A CIL Tutorial

Using CIL for language extensions and program analysis

Author:
Zachary Anderson
zanderso@acm.org

Systems Group
Department of Computer Science
ETH Zürich

January 7, 2013

Contents

Preface

The collection of techniques and code examples in this tutorial were developed over several years during my Ph.D. at Berkeley, and during my postdoctoral studies in the Systems Group at ETH Zürich. At ETH I used this tutorial and the accompanying project template to bring students up-to-speed in using CIL for their projects. I found that it was suitable for this purpose not only for students beginning their MS thesis work, but also for advanced undergrads in a course I taught on program analysis and transformation during the Spring semester of 2012.

My hope in sharing this tutorial is that it will help students of program analysis and programming language design have a quicker and smoother beginning toward building interesting and useful tools.

Good Luck,
Zachary Anderson
Zürich, Switzerland
January 7, 2013

Introduction

The C Intermediate Language(CIL) [8] is a source-to-source compiler for C. Since CIL is written in OCaml [6], and since it performs a number of simplifying transformations to the C AST, it is very well suited for use in rapid-prototyping new static and dynamic analyses, and for designing and trying out new language extensions. The purpose of this tutorial is to show how CIL can be used to construct a compiler front-end that performs additional analysis, or implements new language extensions. Through a series of examples, we will cover the basics of CIL's AST, its dataflow analysis framework, its facilities for instrumenting programs, the ease of extending C's type system, and how to employ a theorem prover in the compilation process. In the later chapters, the tutorial will scratch the surface of some more advanced techniques. These examples will hopefully point in the direction of full-fledged implementations of these advanced techniques, and serve as a template and starting point for your future projects.

Alternatives

Similar results could certainly be achieved by working directly with `gcc` or Clang/LLVM [1, 5], however the learning curve for these tools is much steeper, and the coding burden much higher. Furthermore, with Clang/LLVM in particular, there is no easy way to add custom statements to C, to make deep changes to its type-system, or to make changes at the level of the AST. Indeed, using Clang/LLVM, language extensions are typically written by interpreting new `#pragma` directives, e.g. in [9]; and new types are added by extending a complicated class hierarchy designed for speed rather than for ease of understanding or rapid prototyping. As we will see in this tutorial, with CIL, making these sorts of changes is straightforward. However, there is at least one downside with CIL: unlike working directly with `gcc`, or with Clang/LLVM, there is no support for C++. Language researchers and analysis designers must consider this trade-off when deciding on a compiler framework.

How to read this tutorial

This tutorial is written in the style of Literate Programming [4]. The source files in the `src`, `ciltut-lib`, and `test` directories may be compiled with the OCaml or C compilers as appropriate, and may be processed by `ocamlweb` [3] (in the case of OCaml code) and `pygmentize` [2] (in the case of C code) to produce the LaTeX code that defines this document. In the "template" version of this source tree, the comments that generate this document are omitted.

Each section begins with a brief introduction that explains the result to be achieved and the overall organization of the source code. This is followed by a heavily commented code listing. Occasionally, repetitive or obvious bits of code are omitted. In the source code, these omissions are indicated with ellipsis dots inside an OCaml comment, i.e. `(*...*)`. Interspersed with the comments are OCaml style tips, and suggestions for exercises and projects. The exercises and projects are intended to highlight how one might take the code in the tutorials and extend it to improve completeness or to add new features. Finally, at the end of some tutorials there is a bibliography with suggested further reading. The papers and books mentioned will either explain concepts from the tutorial in greater depth, or describe projects that have applied CIL using techniques from the tutorial.

We use the following textual conventions when referring to files, programs, modules, and variables. On first mention, files in the tutorial tree are referred to using their full path, e.g. `src/main.ml`. On first mention, files in the CIL tree are referred to using their full path rooted in the `cil` directory, e.g. `cil/src/cil.ml`. Program names are set as, e.g., `gcc`. Module names and variable names are typeset according to the conventions of `ocamlweb`.

Other Resources

The documentation in this tutorial is not intended as the sole source of information about CIL. In particular, we try to avoid duplicating information already contained in the official documentation [7], and the comments in the core CIL module in `cil/src/cil.mli`. Rather, this tutorial attempts to give hints about how to structure code, how to achieve some higher-level goals, and a few handy tricks that we have acquired in our own work while using CIL. Thus, if something seems ill-defined or unclear, please refer to CIL's official documentation.

Finally, the source code mentioned in this tutorial is available in a mercurial repository, and can be acquired by issuing the command:

```
$ hg clone https://bitbucket.org/zanderso/cil-template
```

The code in that repository can be viewed as a template for your own projects. It and CIL are both released under the BSD license.

References

[1] clang: a C language family frontend for LLVM, December 2011. http://clang.llvm.org/.

[2] Georg Brandl. Pygments—python syntax highlighter, December 2012. http://pygments.org/.

[3] Jean-Christophe Filliâtre and Claude Marché. ocamlweb: A literate programming tool for Objective caml, December 2011. http://www.lri.fr/~filliatr/ocamlweb/.

[4] Donald E. Knuth. *Literate Programming*. CSLI Lecture Notes Number 27. 1992.

[5] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[6] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jéròme Vouillon. The OCaml system: release 3.12, July 2011. http://caml.inria.fr.

[7] George C. Necula. CIL API documentation. http://www.cs.berkeley.edu/~necula/cil.

[8] George C. Necula, Scott McPeak, and Westley Weimer. CIL: Intermediate language and tools for the analysis of C programs. In *CC'04*, pages 213–228. http://cil.sourceforge.net/.

[9] Prakash Prabhu, Soumyadeep Ghosh, Yun Zhang, Nick P. Johnson, and David I. August. Commutative set: a language extension for implicit parallel programming. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 1–11, New York, NY, USA, 2011. ACM.

Chapter 0

Overview and Organization

Overall, this tutorial is organized as an OCaml program that uses CIL as a library. A Perl script sits in front of the resulting OCaml program to make the result resemble various off-the-shelf C compilers as much as possible. The code accompanying this tutorial provides nearly all of the necessary boilerplate, and will let you dive right in and start analyzing and manipulating the AST.

0.1 Source file to AST

Day-to-day, it is generally not needed to know how this boilerplate code produces an AST that we can work with in our OCaml program. It is good to understand how it works, though, in case your project requires you to modify the compilation process in some way, for example to force linking against a custom runtime library.

By default, CIL operates on one source file at a time (whole-program analysis is discussed by Chapter 13). The result of parsing a preprocesssed source file is a data-structure of type `Cil.file`. This structure contains a list of C definitions and declarations at global scope.

The details of parsing are handled in two places. Firstly in the Perl script mentioned above. It is called `cil/lib/Cilly.pm`, and is part of the CIL installation. For a project that uses the CIL library, like this one, we use our own Perl script, `lib/Ciltut.pm`, that derives from, overrides, and extends `Cilly.pm` to specialize it to our purposes[1]. This Perl script also takes care of running the preprocessor on the source file, interpreting some of the compiler flags, and performing the final link. Secondly, the OCaml program mentioned above reads in the preprocessed source file and passes it to `Frontc.parse_with_cabs`, which parses the file, and turns it into a `Cil.file`.

The details of the `Cil.file` instance are covered in Chapter 1. For now it should suffice to mention that the boilerplate code in `src/main.ml` takes care of setting up CIL's internal data-structures and options, parsing the input source file, and later on, emitting the `.cil.c` file that the perl script passes to `gcc`, or whatever your favorite C compiler happens to be.

Everything that happens in between parsing the input and emitting the output happens in source files like `src/tut0.ml`. Here, we write a function that takes a `Cil.file`, analyzes it, performs some

[1] This is where we could provide additional command line flags to the back-end compiler or to force linking with a custom runtime library, see the comments in that source file for details.

modifications to it, and optionally returns a result up to the caller in `main.ml`. We include this source file in the build by adding it to the list `CILTUT_SRC` in `CMakeLists.txt`.

Functions, like `tut0` below, that do this sort of work should be called from `processOneFile` in `main.ml`.

0.2 `tut0.ml`

The first step in writing an OCaml module that uses the CIL library is to `open` the `Cil` module (or bind it to an alias, like we do with the `Errormsg` module). The `CMakeLists.txt` file provided with this tutorial ensures that the CIL library is correctly found and linked assuming that it has been correctly installed. From now on, in the following chapters, the code to open modules and create aliases will be included only for the first mention of a module.

```
open Cil                                    (* The CIL library *)
module E = Errormsg                         (* CIL's error message library *)
```

`tut0` is a dummy function that doesn't do anything. However, there is certainly nothing preventing it from inspecting `f` and making changes to its mutable fields if it really wanted to.

```
let tut0 (f : file) : unit =
  E.log "I'n in tut0 and I could change %s if I wanted to!\n" f.fileName;
  ()
```

When the compiler front-end embodied by this tutorial is built and installed, the code in this module may be run by providing the `--enable-tut0` command line switch, like so:

```
$ ciltutcc --enable-tut0 -o program program.c
I'm in tut0 and I could change program.i if I wanted to!
```

Finally, if `ciltutcc` fails due to an unhandled exception, a stack trace may be obtained by re-executing the command as follows:

OCaml Style Note: Types are inferred, so you don't have to write them all the time. However, if you want to remember what your code does a month from now, writing in some of the types is a *Good Idea*.

```
$ OCAMLRUNPARAM=b ciltutcc --bytecode ...
```

Chapter 1

The AST

The *Concrete Syntax* of a program, is the textual representation of the program. An *Abstract Syntax Tree*(AST) is the in-memory data-structure that represents the parsed program. Parsing turns the concrete syntax into an AST. Parsing, the specification of grammars, and the various algorithms for lexical analysis and parsing, are covered in detail in many traditional textbooks on compilers. The textbooks by Appel [2] and Muchnick [3] are good starting points.

This chapter of the tutorial explains the structure of CIL's AST for C. The AST is defined in the `Cil` module in `cil/src/cil.ml` between lines 130 and 850. It might seem pretty complex at first glance, but you will find that it is very well documented, and the names of types, fields and constructors are logically assigned. Also, if you are familiar with other C or C++ front-ends, this AST is refreshingly straightforward. Finally, one of the key advantages to using CIL is that it makes a number of transformations that simplify analysis and instrumentation. Biggest among these is the absence of side-effects in expressions. Function calls and assignments are all lifted out of expressions into the `Cil.instr` nodes.

In this example code, we go digging through the AST looking for a function called `target`, when we find it, we filter out assignments to a global variable called `deleted`. In `src/tut2.ml` we'll see how to do this a bit more elegantly.

Coding Hint: When working with CIL, have two windows of source on your screen. One window is the project you are hacking on. The other window is `cil.ml`, so you can refer quickly to the AST type definitions and utility functions.

1.1 tut1.ml

The function `tut1FixInstr` returns `false` if the instruction `i` is an assignment to a global variable called `deleted`. Otherwise it returns `true`. We'll be using this function as an argument to `List.filter`.

```
let tut1FixInstr (i : instr) : bool =
  match i with
  | Set((Var vi, NoOffset), _, loc)
      when vi.vname = "deleted" ∧ vi.vglob →
    E.log "%a: Deleted assignment: %a\n" d_loc loc d_instr i;
    false
  | _ → true
```

The `tut1FixStmt` function inspects a statement. If the statement is a list of instructions, it overwrites the mutable `skind` field of the statement with a list of instructions filtered by `tut1FixInstr`. Otherwise, it recursively descends into sub-statements.

```
let rec tut1FixStmt (s : stmt) : unit =
  match s.skind with
  | Instr il →
    s.skind ← Instr(List.filter tut1FixInstr il)
  | If(_,tb,fb,_) →
    tut1FixBlock tb;
    tut1FixBlock fb
  (* and so forth.*)
  | _ → ()
and tut1FixBlock (b : block) : unit = List.iter tut1FixStmt b.bstmts
let tut1FixFunction (fd : fundec) : unit = tut1FixBlock fd.sbody
```

Now that we have defined a set of functions for traversing the AST of a function definition, we can write a function for the entry point into this module that iterates over all of the globals of a file. When we find the function called "target" we invoke `tut1FixFunction`.

```
let tut1 (f : file) : unit =
  List.iter (fun g →
    match g with
    | GFun (fd, loc) when fd.svar.vname = "target" →
      tut1FixFunction fd
    | _ → ())
  f.globals
```

1.2 Printing the AST

The function `tut1FixInstr` also uses CIL's logging and pretty-printing features. The call to `E.log` causes messages to be printed to `stdout`. It is similar to `Printf.printf` from the OCaml standard library with a few differences that are documented in `cil/ocamlutil/errormsg.mli`. Most importantly, you can print elements of the CIL AST by using the %a format specifier. Then, corresponding to that format you must supply two arguments: a function (`unit` $\rightarrow$ α $\rightarrow$ `doc`), and such an α that you want to print. Functions like this for the AST elements are defined in `cil.ml` (e.g. `Cil.d_exp` for `Cil.exp`'s). Functions for building objects of type `doc` are available in the `Pretty` module documented in `cil/ocamlutil/pretty.mli`.

