whole program.

The merger is a tool that combines all of the C source files in a project into a single C file. There are two tasks that a merger must perform:

1. Detect what are all the sources that make a project and with what compiler arguments they are compiled.

2. Merge all of the source files into a single file.

For the first task the merger impersonates a compiler and a linker (both a GCC and a Microsoft Visual C mode are supported) and it expects to be invoked (from a build script or a Makefile) on all sources of the project. When invoked to compile a source the merger just preprocesses the source and saves the result using the name of the requested object file. By preprocessing at this time the merger is able to take into account variations in the command line arguments that affect preprocessing of different source files.

When the merger is invoked to link a number of object files it collects the preprocessed sources that were stored with the names of the object files, and invokes the merger proper. Note that arguments that affect the compilation or linking must be the same for all source files.

For the second task, the merger essentially concatenates the preprocessed sources with care to rename conflicting file-local declarations (we call this process alpha-conversion of a file). The merger also attempts to remove duplicate global declarations and definitions. Specifically the following actions are taken:

- File-scope names (`static` globals, names of types defined with `typedef`, and structure/union/enumeration tags) are given new names if they conflict with declarations from previously processed sources. The new name is formed by appending the suffix `___n`, where `n` is a unique integer identifier. Then the new names are applied to their occurrences in the file.

- Non-static declarations and definitions of globals are never renamed. But we try to remove duplicate ones. Equality of globals is detected by comparing the printed form of the global (ignoring the line number directives) after the body has been alpha-converted. This process is intended to remove those declarations (e.g. function prototypes) that originate from the same include file. Similarly, we try to eliminate duplicate definitions of `inline` functions, since these occasionally appear in include files.
- The types of all global declarations with the same name from all files are compared for type isomorphism. During this process, the merger detects all those isomorphisms between structures and type definitions that are **required** for the merged program to be legal. Such structure tags and typenames are coalesced and given the same name.
- Besides the structure tags and type names that are required to be isomorphic, the merger also tries to coalesce definitions of structures and types with the same name from different file. However, in this case the merger will not give an error if such definitions are not isomorphic; it will just use different names for them.
- In rare situations, it can happen that a file-local global in encountered first and it is not renamed, only to discover later when processing another file that there is an external symbol with the same name. In this case, a second pass is made over the merged file to rename the file-local symbol.

Here is an example of using the merger:
The contents of `file1.c` is:

```
struct foo; // Forward declaration
extern struct foo *global;
```

The contents of `file2.c` is:

```
struct bar {
 int x;
 struct bar *next;
};
extern struct bar *global;
struct foo {
 int y;
};
extern struct foo another;
void main() {
}
```

There are several ways in which one might create an executable from these files:

- `gcc file1.c file2.c -o a.out`
- ```
  gcc -c file1.c -o file1.o
  gcc -c file2.c -o file2.o
  ld file1.o file2.o -o a.out
  ```
- ```
  gcc -c file1.c -o file1.o
  gcc -c file2.c -o file2.o
  ar r libfile2.a file2.o
  gcc file1.o libfile2.a -o a.out
  ```
- ```
  gcc -c file1.c -o file1.o
  gcc -c file2.c -o file2.o
  ar r libfile2.a file2.o
  gcc file1.o -lfile2 -o a.out
  ```
  ```

In each of the cases above you must replace all occurrences of `gcc` and `ld` with `cilly --merge`, and all occurrences of `ar` with `cilly --merge --mode=AR`. It is very important that the `--merge` flag be used throughout the build process. If you want to see the merged source file you must also pass the `--keepmerged` flag to the linking phase.

The result of merging file1.c and file2.c is:

```
// from file1.c
struct foo; // Forward declaration
extern struct foo *global;

// from file2.c
struct foo {
 int x;
 struct foo *next;
};
struct foo___1 {
 int y;
};
extern struct foo___1 another;
```

14 Using the patcher

Occasionally we have needed to modify slightly the standard include files. So, we developed a simple mechanism that allows us to create modified copies of the include files and use them instead of the standard ones. For this purpose we specify a patch file and we run a program caller Patcher which makes modified copies of include files and applies the patch.

The patcher is invoked as follows:

```
lib/patcher [options]