1.3 test/tut1.c

The module `tut1.ml` deletes writes inside functions named "target" to global variables named "deleted". Thus, the function `target` below will return 37.

../test/tut1.c

```c
#include <stdio.h>

int deleted = 37;

int target()
{
  int l;
  deleted = 0;
  l = deleted;
  return l;
}

int main()
{
  int r;
  r = target();
  printf("r = %d\n", r);
  return 0;
}
```

We can build this file with the following command:

```
$ ciltutcc --enable-tut1 -o tut1 test/tut1.c
test/tut1.c:14:  Deleted assignment:  #line 14 "test/tut1.c"
deleted = 0;
```

Using the log message emitted by `tut1FixInstr`, `ciltutcc` informs us that it has removed an assignment to `deleted`. Now, we can run the resulting program:

```
$ ./tut1
r = 37
```

As expected, the assignment of 0 to `deleted` has been removed from the function `target`, and the program prints the original value.

References

[1] Andrew W. Appel. *Modern Compiler Implementation in ML.* Cambridge University Press, 1998.

[2] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

Chapter 2

Visiting the AST

This chapter explains the use of the visitor pattern [1] for traversing and modifying the CIL AST. The Visitor pattern defined in`cil.ml` takes care of the boilerplate code for traversing the AST, meaning that we don't have to write a bunch of mutually recursive functions every time we want to walk the AST like we did in Chapter 1.

The visitor pattern in CIL uses the O(bjective) features of OCaml. We inherit from `nopCilVisitor`, which simply walks the AST without doing anything. To do something other than that, we override the base class's methods.

In this example, we want to pull out assignments to a global variable, which is given as a parameter to `tut2`, from a function, which is also given as a parameter to `tut2`. Thus, we override the `vinst` ("visit instruction") method.

The result of one of the methods of the visitor class tells the visitor how to proceed next. There are four options for the result of a visitor method:

- `SkipChildren` — Stops the visitor from recursing into child AST nodes
- `DoChildren` — Directs the visitor to recurse into child AST nodes
- `ChangeTo x` — Replaces the AST node with `x`
- `ChangeDoChildrenPost(x, f)` — Like `ChangeTo x`, but runs `f` on the result of rebuilding `x` with the result of the visitor running on `x`'s children.

For `ChangeTo x` and `ChangeDoChildrenPost(x,f)`, check `nopCilVisitor` in `cil.ml` for the correct types for `x` and `f`. Further, note the following important difference between `ChangeTo x` and `ChangeDoChildrenPost(x,f)`: The visitor does *not* recurse into new nodes added with `ChangeTo x`. The visitor *does* however recurse into new nodes added with `ChnageDoChildrenPost(x,f)`, and additionally you can give a function(`f`) to apply to the result.

2.1 `tut2.ml`

In this module, we will first subclass CIL's default AST visitor, which simply returns `DoChildren` from every method. Then, we'll run the visitor over every function definition in the `Cil.file`.

```
open Tututil                                          (* for |> and onlyFunctions *)
```

`assignRmVisitor` is an OCaml class parameterized by the name of the variable whose assignment we are removing. The class inherits from `nopCilVisitor` as described above. `assignRmVisitor` overrides the `vinst` method. In our `vinst` method, we do case analysis on the incoming instruction. If it is an assignment (i.e. a `Set`) whose destination is the global variable that we're looking for, we use `ChangeTo` to replace the instruction with an empty list of instructions. If the instruction is any other assignment, a function call, or inline assembly, we leave it be with either `SkipChildren` or `DoChildren`.

```
class assignRmVisitor (vname : string) = object(self)
  inherit nopCilVisitor                    (* Inherit the default visitor, which does nothing *)
  method vinst (i : instr) =                (* Override the method for visiting instructions *)
    match i with
    | Set((Var vi, NoOffset), _, loc) when vi.vname = vname ∧ vi.vglob →
      E.log "%a: Assignment deleted: %a\n" d_loc loc d_instr i;
      ChangeTo []                                          (* Remove the Set instruction *)
    | _ → SkipChildren                      (* Don't care, could also say DoChildren here. *)
end
```

Now, the `processFunction` function is parameterized by the target function and variable names. If the function is the one we're looking for, we create an instance of `assignmRmVisitor` parameterized by the target variable name, and invoke the function `visitCilFunction` from `cil.ml`.

```
let processFunction ((tf, tv) : string × string) (fd : fundec) (loc : location) : unit =
  if fd.svar.vname ≠ tf then () else begin
    let vis = new assignRmVisitor tv in
    ignore(visitCilFunction vis fd)
  end
```

`tut2` is the external interface to this module. In a larger project, it would be appropriate to create a `.mli` file referring only to `tut2`. The function takes a pair of strings which are the target function and variable names along with the `Cil.file` to transform. The functions used to write `tut2` are described in detail below.

- `iterGlobals` is defined in `cil.ml`. It applies the function given by the first argument to every global in the file given by the second argument.
- `onlyFunctions` is defined in `src/tututil.ml`. It does nothing to globals that are not `GFun`'s, and passes the `fundec` and `loc` of `GFun`'s on to the function given by its argument.

Coding Hint: The infix function |> has type $(\alpha \rightarrow (\alpha \rightarrow \beta) \rightarrow \beta)$. The code in `tut2` is another way of writing `iterGlobals f (onlyFunctions (processFunction funvar))` but without all the extra parenthesis. |> is included in the F# standard library, but since it isn't in OCaml's we define it in `Tututil`. You can read it as "expression on the left is passed to function on right", and we can chain them up like I have in `tut2` without writing lots of extra parenthesis.

```
let tut2 (funvar : string × string) (f : file) : unit =
  funvar |> processFunction |> onlyFunctions |> iterGlobals f
```

2.2 test/tut2.c

In `main.ml` `tut2` in `tut2.ml` is called with argument `("foo","bar")` meaning that assignments to global variables called `bar` should be removed from functions called `foo`. Thus, when the code below is compiled with `tut2` enabled, the program should print `37` and exit.

../test/tut2.c

```
#include <stdio.h>

int bar = 37;

int foo()
{
  int l;
  bar = 0;
  l = bar;
  return l;
}

int main()
{
  int r;
  r = foo();
  printf("r = %d\n", r);
  return 0;
}
```

As with `test/tut1.c`, we can build this file with the following command:

```
$ ciltutcc --enable-tut2 -o tut2 test/tut2.c
test/tut2.c:16:  Deleted assignment:  #line 16 "test/tut2.c"
bar = 0;
```

Again, using the `E.log` function above, `ciltutcc` informs us that it has removed an assignment to `bar` on line 16. Now, we can run the resulting program:

```
$ ./tut2
r = 37
```

As expected, the assignment of 0 to `bar` has been removed from the function `foo`, and the program prints the original value.

2.3 Exercises

1. Take a look at the methods available for use in a visitor class. Rewrite this example without using `iterGlobals`. That is, use only the visitor class to do all the work.

References

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

Chapter 3

Dataflow Analysis

Dataflow Analyses can be used to calculate an abstraction of the program state at each program location. Dataflow analysis is a flow-sensitive analysis, meaning that, unlike flow-insensitive analysis, the order of the statements in the program is taken into account when calculating the abstraction of the state.

Traditionally, compiler writers use dataflow analyses extensively in performing compiler optimizations. For example, an *available expressions* dataflow analysis can eliminate redundant computation. Most compiler textbooks discuss dataflow analysis in detail [2, 3]. A more formal treatment can be found in the textbook by Nielson et al. [4].

CIL has a number of useful dataflow analyses built-in, which you can find in the directory `cil/src/ext`. They include a liveness analysis, an available expressions analysis, a reaching definitions analysis, and others. The utility of even a simple liveness analysis cannot be underestimated. Consider the following code snippet:

```
int *x = calloc(n, sizeof int);
f(x);
// x is never mentioned again
```

Since `x` does not escape the function, and since it is not live after the call to `f`, even a simple liveness analysis can conclude that `f` is taking ownership of the memory allocated on the first line.

In this tutorial we explore CIL's dataflow analysis features. In particular, we employ the following functor from CIL's `Dataflow` module found in `cil/src/ext/dataflow.ml`:

```
module ForwardsDataFlow (T : ForwardsTransfer) = struct ...,
```

which yields a module containing only a function called `compute`, which performs the dataflow analysis using the standard worklist algorithm. The functor is documented in detail in `cil/src/ext/dataflow.mli` and in the CIL documentation.

In order to use this functor, we must create an OCaml module implementing the `ForwardsTransfer` signature, which defines operations over the abstract state along with the transfer functions for the

static analysis. Now is a good time to take a look in `dataflow.mli` at the signature of the module that we'll be implementing, and what the functions of the module mean. Additionally, the `Dataflow` module is well documented in the main CIL documentation.

3.1 `tut3.ml`

The dataflow analysis here is a common textbook example for abstract interpretation, an even/odd analysis. First, we'll define types and operations over the abstract state of the program. Then, we'll apply the functor. Following this, we'll write some boilerplate code for accessing the results of the analysis in an AST visitor. It should be straightforward to repurpose the code in this tutorial for many other kinds of dataflow analysis, so feel free to use it as a starting point. For dataflow analysis in the backwards direction, there is also a `BackwardsDataFlow` functor in the `Dataflow` module.

```
module IH = Inthash                    (* An int → α hashtable library *)
module DF = Dataflow                   (* CIL's dataflow analysis library *)
```

When `debug` is true, the dataflow library emints out lots of debugging information.

```
let debug = ref false
```

3.1.1 Type Definitions

The abstract state for the analysis is a mapping from local variables of integral type to one of the `oekind` constructors. When a variable is mapped to one of these kinds, it has the following meaning:

- `Top` — The variable could be either odd or even.
- `Even` — The variable is an even integer.
- `Odd` — The variable is an odd integer.
- `Bottom` — The variable is uninitialized.

```
type oekind = Top | Odd | Even | Bottom
```

We'll use association lists to represent the mapping. An element of the mapping for a variable `vi : varinfo` is like: (`vi.vid`, (`vi`, `kind`)). We'll also need some utility functions for examining and manipulating the mappings.

Coding Hint: Whenever you define a new type `t`, write a `string_of_t` function, and if it makes sense, a `t_of_string` function as well. Here, we have to in order to implement one of the functions we need for the `ForwardsTransfer` signature, but it's also a good idea in general for at least two reasons. First, they come in handy for debugging, and second, they give you a sense of how easy the types you've defined will be to program with.

```
type varmap = int × (varinfo × oekind)
let id_of_vm (vm : varmap) : int = fst vm
let vi_of_vm (vm : varmap) : varinfo = vm |> snd |> fst
let kind_of_vm (vm : varmap) : oekind = vm |> snd |> snd
```

3.1.2 Functions for pretty printing the state

One of the functions we'll need to define for the `ForwardsDataFlow` functor is a function to pretty-print the abstract state. In particular, we must define a function that turns the abstract state into an instance of type `Pretty.doc`. Functions for constructing objects of this type are defined in the module `Pretty` in `pretty.ml`. For now, though, we'll just turn our abstract state into an OCaml `string`, and then use the function `Pretty.text : string → doc` to obtain a `doc`.

```
let string_of_oekind (k : oekind) : string =
  match k with
  | Top → "Top"
  | Odd → "Odd"
  | Even → "Even"
  | Bottom → "Bottom"
```

The `string_of_varmap` function converts a `varmap` tuple to a string:

```
let string_of_varmap (vm : varmap) : string =
  let vi = vi_of_vm vm in
  "("^vi.vname^", "^(vm |> kind_of_vm |> string_of_oekind)^")"
```

The `string_of_varmap_list` converts a list of `varmaps` into a comma-separated list:

```
let string_of_varmap_list (vml : varmap list) : string =
  vml
  |> L.map string_of_varmap
  |> String.concat ", "
```

The function `varmap_list_pretty` uses the functions `string_of_varmap_list` and `Pretty.text` to convert a `varmap list` to a `doc`:

```
let varmap_list_pretty () (vml : varmap list) =
  vml |> string_of_varmap_list |> text
```

3.1.3 Functions for manipulating kinds and states

Now, we must define some functions that manipulate lists of `varmap`s, culminating in the functions `varmap_list_combine` and `varmap_list_replace`. `varmap_list_combine` finds the least-upper-bound of two `varmap` lists. This function is responsible for determining the abstract state of the program at a join point. `varmap_list_replace` adds or replaces a mapping from the `varmap` list. It is used in the transfer function for assignments.

The function `oekind_neg` gets the opposite kind:

```
let oekind_neg (k : oekind) : oekind =
  match k with
  | Even → Odd
  | Odd → Even
  | _ → k
```

The function `varmap_equal` checks if two `varmap`'s are exactly the same:

```
let varmap_equal (vm1 : varmap) (vm2 : varmap) : bool =
  (id_of_vm vm1) = (id_of_vm vm2) ∧
  (kind_of_vm vm1) = (kind_of_vm vm2)
```

The function `varmap_list_equal` checks that two `varmap` lists are exactly the same. We sort the lists first because order doesn't matter.

```
let varmap_list_equal (vml1 : varmap list) (vml2 : varmap list) : bool =
  let sort = L.sort (fun (id1, _) (id2, _) → compare id1 id2) in
  list_equal varmap_equal (sort vml1) (sort vml2)
```

The function `oekind_includes` defines the partial order for our lattice:

- `Bottom`, `Odd`, `Even` < `Top`
- `Bottom` < `Odd`, `Even`
- But `Even` <> `Odd`

```
let oekind_includes (is_this : oekind) (in_this : oekind) : bool =
  match is_this, in_this with
  | _, Top → true
  | Bottom, _ → true
  | _, _ → false
```

The function `oekind_combine` defines the least-upper-bound(LUB) operation for the lattice: Anything joined with `Top` gives `Top`. `Even` joined with `Odd` also gives `Top`. `Odd` joined with itself or `Bottom` gives `Odd`, and likewise with `Even`. `Bottom` joined with `Bottom` gives `Bottom`.

```
let oekind_combine (k1 : oekind) (k2 : oekind) : oekind =
  match k1, k2 with
  | Top, _ | _, Top | Odd, Even | Even, Odd → Top
  | Odd, _ | _, Odd → Odd
  | Even, _ | _, Even → Even
  | Bottom, Bottom → Bottom
```

The function `varmap_combine` finds the LUB of two `varmap`s. It is undefined if the `varmap`s are not for the same variable. In that case, we return `None`.

```
let varmap_combine (vm1 : varmap) (vm2 : varmap) : varmap option =
  match vm1, vm2 with
  | (id1, _), (id2, _) when id1 ≠ id2 → None
  | (id1, (vi1, k1)), (_,(_,k2)) → Some(id1,(vi1,oekind_combine k1 k2))
```

The function `varmap_list_combine_one` finds the LUB of a `varmap` list with a single `varmap`. The function `forceOption : α option → α` returns the object wrapped up in a `Some` constructor, and throws an exception when passed `None`. However, we know that `varmap_combine` will return `Some` since we know that `vm` and `vm'` have the same `id`.

```
let varmap_list_combine_one (vml : varmap list) (vm : varmap) : varmap list =
  let id = id_of_vm vm in
  if L.mem_assoc id vml then
    let vm' = (id, L.assoc id vml) in
    let vm'' = forceOption (varmap_combine vm vm') in
    vm'' :: (L.remove_assoc (id_of_vm vm) vml)
  else vm :: vml
```

The function `varmap_list_combine` Finds the LUB of two `varmap` lists for a join point in the program.

```
let varmap_list_combine (vml1 : varmap list) (vml2 : varmap list) : varmap list =
  L.fold_left varmap_list_combine_one vml1 vml2
```

The function `varmap_list_replace` replaces the entry for a variable in the state, e.g. for an assignment.

```
let varmap_list_replace (vml : varmap list) (vm : varmap) : varmap list =
  vm :: (L.remove_assoc (id_of_vm vm) vml)
```

3.1.4 The `oekind` of an expression

These functions determine the `oekind` of CIL expressions by recursing over their sub-expressions. We also use CIL's built-in constant folding to obtain constants for some expressions, e.g. `SizeOf`.

The function `kind_of_int64` determine whether concrete integers are `Odd` or `Even`

```
let kind_of_int64 (i : Int64.t) : oekind =
  let firstbit = Int64.logand i Int64.one in
  if firstbit = Int64.one then Odd else Even
```

Depending on the abstract state, determine whether an expression `e` is for an `Odd` or `Even` integer. The function `oekind_of_exp` uses CIL's constant folding function to deal with things like `sizeof(type)`. Mutually recursive functions descend into unary and binary operations. Non-integer constants, addresses, and complex lvalues are translated as `Top`.

```
let rec oekind_of_exp (vml : varmap list) (e : exp) : oekind =
  match e with
  | Const(CInt64(i, _, _)) → kind_of_int64 i
  | Lval(Var vi, NoOffset) → vml |> L.assoc vi.vid |> snd
  | SizeOf _ | SizeOfE _ | SizeOfStr _ | AlignOf _ | AlignOfE _ →
    e |> constFold true |> oekind_of_exp vml
  | UnOp(uo, e, t) → oekind_of_unop vml uo e
  | BinOp(bo, e1, e2, t) → oekind_of_binop vml bo e1 e2
  | CastE(t, e) → oekind_of_exp vml e
  | _ → Top
```

The function `oekind_of_unop` determines whether the result of unary operation is `Odd` or `Even`. If the analysis were more clever, we *could* have said something about the `LNot` case, but for now we'll just say `Top`.

```
and oekind_of_unop (vml : varmap list) (u : unop) (e : exp) : oekind =
  match u with
  | Neg → oekind_of_exp vml e
  | BNot → e |> oekind_of_exp vml |> oekind_neg
  | LNot → Top
```

The function `oekind_of_binop` determines whether the result of a binary operation is `Odd` or `Even`. In this example we are handling only the cases for addition, subtraction, and multiplication, but it is also possible to handle other cases.

```
and oekind_of_binop (vml : varmap list) (b : binop) (e1 : exp) (e2 : exp) : oekind =
  let k1, k2 = oekind_of_exp vml e1, oekind_of_exp vml e2 in
  match b with
  | PlusA → begin
    match k1, k2 with
    | Even, Even → Even
    | Odd, Odd → Even
    | Even, Odd → Odd
    | Odd, Even → Odd
    | _, _ → Top
  end
    (* ... *)
  | _ → Top
```

If there is a write to memory or a function call, we must destroy anything we knew about a local variable whose address has been taken. In the case of a function call, if the analysis were to also handle global variables, it would also have to clear their state even if their addresses haven't been taken (but a global whose address hasn't been taken would still be safe from a memory write.) The function `varmap_list_kill` moves the state of variables whose address has been taken to `Top` in the resulting `varmap`.

```
let varmap_list_kill (vml : varmap list) : varmap list =
  L.map (fun (vid, (vi, k)) →
    if vi.vaddrof then (vid, (vi, Top)) else (vid, (vi, k)))
  vml
```

The function `varmap_list_handle_inst` implements the transfer function for our dataflow analysis. That is it looks at an instruction and a state, and calculates what the state should be after the instruction is run. On a simple assignment to a local variable, we replace its mapping in the input state with the `oekind` of the right-hand-side of the assignment. On memory writes, and function calls, we use `varmap_list_kill` to move the affected variables to `Top`.

```
let varmap_list_handle_inst (i : instr) (vml : varmap list) : varmap list =
  match i with
  | Set((Var vi, NoOffset), e, loc) when ¬(vi.vglob) ∧ (isIntegralType vi.vtype) →
    let k = oekind_of_exp vml e in
    varmap_list_replace vml (vi.vid,(vi,k))
  | Set((Mem _, _), _, _)
  | Call _ → varmap_list_kill vml                                    (* Kill vars with vaddrof set *)
  | _ → vml                          (* Not handling inline assembly. Ignoring writes to struct fields *)
```

The module `OddEvenDF` implements the `ForwardsTransfer` signature, and is the input module to the `DF.ForwardsDataFlow` functor. The meanings of its members are documented in detail in `dataflow.mli`. We include here only the interesting members. The function `combinePredecessors` is used to determine when the analysis should terminate. If newly calculated incoming state fails to change the old value, then it returns `None`, otherwise it calculates the join of the states and returns the result. The function `doInstr` calls our transfer function `varmap_list_handle_inst` to obtain the state after an instruction given the incoming state `ll`.

```
module OddEvenDF = struct
(*...*)
  let combinePredecessors (s : stmt) ~(old : t) (ll : t) =
    if varmap_list_equal old ll then None else
    Some(varmap_list_combine old ll)
  let doInstr (i : instr) (ll : t) =
    let action = varmap_list_handle_inst i in
    DF.Post action
(*...*)
end
```

Other members of the `ForwardsTransfer` signature may also be implemented to implement a more sophisticated analysis.

We finally invoke the functor to obtain the module `OddEven`, which contains the function `compute`, which performs the dataflow analysis.

```
module OddEven = DF.ForwardsDataFlow(OddEvenDF)
```

3.1.5 Running the analysis

The `collectVars` function determines the initial abstract state (`Bottom`) on entry to the first statement of a function.

```
let collectVars (fd : fundec) : varmap list =
  (fd.sformals @ fd.slocals)
  |> L.filter (fun vi → isIntegralType vi.vtype)
  |> L.map (fun vi → (vi.vid, (vi, Bottom)))
```

Here, the function `computeOddEven` calls the `compute` function of the `OddEven` module. First though, use CIL's `Cfg` module to compute the control-flow graph for the function. Then, we grab the first statement of the function and add it to the `stmtStartData` hash with a state indicating that on entry to the function the state for every variable is `Bottom`, which we get from the `collectVars` function. In practice, this code should be placed in a `try...with` block, to handle cases where, e.g. the function is empty.

```
let computeOddEven (fd : fundec) : unit =
  Cfg.clearCFGinfo fd;
  ignore(Cfg.cfgFun fd);
  let first_stmt = L.hd fd.sbody.bstmts in
  let vml = collectVars fd in
  IH.clear OddEvenDF.stmtStartData;
  IH.add OddEvenDF.stmtStartData first_stmt.sid vml;
  OddEven.compute [first_stmt]
```

3.1.6 Using the results

The function `getOddEvens` returns the state on entry to a statement by looking it up in the hash table `OddEvenDF.stmtStartData`, which is filled in by `OddEven.compute`.

```
let getOddEvens (sid : int) : varmap list option =
  try Some(IH.find OddEvenDF.stmtStartData sid)
  with Not_found → None
```

The function `instrOddEvens` calculates the states that exist on entry to each of a list of instructions. In addition to the list of instructions, it takes as input the state on entry to the first instruction in the list.

```
let instrOddEvens (il : instr list) (vml : varmap list) : varmap list list =
  let proc_one hil i =
    match hil with
    | [] → (varmap_list_handle_inst i vml) :: hil
    | vml' :: rst as l → (varmap_list_handle_inst i vml') :: l
  in
  il |> L.fold_left proc_one [vml]
     |> L.tl
     |> L.rev
```

Now that we can get the state on entry to statements and instructions, we can make a special visitor class such that when we inherit from it, the resulting visitor will have the current `OddEven` state available in every method.