Options:
  --help       Prints this help message
  --verbose    Prints a lot of information about what is being done
  --mode=xxx   What tool to emulate:
                GNUCC     - GNU CC
                MSVC      - MS VC cl compiler

  --dest=xxx   The destination directory. Will make one if it does not exist
  --patch=xxx  Patch file (can be specified multiple times)
  --ppargs=xxx An argument to be passed to the preprocessor (can be specified
               multiple times)

  --ufile=xxx  A user-include file to be patched (treated as \#include "xxx")
  --sfile=xxx  A system-include file to be patched (treated as \#include <xxx>)

  --clean       Remove all files in the destination directory
  --dumpversion Print the version name used for the current compiler

 All of the other arguments are passed to the preprocessor. You should pass
 enough arguments (e.g., include directories) so that the patcher can find the
 right include files to be patched.
```

Based on the given `mode` and the current version of the compiler (which the patcher can print when given the `dumpversion` argument) the patcher will create a subdirectory of the `dest` directory, such as:

```
/usr/home/necula/cil/include/gcc_2.95.3-5
```

In that file the patcher will copy the modified versions of the include files specified with the `ufile` and `sfile` options. Each of these options can be specified multiple times.

The patch file (specified with the `patch` option) has a format inspired by the Unix `patch` tool. The file has the following grammar:

```
<<< flags
patterns
===
replacement
>>>
```

The flags are a comma separated, case-sensitive, sequence of keywords or keyword = value. The following flags are supported:

- `file=foo.h` - will only apply the patch on files whose name is `foo.h`.
- `optional` - this means that it is Ok if the current patch does not match any of the processed files.
- `group=foo` - will add this patch to the named group. If this is not specified then a unique group is created to contain just the current patch. When all files specified in the command line have been patched, an error message is generated for all groups for whom no member patch was used. We use this mechanism to receive notice when the patch triggers are out-dated with respect to the new include files.
- `system=sysname` - will only consider this pattern on a given operating system. The "sysname" is reported by the "$Ô" variable in Perl, except that Windows is always considered to have sysname "cygwin." For Linux use "linux" (capitalization matters).
- `ateof` - In this case the patterns are ignored and the replacement text is placed at the end of the patched file. Use the `file` flag if you want to restrict the files in which this replacement is performed.
- `atsof` - The patterns are ignored and the replacement text is placed at the start of the patched file. Uf the `file` flag to restrict the application of this patch to a certain file.
- `disabled` - Use this flag if you want to disable the pattern.

The patterns can consist of several groups of lines separated by the `|||` marker. Each of these group of lines is a multi-line pattern that if found in the file will be replaced with the text given at the end of the block.

The matching is space-insensitive.

All of the markers `<<<`, `|||`, `===` and `>>>` must appear at the beginning of a line but they can be followed by arbitrary text (which is ignored).

The replacement text can contain the special keyword `@__pattern__@`, which is substituted with the pattern that matched.

15 Debugging support

Most of the time we debug our code using the Errormsg module along with the pretty printer. But if you want to use the Ocaml debugger here is an easy way to do it. Say that you want to debug the invocation of cilly that arises out of the following command:

```
cilly -c hello.c
```

You must follow the installation instructions to install the Elist support files for ocaml and to extend your .emacs appropriately. Then from within Emacs you do

```
ALT-X my-camldebug
```

This will ask you for the command to use for running the Ocaml debugger (initially the default will be "ocamldebug" or the last command you introduced). You use the following command:

```
cilly --ocamldebug -c hello.c
```

This will run `cilly` as usual and invoke the Ocaml debugger when the cilly engine starts. The advantage of this way of invoking the debugger is that the directory search paths are set automatically and the right set or arguments is passed to the debugger.

16 Who Says C is Simple?

When I (George) started to write CIL I thought it was going to take two weeks. Exactly a year has passed since then and I am still fixing bugs in it. This gross underestimate was due to the fact that I thought parsing and making sense of C is simple. You probably think the same. What I did not expect was how many dark corners this language has, especially if you want to parse real-world programs such as those written for GCC or if you are more ambitious and you want to parse the Linux or Windows NT sources (both of these were written without any respect for the standard and with the expectation that compilers will be changed to accommodate the program).

The following examples were actually encountered either in real programs or are taken from the ISO C99 standard or from the GCC's testcases. My first reaction when I saw these was: *Is this C?*. The second one was : *What the hell does it mean?*.

If you are contemplating doing program analysis for C on abstract-syntax trees then your analysis ought to be able to handle these things. Or, you can use CIL and let CIL translate them into clean C code.

16.1 Standard C

1. Why does the following code return 0 for most values of `x`? (This should be easy.)

   ```
   int x;
   return x == (1 && x);
   ```

   See the CIL output for this code fragment

2. Why does the following code return 0 and not -1? (Answer: because `sizeof` is unsigned, thus the result of the subtraction is unsigned, thus the shift is logical.)

   ```
   return (((1 - sizeof(int)) >> 16) >> 16);
   ```

   See the CIL output for this code fragment

3. Scoping rules can be tricky. This function returns 5.

   ```
   int x = 5;
   int f() {
     int x = 3;
     {
       extern int x;
       return x;
     }
   }
   ```

   See the CIL output for this code fragment

4. Functions and function pointers are implicitly converted to each other.

```
int (*pf)(void);
int f(void) {

   pf = &f; // This looks ok
   pf = ***f; // Dereference a function?
   pf(); // Invoke a function pointer?
   (****pf)();  // Looks strange but Ok
   (***************f)(); // Also Ok
}
```

   See the CIL output for this code fragment

5. Initializer with designators are one of the hardest parts about ISO C. Neither MSVC or GCC implement them fully. GCC comes close though. What is the final value of `i.nested.y` and `i.nested.z`? (Answer: 2 and respectively 6).

```
struct {
   int x;
   struct {
       int y, z;
   } nested;
} i = { .nested.y = 5, 6, .x = 1, 2 };
```

   See the CIL output for this code fragment

6. This is from c-torture. This function returns 1.

```
typedef struct
{
  char *key;
  char *value;
} T1;

typedef struct
{
  long type;
  char *value;
} T3;

T1 a[] =
{
  {
    "",
    ((char *)&((T3) {1, (char *) 1}))
  }
};
int main() {
   T3 *pt3 = (T3*)a[0].value;
   return pt3->value;
}
```

   See the CIL output for this code fragment

7. Another one with constructed literals. This one is legal according to the GCC documentation but somehow GCC chokes on (it works in CIL though). This code returns 2.

```
return ((int []){1,2,3,4})[1];
```

   See the CIL output for this code fragment

8. In the example below there is one copy of "bar" and two copies of "pbar" (static prototypes at block scope have file scope, while for all other types they have block scope).

```
int foo() {
   static bar();
   static (*pbar)() = bar;

}

static bar() {
  return 1;
}

static (*pbar)() = 0;
```

   See the CIL output for this code fragment

9. Two years after heavy use of CIL, by us and others, I discovered a bug in the parser. The return value of the following function depends on what precedence you give to casts and unary minus:

```
unsigned long foo() {
  return (unsigned long) - 1 / 8;
}
```

   See the CIL output for this code fragment

   The correct interpretation is `((unsigned long) - 1) / 8`, which is a relatively large number, as opposed to `(unsigned long) (- 1 / 8)`, which is 0.

10. An example with `typedef` wierdness. Example due to James Cheney.

```
typedef int int_t;
typedef int int2_t;

int_t f(int2_t int2_t[]) {
  int_t int_t = int2_t[0];
  {
    int int2_t = 2*int_t;
    return int2_t;
  }
}
```

   See the CIL output for this code fragment

16.2 GCC ugliness

1. GCC has generalized lvalues. You can take the address of a lot of strange things:

   ```
   int x, y, z;
   return &(x ? y : z) - & (x++, x);
   ```

   See the CIL output for this code fragment

2. GCC lets you omit the second component of a conditional expression.

   ```
   extern int f();
   return f() ? : -1; // Returns the result of f unless it is 0
   ```

   See the CIL output for this code fragment

3. Computed jumps can be tricky. CIL compiles them away in a fairly clean way but you are on your own if you try to jump into another function this way.

   ```
   static void *jtab[2]; // A jump table
   static int doit(int x){