The `vmlVisitorClass` class inherits from `nopCilVisitor`. When visiting a statement, if the statement is a list of instructions, it uses `instrOddEvens` to store the entry states into a mutable instance field `state_list`. If the statement is not an instruction list, it uses `getOddEvens` to put the state on entry to the statement into another mutable instance field `current_state`. When an instruction is visited, it takes the head of `state_list`, writes it to `current_state`, and removes the first state of `state_list`. The visitor also adds a method `get_cur_vml()`, which returns `current_state`.

Inheriting from this visitor will cause the state found by the analysis on entry to a statement or an instruction to be available in the `vstmt` and `vinst` methods of the inheriting class. Classes inheriting from `vmlVisitorClass` must call `super#vstmt` and `super#vinst` if they override `vstmt` or `vinst` respectively. To get the current state, inheriting classes can call `self#get_cur_vml`. Additionally, when passing an inheritor of `vmlVisitorClass` to a function requiring a `nopCilVisitor`, we must do an up-cast because of the additional method `get_cur_vml()`. See the function `evenOddAnalysis` below.

```
class vmlVisitorClass = object(self)
  inherit nopCilVisitor
  val mutable sid = -1
  val mutable state_list = []
  val mutable current_state = None
  method vstmt stm =
    sid ← stm.sid;
    begin match getOddEvens sid with
    | None → current_state ← None
    | Some vml → begin
      match stm.skind with
      | Instr il →
        current_state ← None;
        state_list ← instrOddEvens il vml
      | _ → current_state ← None
    end end;
```

```
    DoChildren
  method vinst i =
    try let data = L.hd state_list in
        current_state ← Some(data);
        state_list ← L.tl state_list;
        DoChildren
    with Failure "hd" → DoChildren
  method get_cur_vml () =
    match current_state with
    | None → getOddEvens sid
    | Some vml → Some vml
end
```

The class `varUseReporterClass` inherits from `vmlVisitorClass`. Whenever it visits a variable use, it emits the `oekind` it finds for it in the current state as given by `super#get_cur_vml()`.

```
class varUseReporterClass = object(self)
  inherit vmlVisitorClass as super
  method vvrbl (vi : varinfo) =
    match self#get_cur_vml () with
    | None → SkipChildren
    | Some vml → begin
      if L.mem_assoc vi.vid vml then begin
        let vm = (vi.vid, L.assoc vi.vid vml) in
        E.log "%a: %a\n" d_loc (!currentLoc) varmap_list_pretty [vm]
      end;
      SkipChildren
    end
end
```

The function `evenOddAnalysis` computes the `OddEven` analysis. It then invokes the `varUseReporter` visitor. Note that we have to coerce our visitor to a `nopCilVisitor` because we added the `get_cur_vml()` method.

```
let evenOddAnalysis (fd : fundec) (loc : location) : unit =
  computeOddEven fd;
  let vis = ((new varUseReporterClass) :> nopCilVisitor) in
  ignore(visitCilFunction vis fd)
```

The `tut3` function is the entry point to this module. It applies `evenOddAnalysis` to all functions.

```
let tut3 (f : file) : unit =
  iterGlobals f (onlyFunctions evenOddAnalysis)
```

3.2 test/tut3.c

The result of the analysis in `tut3.ml` will be to print a message to the console for each variable of integral type wherever it is used indicating whether it is even or odd at that program point. We consider the results of the analysis on the code below:

../test/tut3.c

```c
#include <stdio.h>

int main()
{
  int a,b,c,d;
  a = 1; b = 2; c = 3; d = 4;
  a += b + c;
  c *= d - b;
  b -= d + a;
  if (a % 2) a++;
  printf("a = %d, b = %d, c = %d, d = %d\n", a, b, c, d);
  return 0;
}
```

We build this source with the dataflow analysis enabled by doing:

```
$ ciltutcc --enable-tut3 -o tut3 test/tut3.c
test/tut3.c:14:  (a, Bottom)
test/tut3.c:14:  (b, Bottom)
test/tut3.c:14:  (c, Bottom)
test/tut3.c:14:  (d, Bottom)
test/tut3.c:15:  (a, Odd)
test/tut3.c:15:  (a, Odd)
test/tut3.c:15:  (b, Even)
test/tut3.c:15:  (c, Odd)
test/tut3.c:16:  (c, Odd)
test/tut3.c:16:  (c, Odd)
test/tut3.c:16:  (d, Even)
test/tut3.c:16:  (b, Even)
test/tut3.c:17:  (b, Even)
test/tut3.c:17:  (b, Even)
test/tut3.c:17:  (d, Even)
test/tut3.c:17:  (a, Even)
test/tut3.c:18:  (a, Even)
test/tut3.c:18:  (a, Even)
test/tut3.c:18:  (a, Even)
test/tut3.c:19:  (a, Top)
test/tut3.c:19:  (b, Even)
```

```
test/tut3.c:19:  (c, Even)
test/tut3.c:19:  (d, Even)
```

Since the program is small, we can check by inspection that the results of the analysis are correct. Note, however, that for the variable `a` at line 19, the result is `Top`. This occurs because the analysis made no attempt to interpret the guard of the `if`-statement on that line. Thus, at the join point after the `if`-statement, as far as this analysis knows, `a` could be either `Even` or `Odd`.

3.3 Exercises

1. Create an option in `src/ciltutoptions.ml` that controls whether debugging information for the analysis is printed.

2. We could represent the mapping from variables to `oekind`s many other ways. If the analysis had to be super-fast, we could use bitsets. Then, the abstract state would be four bitsets, one for each `oekind`, with each local variable of integral type belonging to exactly one of the bitsets. Note the `Bitmap` module in `cil/ocamlutil/bitmap.ml`.

3. In `oekind_of_unop` for the `LNot` case, there are two cases `e` $=$ 0 and `e` $\neq$ 0. If we know that `e` will be zero, then !`e` will be 1 and therefore `Odd`. If we know that `e` will be non-zero, then !`e` will be zero and therefore `Even`. If we don't know anything about `e`, then !`e` is `Top` as we have here. Modify this analysis to also track whether variables are zero or non-zero. (Such an analysis can have wider applications. Think about pointers.)

4. There are a number of other binary operations. Figure out how to handle them in `oekind_of_binop`.

   item Modify the analysis to handle not only local variables, but also global variables. What should their initial state be? How would the analysis change in the presence of multiple threads?

5. Implement the `doGuard` function of `OddEvenDF` so that the analysis may conclude that `a` is `Even` at line 19.

Project: Add another layer of generalization on top of CIL's `Dataflow` functors so that one must only define the functions operating on the abstract state (and possibly also a function for giving the initial state for a function.) One should then get a function for invoking the analysis and a visitor class for visiting the AST with the results of the analysis. (Essentially everything from the `module OddEvenDF` and down to the end of `vmlVisitorClass`, except for `collectVars`, should be automatic).

3.4 Further Reading

The analysis described above for CIL works only within a single procedure, but it can be very valuable to analyze a program across function calls. Without making approximations, the cost of adding context-sensitivity to an analysis can be very high. Thus, researchers are always coming up with clever new schemes to retain precision, while still performing the inter-procedural analysis efficiently. For a recent example see [1]. There is more in this tutorial on whole-program analysis with CIL in Chapter 13.

References

[1] Aws Albarghouthi, Rahul Kumar, Aditya V. Nori, and Sriram K. Rajamani. Parallelizing top-down interprocedural analyses. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 217–228, New York, NY, USA, 2012. ACM.

[2] Andrew W. Appel. *Modern Compiler Implementation in ML.* Cambridge University Press, 1998.

[3] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[4] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

Chapter 4

Instrumentation

In Chapters 1 and 2, we saw how to use the AST visitor to modify the program by removing an instruction. It was probably easy to see how to change that example to add statements and instructions, instead. However, adding function calls is a bit trickier. In particular, the `Call` constructor of the type `Cil.instr` requires a `Cil.exp` expression for the function. We can build an expression for the function out of a `varinfo` for the function, which we can find simply by iterating over the globals of a `Cil.file`. In this example we'll look at some handy patterns for accomplishing that, and for getting the instrumentation calls into the code.

It almost goes without saying that the ability insert function calls is imensely useful in the dynamic analysis of programs and the implementation of new language runtime features.

4.1 `tut4.ml`

First, we'll add calls to functions that will do some profiling of loops. At the end of each loop, the instrumented program will print the source location of the loop and the number of iterations that ran. First, we'll set up a quick way to refer to the instrumentation calls. Then, we'll write a visitor to wrap the calls around loops.

4.1.1 Setup of instrumentation calls

It isn't strictly necessary to wrap up the `varinfo`s for the instrumentation calls in a record type. They could just be global variables of this module, but I find this helps me stay a bit more organized. Also, if you add a new field to the record, the OCaml compiler gives you error messages if you forget some part of the initialization.

Additionally, in larger projects, I tend to put all of this set-up code in its own module, but in this small example we should be okay.

```
type functions = {
  mutable begin_loop : varinfo;
  mutable end_loop : varinfo;
}
```

A dummy varinfo that we can use to initialize the functions record

```
let dummyVar = makeVarinfo false "_tut_foo" voidType
```

`tutfuns` is an instance of the record type `functions` that lets us quickly refer to the `varinfos` for our instrumentation functions.

```
let tutfuns = {
  begin_loop = dummyVar;
  end_loop = dummyVar;
}
```

We set up some variables for the names of the instrumentation functions so that we only have one string to change if the name of the functions changes in the C code.

```
let begin_loop_str = "tut_begin_loop"
let end_loop_str = "tut_end_loop"
```

We may wish subsequent passes over the code to ignore calls to instrumentation that we have added. To make that easier, we define a list of the function names, `tut_function_names`, and a function `isTutFun` that tells if a function is an instrumentation call.

```
let tut_function_names = [
  begin_loop_str;
  end_loop_str;
]
let isTutFun (name : string) : bool =
  L.mem name tut_function_names
```

The function `mkFunTyp` is a helper function for making simple function types. It can be used when arguments and the function type itself have no attributes.

```
let mkFunTyp (rt : typ) (args : (string × typ) list) : typ =
  TFun(rt, Some(L.map (fun a → (fst a, snd a, [])) args), false, [])
```

The function `initTutFunctions` initializes the `tutfuns` record. It does this by calling the `Cil` function:

```
findOrCreateFunc : file → string → typ → varinfo.
```

If the function is found, the `varinfo` for it is returned, otherwise a prototype for the function is added to the file, and the `varinfo` for it is returned.

```
let initTutFunctions (f : file) : unit =
  let focf : string → typ → varinfo = findOrCreateFunc f in
  let bl_type = mkFunTyp voidType ["f", charConstPtrType; "l", intType] in
  let el_type = mkFunTyp voidType ["f", charConstPtrType; "l", intType; "c", intType;] in
  tutfuns.begin_loop ← focf begin_loop_str bl_type;
  tutfuns.end_loop ← focf end_loop_str el_type
```

4.1.2 Loop instrumentation

The function `makeInstrStmts` creates four statements that we'll add to loops. In the returned tuple, the first statement calls the `loop_begin` instrumentation function. The second statement initializes the iteration counter to zero. The third statement increments the iteration counter, and the fourth statement calls the `loop_end` instrumentation function.

In `makeInstrStmts` we use some shorthand. `mkString` turns an OCaml `string` into a `Cil.exp` constant expression. The `integer` function does the same for OCaml `int`s. The `v2e` function creates an `exp` out of a `varinfo`. The `var` function creates an `lval` from a `varinfo`, and the `i2s` function creates a `stmt` from an `instr`. All these can be found in `Tututil` or `Cil`.

```
let makeInstrStmts (counter : varinfo) (loc : location)
                   : stmt × stmt × stmt × stmt =
  let f, l = mkString loc.file, integer loc.line in
  i2s (Call(None, v2e tutfuns.begin_loop, [f; l], loc)),
  i2s (Set(var counter, zero, loc)),
  i2s (Set(var counter,BinOp(PlusA, v2e counter, one, counter.vtype), loc)),
  i2s (Call(None, v2e tutfuns.end_loop, [f; l; v2e counter], loc))
```

The class `loopInstrumenterClass` is a visitor that uses `makeInstrStmts` to instrument loops. In `vstmt` we update the statement `s` by writing to the mutable `skind` field instead of rebuilding a whole new statement so that we don't have to worry about copying over all the other fields, which we'd like to remain the same. In particular, it is very easy to forget about statement labels.

Further, we use `ChangeDoChildrenPost` because there might be nested loops. Remember: `ChangeDoChildrenPost` recurses into `s`'s children before handling `s`. That way, we have no infinite loops due to turning `s` into a statement that contains the original `s`.

```
class loopInstrumenterClass (fd : fundec) = object(self)
  inherit nopCilVisitor
```

```
  method vstmt (s : stmt) =
    let action s =
      match s.skind with
      | Loop(b, loc, co, bo) →
        let counter = makeTempVar fd intType in
        let ss, cis, is, es = makeInstrStmts counter loc in
        b.bstmts ← is :: b.bstmts;
        let nb = mkBlock [ss; cis; mkStmt s.skind; es] in
        s.skind ← Block nb;
        s
      | _ → s
    in
    ChangeDoChildrenPost(s, action)
end
```

The function `processFunction` applies the `loopInstrumenterClass` to a function.

```
let processFunction (fd : fundec) (loc : location) : unit =
  let vis = new loopInstrumenterClass fd in
  ignore(visitCilFunction vis fd)
```

The function `tut4` is the entry point in the module. It applies `processFunction` to every function in a `file`.

```
let tut4 (f : file) : unit =
  initTutFunctions f;
  iterGlobals f (onlyFunctions processFunction)
```

Now we have added a bunch of function calls to the code. Several questions remain:

- Where are these functions defined? — In `ciltut-lib/src/tut4.c`
- How does that file get built? — We use `cmake` to configure and build the library `libciltut`. The build of the library can be adjusted by modifying the `CMakeLists.txt` files under the `ciltut-lib` directory. The `Makefile` generated by `cmake` is invoked from the root `Makefile` when `ciltutcc` is built.
- How does that library get linked in to something that `ciltutcc` is building? — There is a Perl script in `lib/Ciltut.pm` that wraps up this OCaml program to make it look like `gcc` (and a couple other compilers). It has a function called `processArguments` where we add the library to a list in `@{$self → {CILTUTLIBS}}`. When this Perl script detects that it is being used for the link stage of a build, it adds `libciltut.a` to the list of object files to link in.

4.2 tut4.c

In this C file, we define the two functions added by `src/tut4.ml`. `tut_begin_loop` currently does nothing, but a more sophisticated analysis may wish to do some bookkeeping before each loop, so we include it as a placeholder. `tut_end_loop` simply outputs the number of iterations that a loop completed. We use `c-1` because the counter is incremented at the top of the loop before the exit test.

../ciltut-lib/src/tut4.c

```
#include <stdio.h>

void tut_begin_loop(const char *f, int l) {}

void tut_end_loop(const char *f, int l, int c)
{
  printf("loop: %s:%d - %d times\n", f, l, c - 1);
  fflush(stdout);
  return;
}
```

4.3 test/tut4.c

We can test the instrumentation on the small example below. At the end of each loop the instrumented code will print a message stating how many iterations there were. Thus, we should get 10 messages for the inner loop stating that there were 5 iterations, and one message for the outer loop stating that there were 10 iterations.

../test/tut4.c

```
#include <stdio.h>

int main()
{
  int i, j;
  int c = 1;

  for (i = 0; i < 10; i++) {
    for (j = 0; j < 5; j++) {
      c *= i;
    }
  }

  return 0;
}
```

Now, we can build the test program by doing the following:

```
$ ciltutcc --enable-tut4 -o tut4 test/tut4.c
```

Then, when we run `tut4`, we get the following output:

```
$ ./tut4
loop:  test/tut4.c:18 - 5 times
loop:  test/tut4.c:18 - 5 times
loop:  test/tut4.c:18 - 5 times
loop:  test/tut4.c:18 - 5 times
loop:  test/tut4.c:18 - 5 times
loop:  test/tut4.c:18 - 5 times
loop:  test/tut4.c:18 - 5 times
loop:  test/tut4.c:18 - 5 times
loop:  test/tut4.c:18 - 5 times
loop:  test/tut4.c:18 - 5 times
loop:  test/tut4.c:17 - 10 times
```

which is what we expected.

4.4 Exercises

1. Count the number of times: a variable is read/written, a particular instruction or statement is executed, how many times a branch is taken one way or the other, etc.

Chapter 5

Interpreted Constructors

CIL has support for interpreted constructors. That is, instead of building up new AST nodes using the OCaml-language constructors, we can write something that looks almost like C. Then, CIL will parse it, and generate the AST for us. This is particularly useful when you want to add for-loops to code. The official CIL documentation explains interpreted constructors in detail, but this complete working example might better show how they are useful.

5.1 `tut5.ml`

In this example, we'll add code to the top of a function that will initialize pointers to `NULL`, including pointers in stack-allocated `structs`, and arrays of those structs. But, since we don't want to slow the program down too much, we only want to write the pointer fields, so we can't use `memset`, or `bzero`.

We begin by defining a set of mutually recursive functions that traverse a type and generate a list of statements that `NULL` out pointers in an instance of that type. When we come to array types in the function `zeroArray`, we'll use one of CIL's interpreted constructors to make the loop over the array.

Assuming that `blv` is of composite type, `compinfoOfLval` returns the coresponding `compinfo`. Since the function should only ever be called on lvals of composite type, if it is not, we emit a bug message, and halt compilation by raising an exception with `E.s`.

```
let compinfoOfLval (blv : lval) : compinfo =
  match unrollType(typeOfLval blv) with
  | TComp (ci, _) → ci
  | _ → E.s(E.bug "Expected TComp for type of %a" d_lval blv)
```

The function `zeroPtr` makes a one-instruction statement for `NULL`ing out a pointer. The caller must check that `blv` is of pointer type.

Show-stopping Errors and Bugs: The function `Errormsg.s` ignores its argument and throws the exception `Errormsg.Error`, which is only caught in `main.ml`. Thus, executing `E.s(E.bug ...)` or `E.s(E.error ...)` has the effect of stopping compilation with the given error message. If compilation could continue despite the problem—and a sensible value with the correct type could be generated—then `E.bug` and `E.error` could be used without `E.s`. Then, compilation will continue, possibly generating further error messages.

```
let zeroPtr (fd : fundec) (blv : lval) : stmt list =
  [i2s (Set(blv, CastE(voidPtrType, zero), locUnknown))]
```

The function `zeroType` is the entry point for `NULL`ing out pointers found by traversing the lvalue `blv`. If `blv` is a pointer, we `NULL` it out. If it is an array, we loop over the array `NULL`ing any pointers found in traversing the base type. If it is a `struct` type, we `NULL` any pointers found in traversing the fields.

```
let rec zeroType (fd : fundec) (blv : lval) : stmt list =
  match unrollType(typeOfLval blv) with
  | TPtr _ → zeroPtr fd blv
  | TArray _ → zeroArray fd blv
  | TComp _ → zeroComp fd blv
  | _ → []
```

The function `zeroComp` `NULL`s out the pointer fields of `blv` assuming `blv` has a `TComp` type. If the type of `blv` is a union, we just `NULL` out the first pointer field.

```
and zeroComp (fd : fundec) (blv : lval) : stmt list =
  let ci = compinfoOfLval blv in
  let sl =
    ci.cfields
    |> L.map (zeroField fd blv)
    |> L.concat
  in
  if ci.cstruct then sl
  else if sl ≠ [] then [L.hd sl]
  else []
```

The function `zeroField` is used by `zeroComp` when iterating over the fields of a `compinfo`. It tacks the field onto `blv` using the `Cil` function `addOffsetLval` and `NULL`s out any pointers in it by calling `zeroType`.

```
and zeroField (fd : fundec) (blv : lval) (fi : fieldinfo) : stmt list =
  zeroType fd (addOffsetLval (Field(fi,NoOffset)) blv)
```

`zeroArray` NULLs out pointers in the array `blv`. We first make a temporary variable `i`, and generate the body of the loop by calling `zeroType` on `blv[i]`. Further, we create an lvalue, `first` that refers to the first element of the array. We'll need this in order to calculate the number of elements of the array. Next, we write the interpreted constructor following the grammer in the CIL documentation.

```
and zeroArray (fd : fundec) (blv : lval) : stmt list =
  let i = makeTempVar fd intType in
  let inits = zeroType fd (addOffsetLval (Index(v2e i, NoOffset)) blv) in
  let first = addOffsetLval (Index(zero, NoOffset)) blv in
  Formatcil.cStmts
    "
          %l:i = sizeof(%l:arr) / sizeof(%l:first) - 1;
          while (%l:i >= 0) {
            { %S:inits }
            %l:i -= 1;
          }
        "
    (fun n t → makeTempVar fd ~name:n t) locUnknown
    [ ("i", Fl(var i));
      ("arr", Fl blv);
      ("first", Fl first);
      ("inits", FS inits);]
```

`processFunction` iterates over the local variables of a function and adds the statements that result from applying `zeroType` to each of them to the start of the function.

```
let processFunction (fd : fundec) (loc : location) : unit =
  let ini_stmts =
    fd.slocals
    |> L.map var
    |> L.map (zeroType fd)
    |> L.concat
  in
  fd.sbody.bstmts ← ini_stmts @ fd.sbody.bstmts
```

Finally, the entry point `tut5` iterates over the functions applying `processFunction` to each of them.

```
let tut5 (f : file) : unit =
  iterGlobals f (onlyFunctions processFunction)
```

5.2 test/tut5.c

In this example, we define a few structure types containing pointers. Then in the `main` function, we declare an array of 37 `struct baz`'s. The effect of the code in `tut5.ml` should be to generate a loop that `NULL`s out all the pointer fields in the `struct baz` array. We will verify this by looking at the output of `ciltutcc` before it goes to the back-end compiler.

../test/tut5.c

```
struct foo {
  int *a, b, *c;
};

struct bar {
  struct foo f;
  int *a, b;
};

struct baz {
  struct bar b;
  int a, *c;
};

int main()
{
  struct baz b[37];
  int i;

  for (i = 0; i < 37; i++) {
    b[i].a = 3;
  }

  return 0;
}
```

Now, we can run the compiler:

```
$ ciltutcc --enable-tut5 --save-temps -o tut5 test/tut5.c
```

The `-save-temps` flag instructs `ciltutcc` to retain its intermediate results. In particular, the instrumented code is placed in the file `tut5.cil.c` in the directory where the compiler was invoked. Looking in this file, we see that the following code has been generated to initialize the pointer fields of `b`, and added to the `main` function:

../test/tut5.out.c

```
__cil_tmp3 = sizeof(b) / sizeof(b[0]) - 1;
while (__cil_tmp3 >= 0) {
  b[__cil_tmp3].b.f.a = (void *)0;
  b[__cil_tmp3].b.f.c = (void *)0;
  b[__cil_tmp3].b.a = (void *)0;
  b[__cil_tmp3].c = (void *)0;
  __cil_tmp3 --;
}
```

Chapter 6

Overriding Functions

When performing a dynamic analysis, it happens frequently that we would like to intercept calls to the C Library, or to system calls. We can accomplish this using the dynamic linker [1].

Using the dynamic linker is preferable to using CIL to replace calls to these library functions. If you use CIL, the call is only replaced in the code that CIL touches. Other code that is linked into the program will use the original version. This will either have strange effects or render your dynamic analysis unsound. Thus, if you want to override a library or system call, don't use CIL; instead, use the dynamic linker as shown below.

6.1 `tut6.ml`

There's nothing going on here in the OCaml module corresponding to this tutorial because all of the action happens in `ciltut-lib/src/tut6.c`.

6.2 Overriding Library Calls

Instead, we'll take this opportunity to discuss a few ways that overriding library calls can be useful.

- You can override `pthread_create` to keep extra state for threads spawned by the application you are analyzing.
- You can override `malloc` and friends to analyze and profile memory allocation.
- You can override system calls that request resources from the Operating System, like cores, or memory, or I/O bandwidth, in order to shape the demands placed on the system.
- ...and many others.