     static int jtab_init = 0;
     if(!jtab_init) { // Initialize the jump table
       jtab[0] = &&lbl1;
       jtab[1] = &&lbl2;
       jtab_init = 1;
     }
     goto *jtab[x]; // Jump through the table
   lbl1:
     return 0;
   lbl2:
     return 1;
   }

   int main(void){
     if (doit(0) != 0) exit(1);
     if (doit(1) != 1) exit(1);
     exit(0);
   }
   ```

   See the CIL output for this code fragment

4. A cute little example that we made up. What is the returned value? (Answer: 1);

   ```
   return ({goto L; 0;}) && ({L: 5;});
   ```

   See the CIL output for this code fragment

5. `extern inline` is a strange feature of GNU C. Can you guess what the following code computes?

   ```
   extern inline foo(void) { return 1; }
   int firstuse(void) { return foo(); }

   // A second, incompatible definition of foo
   ```

```
int foo(void) { return 2; }

int main() {
    return foo() + firstuse();
}
```

See the CIL output for this code fragment

The answer depends on whether the optimizations are turned on. If they are then the answer is 3 (the first definition is inlined at all occurrences until the second definition). If the optimizations are off, then the first definition is ignore (treated like a prototype) and the answer is 4.

CIL will misbehave on this example, if the optimizations are turned off (it always returns 3).

6. GCC allows you to cast an object of a type T into a union as long as the union has a field of that type:

```
union u {
   int i;
   struct s {
      int i1, i2;
   } s;
};

union u x = (union u)6;

int main() {
  struct s y = {1, 2};
  union u  z = (union u)y;
}
```

See the CIL output for this code fragment

7. GCC allows you to use the `__mode__` attribute to specify the size of the integer instead of the standard `char`, `short` and so on:

```
int __attribute__ ((__mode__ (  __QI__ ))) i8;
int __attribute__ ((__mode__ (  __HI__ ))) i16;
int __attribute__ ((__mode__ (  __SI__ ))) i32;
int __attribute__ ((__mode__ (  __DI__ ))) i64;
```

See the CIL output for this code fragment

8. The "alias" attribute on a function declaration tells the linker to treat this declaration as another name for the specified function. CIL will replace the declaration with a trampoline function pointing to the specified target.

```
static int bar(int x, char y) {
  return x + y;
}

//foo is considered another name for bar.
int foo(int x, char y) __attribute__((alias("bar")));
```

See the CIL output for this code fragment

16.3 Microsoft VC ugliness

This compiler has few extensions, so there is not much to say here.

1. Why does the following code return 0 and not -1? (Answer: because of a bug in Microsoft Visual C. It thinks that the shift is unsigned just because the second operator is unsigned. CIL reproduces this bug when in MSVC mode.)

```
return -3 >> (8 * sizeof(int));
```

2. Unnamed fields in a structure seem really strange at first. It seems that Microsoft Visual C introduced this extension, then GCC picked it up (but in the process implemented it wrongly: in GCC the field `y` overlaps with `x`!).

```
struct {
  int x;
  struct {
     int y, z;
     struct {
       int u, v;
     };
 };
} a;
return a.x + a.y + a.z + a.u + a.v;
```

See the CIL output for this code fragment

17 Authors

The CIL parser was developed starting from Hugues Casse's `frontc` front-end for C although all the files from the `frontc` distribution have been changed very extensively. The intermediate language and the elaboration stage are all written from scratch. The main author is George Necula, with significant contributions from Scott McPeak, Westley Weimer, Ben Liblit, Matt Harren, Raymond To and Aman Bhargava.

This work is based upon work supported in part by the National Science Foundation under Grants No. 9875171, 0085949 and 0081588, and gifts from Microsoft Research. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or the other sponsors.

Starting with version 1.4.0, CIL is maintained by Gabriel Kerneis since the UC Berkeley does have time anymore to support it.

18 License

- and the CIL contributors for various patches.

19 Bug reports

We are certain that there are still some remaining bugs in CIL. If you find one please file a bug report in our Source Forge space http://sourceforge.net/projects/cil.

You can find there the latest announcements, a source distribution, bug report submission instructions and a mailing list: cil-users[at sign]lists.sourceforge.net. Please use this list to ask questions about CIL, as it will ensure your message is viewed by a broad audience.