[1] Intercepting system calls can be a bit more complicated than intercepting library calls. Applications may directly use the `syscall` system call instead of the C Library interface. Luckily, one may still wrap this call, with a bit more work.

```
open Cil
let tut6 (f : file) : unit = ()
```

6.3 tut6.c

In `tut6.c`, we demonstrate how to use the dynamic linker to override library functions. In particular, we wrap calls to `pthread_mutex_lock` and `pthread_mutex_unlock`. We'll do this by calling `dlsym` found in `dlfcn.h` with the flag `RTLD_NEXT`, which fetches a pointer to the original function we are overriding. Below are the includes we'll need.

../ciltut-lib/src/tut6.c

```
#define _GNU_SOURCE  // Needed for RTLD_NEXT
#include <stdio.h>   // For printf
#include <dlfcn.h>   // for RTLD_NEXT
#include <pthread.h> // for pthread_*
#include <ciltut.h>  // for checked_dlsym
```

First, we set up function pointers at global scope to point at the original versions of the functions, which we'll call from inside of our wrappers.

../ciltut-lib/src/tut6.c

```
static int (*pthread_mutex_lock_orig)  (pthread_mutex_t *m) = NULL;
static int (*pthread_mutex_unlock_orig)(pthread_mutex_t *m) = NULL;
```

We'll also declare a flag so that we can enable and disable lock tracing.

../ciltut-lib/src/tut6.c

```
static int enable_lock_tracking = 0;
```

With the pointers to the original functions declared, we can now write the wrappers. The wrappers have the same name as the original functions. This way, calls to the functions will be routed to these versions. The first thing that the wrapper functions do is set up the pointers to the original calls using `checked_dlsym`. `checked_dlsym` will abort the program if the function named by the string does not exist. After ensuring that the original functions exist, the wrappers may then execute whatever actions are needed to implement their purpose, and to call the original functions.

../ciltut-lib/src/tut6.c

```
int pthread_mutex_lock(pthread_mutex_t *m)
{
  int res;
  if (!pthread_mutex_lock_orig)
    pthread_mutex_lock_orig = checked_dlsym(RTLD_NEXT, "pthread_mutex_lock");
  res = pthread_mutex_lock_orig(m);
  if (enable_lock_tracking) {
    printf("thread: %d - pthread_mutex_lock(%p)\n", gettid(), m);
    fflush(stdout);
  }
  return res;
}

int pthread_mutex_unlock(pthread_mutex_t *m)
{
  int res;
  if (!pthread_mutex_unlock_orig)
    pthread_mutex_unlock_orig = checked_dlsym(RTLD_NEXT, "pthread_mutex_unlock");
  if (enable_lock_tracking) {
    printf("thread: %d - pthread_mutex_unlock(%p)\n", gettid(), m);
    fflush(stdout);
  }
  res = pthread_mutex_unlock_orig(m);
  return res;
}

void toggle_lock_tracking()
{
  enable_lock_tracking = !enable_lock_tracking;
}
```

6.4 test/tut6.c

In this test, we simply declare a lock at global scope, then acquire and release it in the main function. We do this simply to test that the wrapper functions we wrote in `ciltut-lib/src/tut6.c` are correctly run whenever we call the lock and unlock functions.

../test/tut6.c

```c
#include <pthread.h>
#include <ciltut.h>

int counter = 0;
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;

int main()
{
  toggle_lock_tracking();
  pthread_mutex_lock(&mtx);
  counter++;
  pthread_mutex_unlock(&mtx);
  return 0;
}
```

Now, we can compile this test by doing:

```
$ ciltutcc --enable-tut6 -lpthread -o tut6 test/tut6.c
```

Then, when we run `tut6`, we should get something similar to the following output:

```
$ ./tut6
thread:  32646 - pthread_mutex_lock(0x602180)
thread:  32646 - pthread_mutex_unlock(0x602180)
```

6.5 Further Reading

Many researchers have studied overriding memory allocation calls (along with the instrumentation of some memory accesses) with the goal of catching memory management errors. Examples are Libsafe [2], LibsafePlus [1], and HeapShield [3] (which is a part of DieHard [4]).

The Trickle [5] userspace bandwidth shaper operates by using the dynamic linker to override IO calls with ones that track the number of bytes sent per unit time. Then, it puts the calling process to sleep when the sending or receiving rate exceeds a user given limit.

References

[1] Kumar Avijit, Prateek Gupta, and Deepak Gupta. TIED, libsafeplus: tools for runtime buffer overflow protection. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 4–4, Berkeley, CA, USA, 2004. USENIX Association.

[2] A. Baratloo, N. Singh, and T. Tsai. Transparent run-time defense against stack smashing attacks. In *Proceedings of the 2000 USENIX Annual Technical Conference*, USENIX'00, 2000.

[3] Emery D. Berger. Heapshield: Library-based heap overflow protection for free. Technical Report UMass CS TR 06-28, Department of Computer Science, University of Massachusetts, 2006.

[4] Emery D. Berger and Benjamin G. Zorn. Diehard: probabilistic memory safety for unsafe languages. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 158–168, New York, NY, USA, 2006. ACM.

[5] Marius A. Eriksen. Trickle: A userland bandwidth shaper for unix-like systems. In *Proceedings of the USENIX 2005 Annual Technical Conference, FREENIX Track*, USENIX'05, pages 61–70, 2005.

Chapter 7

Type Qualifiers

Over the next three chapters, we'll explore how to make changes to C's type-system. This will be achieved by adding type-qualifiers to C's types, and by performing some extra type-checking. In this section, we'll write a very basic type-checker for types that may be qualified by one or more of the following colors: red, green, or blue. In the exercises, you'll find suggestions about how to complete it. In the next section, we'll look at interpreting dependent type qualifiers. Finally, in Chapter 9, we'll see how to do some basic type qualifier inference.

7.1 `tut7.ml`

In `tut7.ml`, first we'll write functions to extract qualifiers from types. Then, we'll perform our additional type-checking.

7.1.1 Qualifier Types

We'll define some OCaml types representing the C type qualifiers. Then, from C types, we'll extract a possibly empty list of the qualifiers.

```
type color = Red | Blue | Green
```

We'll set up some global constants for the string representation of the qualifiers, and use them everywhere instead of the strings, in case we want to change them later on.

```
let redStr = "red"
let blueStr = "blue"
let greenStr = "green"
```

Putting the strings in a list will help a bit later on.

```
let color_strings = [redStr; blueStr; greenStr;]
```

As mentioned in a previous chapter, it is useful to have functions that convert a type to and from a string. The function `string_of_color` returns `redStr`, `blueStr`, or `greenStr` as appropriate.

```
let string_of_color (c : color) : string =
  match c with
  | Red → redStr
  | Blue → blueStr
  | Green → greenStr
```

The function `color_of_string` returns the `color` corresponding to the string `cs` it gets as input.

```
let color_of_string (cs : string) : color =
  match S.lowercase cs with
  | s when s = redStr → Red
  | s when s = blueStr → Blue
  | s when s = greenStr → Green
  | _ → E.s(E.bug "Expected a color string, got: %s" cs)
```

The function `isColorType` returns true when a type is qualified by a particular color attribute. The function `isTypeColor` the same with the order of the arguments reversed. The next three functions tell if a type is qualified by a particular color. These functions to largely similar things, but they'll be useful in different situations.

`isColorType` is written with the function `hasAttribute`, which comes from the core `Cil` module. It returns `true` when a list of attributes contains an attribute with the given name. `typeAttrs` extracts the attributes from a type. Attributes are one of the extensions to C accepted by `gcc`. CIL parses these attributes, even custom attributes not understood by `gcc`, and includes them in its AST attached to types, fields, functions, formal parameters, and blocks of code. All we care about for now, though, is whether or not a type is qualified by one of the colors.

```
let isColorType (cs : string) (t : typ) : bool =
  hasAttribute cs (typeAttrs t)
let isTypeColor (t : typ) (cs : string) : bool = isColorType cs t
let isRedType : typ → bool = isColorType redStr
let isBlueType : typ → bool = isColorType blueStr
let isGreenType : typ → bool = isColorType greenStr
```

The function `colors_of_type` takes a type and returns a list of colors that qualify the type.

```
let colors_of_type (t : typ) : color list =
  color_strings
  |> L.filter (isTypeColor t)
  |> L.map color_of_string
```

7.1.2 Type-checking

Now that we can extract the qualifiers we care about from a type, we can check that two types are compatible. First we define a type codifying the results of comparing two types, `typecheck_result`. Then, we'll write a function that performs the comparison, `checkColorTypes`. `TypesMismatch` and `ColorsMismatch` are parameterized by the types that didn't match. This is to enable writing better error and warning messages.

```
type typecheck_result =
  | TypesOkay
  | TypesMismatch of typ × typ
  | ColorsMismatch of typ × typ
```

The function `colorTypesCompat` first ensures that two types have the same colors before descending recursively into the structure of a type. That is, we check that pointer types have the same colors before checking that the pointed-to types also have the same colors. We'll leave as exercises the correct handling of function types, and also make some suggestions about how to modify this code to make it more general-purpose.

```
let rec colorTypesCompat (t1 : typ) (t2 : typ) : typecheck_result =
  let cl1 = colors_of_type t1 in
  let cl2 = colors_of_type t2 in
  if cl1 ≠ cl2 then ColorsMismatch(t1, t2) else begin
    match t1, t2 with
    | TVoid _, TVoid _ → TypesOkay
    | TPtr(t1, _), TPtr(t2, _)
    | TArray(t1, _, _), TArray(t2, _, _) → colorTypesCompat t1 t2
    | TFun _, TFun _ → TypesOkay (* See the exercise below *)
        (* ... *)
    | _, _ → TypesMismatch(t1, t2)
  end
```

The function `warning_for_tcres` generates warning messages for the various kinds of `typecheck_results`. In certain situations, some mismatches may be okay. To handle that, one would extend this function with a second parameter that indicates which mismatches are okay in a particular situation.

```
let warning_for_tcres (tcr : typecheck_result) : unit =
  match tcr with
  | TypesMismatch(t1, t2) →
    E.warn "%a: type mismatch: %a <> %a" d_loc (!currentLoc) d_type t1 d_type t2
  | ColorsMismatch(t1, t2) →
    E.warn "%a: color mismatch: %a <> %a" d_loc (!currentLoc) d_type t1 d_type t2
  | TypesOkay → ()
```

Now, we'll visit the AST looking for places where we must check type compatibility. We must check compatibility at assignments, casts, and parameter passing. In the case for casts, we allow constants to be cast to a qualified type. Without this exception, it would be impossible to initialize variables with qualified types without an error or warning. The case for function calls is left as an exercise.

```
class colorCheckVisitor = object(self)
  inherit nopCilVisitor
  method vinst (i : instr) =
    match i with
    | Set(lv, e, loc) →
      let tcres = colorTypesCompat (typeOfLval lv) (typeOf e) in
      warning_for_tcres tcres;
      DoChildren
    | Call(rlvo, fe, args, loc) → DoChildren                    (* See exercise *)
    | _ → DoChildren
  method vexpr (e : exp) =
    match e with
    | CastE(t, e) when ¬(isConstant e) →
      let tcres = colorTypesCompat t (typeOf e) in
      warning_for_tcres tcres;
      DoChildren
    | _ → DoChildren
end
```

The function `checkColorTypes` invokes the visitor `colorCheckVisitor` on a function.

```
let checkColorTypes (fd : fundec) (loc : location) : unit =
  let vis = new colorCheckVisitor in
  ignore(visitCilFunction vis fd)
```

Since `gcc` doesn't understand the color type attributes, we must use the `colorEraserVisitor` to remove them from the program before passing the code on to it. We do this by overriding the `vattr` method of the `nopCilVisitor` and filtering out the color attributes.

```
class colorEraserVisitor = object(self)
  inherit nopCilVisitor
  method vattr (a : attribute) =
    match a with
    | Attr(s, _) when L.mem s color_strings → ChangeTo []
    | _ → DoChildren
end
```

The function `eraseColors` invokes the visitor `colorEraserVisitor` on a file.

```
let eraseColors (f : file) : unit =
  let vis = new colorEraserVisitor in
  visitCilFile vis f
```

The `tut7` function is the entry point for this module. It checks the color types in all functions.

```
let tut7 (f : file) : unit =
  iterGlobals f (onlyFunctions checkColorTypes);
        eraseColors f
```

7.2 test/tut7.c

In this test, we declare a global `blue` integer `b`, and a local `green` integer `g`. We initialize `g` using the `AddColor` macro, which is defined in `ciltut.h`. It simply casts the constant in the second argument to the given color. Then, we attempt to assign `g` to `b`, which should elicit a warning from the compiler.

../test/tut7.c

```
#include <ciltut.h>

int blue b;

int main()
{
  int green g = AddColor(green, 5);
  b = g;
  return 0;
}
```

Now, when we attempt to compile this test, we get a warning:

```
$ ciltutcc -enable-tut7 -o tut7 test/tut7.c
Warning: test/tut7.c:16: color mismatch: int __attribute__((__blue__)) <> int
__attribute__((__green__))
```

Which is what we expected.

7.3 Exercises

1. Modify `typecheck_result` and `colorTypesCompat` to give more information when types do not match. For example, instead of returning `TypesMismatch`, `colorTypesCompat` might return `PtrIntMismatch` when `t1` is a `TPtr` and `t2` is a `TInt` (but the colors still match).

2. Corret the rule for function types in `colorTypesCompat`. Possibly add (a) new constructor(s) to `typecheck_result` for the case when function types do not match.

3. In combination with the above two exercises. Generalize the type checking code in `colorTypesCompat`. Instead of checking the color qualifiers, however, `colorTypesCompat` would accept a function argument for deciding whether the set of type attributes on `t1` and `t2` are compatible.

4. Write the `Call` case in `colorCheckVisitor#vinst`. Extract the type of the function from `fe`. Check the return type of the function against the destination of the return value (`rlvo`). Check the types of the actual arguments (`args`) against the types of the formal parameters. Note that there might be more actuals than formals if it is a variable argument function!

5. The `__attribute__` syntax in the warning message above could be cleaned up by inheriting from `Cil`'s `defaultCilPrinterClass` and overriding the methods for printing attributes.

7.4 Further Reading

Researchers have added flow-sensitive [3], and insensitive type-qualifiers [2], type-qualifier inference, and type-qualifier polymorphism [1] to languages such as C and Java [4].

In particular, the CCured [5] tool used flow-insensitive type-qualifier inference to determine the kind of fat pointer needed to check the correctness of pointer arithmetic in C, among other purposes.

References

[1] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, PLDI '99, pages 192–203, New York, NY, USA, 1999. ACM.

[2] Jeffrey S. Foster, Robert Johnson, John Kodumal, and Alex Aiken. Flow-insensitive type qualifiers. *ACM Trans. Program. Lang. Syst.*, 28(6):1035–1087, November 2006.

[3] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, pages 1–12, New York, NY, USA, 2002. ACM.

[4] David Greenfieldboyce and Jeffrey S. Foster. Type qualifier inference for java. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 321–336, New York, NY, USA, 2007. ACM.

[5] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, May 2005.

Chapter 8

Dependant Type Qualifiers

Of course, there are more colors than just red, green, and blue. Furthermore, the color of a type might depend on some other data in the program. In this example we introduce type qualifiers parameterized by arbitrary C expressions. Since types may depend on runtime data, type-checking must happen partially at runtime, which we will accomplish by instrumenting the code with the appropriate checks. Furthermore, the information we know about the color of a type may be imprecise, or rather, for example, a function may accept a range of colors rather than only one exact color. We introduce type qualifiers for expressing this.

First, we'll define OCaml types for representing the type qualifiers. Then, we'll define functions for extracting these qualifiers from C type attributes. This will involve keeping a typing context that maps strings to expressions so that the identifiers in type attributes may be translated into CIL variables. With the ability to extract colors from types, we will then be able to perform type checking as in the previous example. However, instead of generating a compile-time answer about correct typing, we must generate a list of instructions that test type compatibility at runtime.

It will be possible to determine at compile time that some of these checks will always succeed. Writing an optimization pass to remove these checks is left as an exercise.

OCaml Style Note: It would be better style to put each of these parts in separate OCaml modules: one for defining OCaml types; one for compiling the color qualifiers in type attributes into CIL expressions; one for setting up the runtime checks; and one containing the visitors used for type checking.

8.1 tut8.ml

The module SM is constructed using the functorial interface to the OCaml standard library Map module. With string as the key type, we'll use it as the context when translating attribute parameters to expressions.

```
module SM = Map.Make(struct
  type t = string
  let compare = Pervasives.compare
end)
```

8.1.1 Types and printing

Here, we define a type to represent the color type qualifier. There are three constructors. One identifies an exact color, and the other two represent upper and lower bounds on the color of a type. A type can have either one exact color, or at most one upper and one lower bound.

```
type rgb = exp × exp × exp
```

A color qualifier can be an exact color (ExactRGB), a lower bound (LowerRGB), or an upper bound (UpperRGB). The qualifiers are parameterized by three CIL expressions representing the amounts of red, blue, green, and blue that are a part of each color.

```
type color =
  | ExactRGB of rgb
  | LowerRGB of rgb
  | UpperRGB of rgb
type colors = color list
```

As in Chapter 7 it is useful to define global variables for string constants, as well as a list of the strings.

OCaml Style Note: It may be good to think of having a personal upper limit for the arity of tuple types. After some point, remembering the meanings of the elements becomes difficult, and it becomes more readable to define record types. The tuple rgb is a triple of three expressions: the amounts of red, green, and blue—hopefully easy enough to remember, so we don't make a record type for it.

```
let exactRGBStr = "ExactRGB"
let lowerRGBStr = "LowerRGB"
let upperRGBStr = "UpperRGB"
let color_strings = [exactRGBStr; lowerRGBStr; upperRGBStr;]
```

The next four functions, leading up to `string_of_colors` assist in turning a list of `color`s into a string. The function `rgb_of_color` returns the `rgb` tuple used in a `color`.

```
let rgb_of_color (c : color) : rgb =
  match c with
  | ExactRGB(r,g,b)
  | LowerRGB(r,g,b)
  | UpperRGB(r,g,b) → r, g, b
```

The function `string_of_rgb` uses `triplemap` (defined in `Tututil`) to convert the elements of an `rgb` tuple to `Pretty`.`docs`, and then to `string`s with `Pretty`.`sprint`.

```
let string_of_rgb (t : rgb) : string =
  let t =
    t |> triplemap (d_exp ())
      |> triplemap (sprint ~width : 80)
  in
  "("^(fst3 t)^", "^(snd3 t)^", "^(thd3 t)^")"
```

The function `string_of_color` uses `string_of_rgb` to convert a `color` to a `string`

```
let string_of_color (c : color) : string =
  let k =
    match c with
    | ExactRGB _ → exactRGBStr
    | LowerRGB _ → lowerRGBStr
    | UpperRGB _ → upperRGBStr
  in
  k^(c |> rgb_of_color |> string_of_rgb)
```

The function `string_of_colors` converts a list of `color`s into a comma separated string.

```
let string_of_colors (c : colors) : string =
  c
  |> L.map string_of_color
  |> S.concat ", "
```

8.1.2 Context for compiling type qualifiers

We would like to allow our color type qualifiers to refer to C expressions. Therefore, in order to interpret them, and to compile them into our OCaml type (`color`), we require a context to translate names into CIL expressions. Instances of the `ctxt` type simply map `string`s to CIL expressions. Here, we'll also write some functions for looking up strings in the context, adding variables, expressions, and fields to a context, and extending a context so that type qualifiers may refer to global varialbes, local variables, and structure fields. We also include a function so that the types of formal parameters may refer to other formal parameters.

```
type ctxt = exp SM.t
let exp_of_string (c : ctxt) (s : string) : exp = SM.find s c
let ctxt_add_var (c : ctxt) (vi : varinfo) : ctxt = SM.add vi.vname (v2e vi) c
let ctxt_add_exp (c : ctxt) (s : string) (e : exp) : ctxt = SM.add s e c
let ctxt_add_field (blv : lval) (c : ctxt) (fi : fieldinfo) : ctxt =
  SM.add fi.fname (Lval(addOffsetLval (Field(fi,NoOffset)) blv)) c
```

The function `context_for_globals` generates a fresh context containing mappings for the global variables. It does this by folding over the list of globals in `f.globals` and using `ctxt_add_var` when encountering a global variable declaration or definition.

```
let context_for_globals (f : file) : ctxt =
  L.fold_left (fun c g →
    match g with
    | GVarDecl(vi, _) → ctxt_add_var c vi
    | GVar(vi, _, _) → ctxt_add_var c vi
    | _ → c)
  SM.empty f.globals
```

The function `context_for_locals` extends a context with mappings for local variables and function parameters.

```
let context_for_locals (c : ctxt) (fd : fundec) : ctxt =
  L.fold_left ctxt_add_var c (fd.slocals @ fd.sformals)
```

Used at a call site, the function `context_for_call` extends a context with mappings from formal parameters to actual parameters. Since there may be more actuals than formals, we use `list_take` defined in `Tututil` to limit the mapped actuals to the number of formals. Then, we map `ctxt_add_exp` over the lists. It is used for resolving function argument types at a call site, which may refer to the other function arguments.

```
let context_for_call (c : ctxt) (fe : exp) (args : exp list) : ctxt =
  match typeOf fe with
  | TFun(_, Some stal, _, _) →
    let formals = L.map fst3 stal in
    let actuals = list_take (L.length stal) args in
    L.fold_left2 ctxt_add_exp c formals actuals
  | _ → c
```

When compiling a field access, the function `context_for_struct` extends a context with mappings from field names to expressions accessing those fields through the given `lval`. It is used for compiling the type of a structure field, which may refer to other fields of the same structure.

```
let context_for_struct (c : ctxt) (loc : location) (lv : lval) : ctxt =
  let blv, off = removeOffsetLval lv in
  match off with
  | NoOffset | Index _ → c
  | Field(fi, NoOffset) → L.fold_left (ctxt_add_field blv) c fi.fcomp.cfields
  | _ → E.s(E.bug "%a: Expected field w/o offset: %a" d_loc loc d_lval lv)
```

8.1.3 Compiling type qualifiers

Now that we can build a context for compiling the type qualifiers, we can go ahead with the actual compilation. In particular, we need to translate a CIL attribute parameter into an CIL expression.

Once we have compiled the type qualifiers, we'll be able to enforce rules about what sorts of color combinations can be on the same type. In particular, a type can have either no color, an exact color, or at most one lower bound and one upper bound.

Below, the function `exp_of_ap` translates `attrparam`s into CIL `exp`s in a given context. If the `attrparam` uses a name not mapped in the context, we print an error and stop compilation.

```
let rec exp_of_ap (c : ctxt) (loc : location) (ap : attrparam) : exp =
  let eoap = exp_of_ap c loc in
  match ap with
  | AInt i → integer i
  | AStr s → mkString s
  | ACons(s, []) → begin
    try exp_of_string c s
    with Not_found →
      E.s (E.error "%a: %s not in context for %a"
             d_loc loc s d_attrparam ap)
  end
  | AUnOp(uop,ap) →
    let e = eoap ap in
```

```
    UnOp(uop, e, typeOf e)
    (* And so forth.... *)
  | _ →
    E.s (E.error "%a: exp_of_ap: Attribute parameter is not an expression: %a"
      d_loc loc d_attrparam ap)
```

The function `make_colorqual` creates the different varieties of `color`s depending on the string `k`.

```
let make_colorqual (k : string) (loc : location) (et : rgb) : color =
  match k with
  | s when s = exactRGBStr → ExactRGB et
  | s when s = lowerRGBStr → LowerRGB et
  | s when s = upperRGBStr → UpperRGB et
  | _ → E.s (E.bug "%a: Expected an RGBStr got %s" d_loc loc k)
```

The function `colorqual_of_type` uses `exp_of_ap` and `make_colorqual` to compile the three components of the C attributes on a type. It uses the `Cil` library function `filterAttributes` to select the color attribute, compiles the components of the attribute (`rap`, `gap`, `bap`), and then calls `make_colorqual` to assemble the right `color`. If there is a syntax error in the attribute, we ignore the attribute and issue a warning.

```
let colorqual_of_type (k : string) (c : ctxt) (loc : location) (t : typ)
                      : colors
  =
  match filterAttributes k (typeAttrs t) with
  | (Attr(_,[rap; gap; bap])) :: rst →
    if rst ≠ [] then
      E.warn "%a: Type with multiple %s qualifiers. Keeping only the first: %a"
        d_loc loc k d_type t;
    [(rap, gap, bap)
     |> triplemap (exp_of_ap c loc)
     |> make_colorqual k loc]
  | (Attr _) :: _ →
    E.warn "%a: Malformed color attribute: %a"
      d_loc loc d_type t;
    []
  | _ → []
```

The three functions `extractRGB_of_type`, `lowerRGB_or_type`, and `upperRGB_of_type` use `colorqual_of_type` to extrac the given color qualifier.

```
let exactRGB_of_type = colorqual_of_type exactRGBStr
let lowerRGB_of_type = colorqual_of_type lowerRGBStr
let upperRGB_of_type = colorqual_of_type upperRGBStr
```

The function `colors_of_type` is our entry point for the type checking phase. Given a context, it returns the colors for a type after enforcing the requirements for a well-formed type mentioned above.

```
let colors_of_type (c : ctxt) (loc : location) (t : typ) : colors =
  let exact = exactRGB_of_type c loc t in
  let lower = lowerRGB_of_type c loc t in
  let upper = upperRGB_of_type c loc t in
  match exact, lower, upper with
  | e, [], [] → e
  | [], l, u → l @ u
  | _ →
    E.error ("%a: At most one exact, or one upper"^^
             "and one lower bound allowed: %a")
      d_loc (!currentLoc) d_type t;
    []
```

8.1.4 Runtime type-checking functions

Now that we can pull colors out of a type, we need to make available the functions we'll use to do the runtime component of the type-checking. This code is very similar to the code in Chapter 4. The important functions are `mkColorEqInst`, and `mkColorLeInst`, which generate instructions that check color equality and inclusion.

```
let mkColorEqInst () = mkColorInst color_funcs.color_eq
let mkColorLeInst () = mkColorInst color_funcs.color_le
```

8.1.5 Comparing colors

With colors extracted from types, and with functions for comparing the runtime values in the colors, we can now generate the necessary calls for for checking that two types are compatible.

The function `color_includes` checks whether one color is "included" in another. An `ExactRGB` defines a particular point, whereas `LowerRGB` and `UpperRGB` define ranges. So, we can just check inclusions for points and ranges.

```
let color_includes (loc : location)
                   (is_this : color) (in_this : color)
                   : instr list
  =
  match is_this, in_this with
  | ExactRGB c1, ExactRGB c2 → [mkColorEqInst () loc c1 c2]
  | ExactRGB c1, LowerRGB c2
  | LowerRGB c1, LowerRGB c2 → [mkColorLeInst () loc c2 c1]
  | ExactRGB c1, UpperRGB c2
  | UpperRGB c1, UpperRGB c2 → [mkColorLeInst () loc c1 c2]
  | _ → E.error "%a: color inclusion test will always fail" d_loc (!currentLoc);
        []
```

The function `colors_includes` checks that every `color` of `is_this` is included in every color of `in_this`. Notice how the nested `List.map` functions perform this all-pairs check, and builds the resulting list of instructions with `List.concat`

```
let colors_includes (loc : location)
                    (is_this : colors) (in_this : colors)
                    : instr list
  =
  if (is_this = [] ∧ in_this ≠ []) ∨ (is_this ≠ [] ∧ in_this = []) then
    (* Either both have colors or neither do *)
    E.error "%a: color mismatch" d_loc loc;
  L.concat (
    L.map (fun c1 →
      L.concat(L.map (color_includes loc c1) in_this)
    ) is_this
  )
```

8.1.6 Type-checking

Now, we can write the visitor that performs type-checking. First, we define two functions that extract colors from `lval`s and `exp`s using `colors_of_type`. The function `colors_of_lval` adds mappings to the context using `context_for_struct` in the case that `lv` accesses a structure field. `colors_of_exp` first checks whether the expression is for an `lval`, and if so uses `colors_of_lval` function. Otherwise it simply extracts the type of the expression and uses `colors_of_type`.

```
let colors_of_lval (c : ctxt) (loc : location) (lv : lval) : colors =
  colors_of_type (context_for_struct c loc lv) loc (typeOfLval lv)
```

```
let colors_of_exp (c : ctxt) (loc : location) (e : exp) : colors =
  match e with
  | Lval lv → colors_of_lval c loc lv
  | _ → colors_of_type c loc (typeOf e)
```

`colorCheckVisitor` visits a function with a context `c` that contains mappings for the local variables and formal parameters. At assignments and function calls, it inserts function calls that the type of the destination lvalue or formal parameter includes the type of the right-hand-side or actual parameter.

These instructions are inserted using `self#queueInstr`. This is yet another mechanism with which one can modify the AST. `self#queueInstr` can be called from any visitor method. Instructions queued up with `self#queueInstr` are dequeued (i.e. inserted) at the latest possible point before the AST node currently being visited. So, if we queue up an instruction while visiting an instruction, the new instruction will be inserted immediately before the one currently being visited.

The case for checking function calls is ommitted because of its length, but it is a very useful pattern, and might help with one of the exercises from a previous chapter.

```
class colorCheckVisitor (c : ctxt) = object(self)
  inherit nopCilVisitor
  method vinst (i : instr) =
    match i with
    | Set(lv, e, loc) →
      let lvc = colors_of_lval c loc lv in
      let ec = colors_of_exp c loc e in
      self#queueInstr (colors_includes loc ec lvc);
      DoChildren
        | _ → DoChildren
end
```

The function `checkColorTypes` invokes the visitor `colorCHeckVisitor` on function `fd` using context `c`.

```
let checkColorTypes (c : ctxt) (fd : fundec) (loc : location) : unit =
  let c = context_for_locals c fd in
  let vis = new colorCheckVisitor c in
  ignore(visitCilFunction vis fd)
```

The function `tut8` is the entry point to the code in this tutorial. It first initializes our dynamic type-checking functions (`tut8_init`), then builds a global context for compiling type attributes (`context_for_globals`), performs the type-check, and finally erases the color attributes.

```
let tut8 (f : file) : unit =
  tut8_init f;
  let c = context_for_globals f in
  c |> checkColorTypes |> onlyFunctions |> iterGlobals f;
  eraseColors f
```

8.2 test/tut8.c

In this test we define a structure type in which the type of one of the fields depends on the values of the other fields, and two functions in which the type of one of the formal parameters depends on the values to be bound to the other formal parameters at each call site. The test demonstrates the type safe initialization of the fields of the structure (i.e. fields whose type depends on the other fields are initialized last.) Also, we check that the compiler catches a call with poorly-typed parameters (i.e. the call to `bar`).

../test/tut8.c

```
# include <ciltut.h>

struct bar {
  int r, g, b;
  int ExactRGB(r,g,b) c;
};

void foo(int LowerRGB(r,g,b) c, int r, int g, int b) {return;}
void bar(int UpperRGB(r,g,b) c, int r, int g, int b) {return;}

int main()
{
  struct bar B;

  B.r = 50;
  B.g = 50;
  B.b = 50;
  B.c = AddRGB(50,50,50,50);

  foo(B.c, B.r, B.g, B.b);
  bar(B.c, 10, 10, 10);

  return 0;
}
```

We compile this example:

```
$ ciltutcc --enable-tut8 -o tut8 test/tut8.c
```

which completes successfully, but when we run it:

```
$ ./tut8
test/tut8.c:40 Bad color coercion:  (50,50,50) > (10,10,10)
```

a runtime type-check assertion fails at the call to `bar`, as expected.

8.3 Exercises

1. The visitor does not check casts, though this is another obvious place to check color compatibility. Add code to the visitor for checking casts, but change the static and runtime type-checks so that only a warning is emitted if colors for a cast are not compatible. (If the programmer is adding an explicit cast, maybe they know what they're doing.)

2. In Chapter 7 we recursively descended into types, checking color compatibility at all levels. Extend this code to do similar checking.

3. When a variable is the target of an assignment, then all types that reference it are changed. The new types must be included in the corresponding old types. In order to capture all cases in which an assignment may change a type, we must place restrictions on the expressions that may appear in types. What are these restrictions? Modify this code to enforce those restrictions, and to place appropriate runtime checks when an assignment may change a type (i.e. inclusion checks for the types pre- and post-assignment).

4. Create an optimization visitor or dataflow analysis that removes checks that we can be sure will always succeed.

8.4 Further Reading

Deputy [2] is an extension to C's type-system that adds dependent type-qualifiers that track the length of memory buffers. In performing type-checking, the compiler adds runtime checks to the program whenever pointer arithmetic is performed (or any variable mentioned in a type-qualifier may be modified), since doing so may alter a type.

Shoal [1] is also an extension to C's type-system that adds dependent type-qualifiers that allow the compiler to track membership of an object in a pointer-based data-structure. In particular, the type of an external pointer into a data-structure may be qualified by a pointer to a distinguished member of the data-structure (i.e. the root of a tree). This has the result that the number of external pointers into a data-structure can be tracked simply by counting the number of pointers qualified by pointers to the distinguished member. In Shoal this is used to check the safety of converting pointer data-structures from thread-private to thread-shared and back again.

References

[1] Zachary Anderson, David Gay, and Mayur Naik. Lightweight annotations for controlling sharing in concurrent data structures. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 98–109, New York, NY, USA, 2009. ACM.

[2] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George Necula. Dependent types for low-level programming. In *ESOP'07*.

Chapter 9

Type Qualifier Inference

In the previous two sections, we saw how to add type-qualifiers to the language, and how to do any extra type-checking that they might require. In this section, we'll see how one can begin to do type-qualifier inference. This might be useful, for example, if we wish to make our changes to C more unobtrusive. That is, we would like to add expressiveness to the language and type-system without thrusting a large annotation burden onto the programmer. One way to accomplish this is by inferring as many type qualifiers as possible. Additionally, analyses like this one, which push type qualifiers around, can be used for purposes aside from type-inference. For example, type qualifier inference can be used to perform an imprecise (though still very useful) thread-escape analysis.

9.1 `tut9.ml`

We'll use the following method to infer type qualifiers. First, we'll place a numeric IDs on each type node in the AST. Then we'll build a graph with these numeric IDs on each node and directed edges given by the relationships between types that we find in the program. Next, we iterate over the nodes until we reach a fix-point that satisfies the relationships encoded by the graph edges. Following that, we add the inferred type qualifiers to the program. If an inferred type qualifier doesn't match the one that the programmer has given explicitly, we emit a compile-time warning. This inference pass is intended to be followed by a type-checking phase as in one of the previous sections. Therefore, it is important that we don't infer anything that won't type-check!
We'll need the standard library `Queue` module, and a few things from Chapter 7, in particular the definition of the `color` type.

```
module Q = Queue
module T = Tut7
type colors = T.color list
```

Even though we are borrowing some OCaml types and functions from `Tut7`, the C type system for this analysis is slightly different. In `Tut7` for two types to be compatible, their colors had to match exactly. In this tutorial, we use the same colors, with a different restriction: type A is {included

in, a subtype of, etc...} type B when the colors of type A are a subset of the colors of type B. For example, a `Red int` is also a `Red Blue int`. Thus, the subset lattice of the colors defines the subtype relation of our type system, and we can use it to find the least-upper-bound of two types, which will be important for the inference algorithm below.

9.1.1 Mark types with node IDs

First we'll introduce two visitors and two functions that run the visitors over files. The first visitor marks each type node in the AST with a unique numeric ID held in an attribute. The second visitor removes these marks, which we must do before we pass the code on to `gcc`. The function `nodeAttr` creates an attribute called `"Node"` parameterized by an integer node ID.

```
let nodeStr = "Node"
let nodeAttr (id : int) : attributes = [Attr(nodeStr, [AInt id])]
```

The function `node_of_type` returns the integer node ID of a type. Since, we are adding the `"Node"` attributes ourselves, it is a bug if one is malformed. Hence, we use `E.bug`. If there is no `"Node"` attribute, this implies that `t` is the result of calling `typeOf` on a constant. We group all of these types at node 0 in the graph. Later, we'll see that this node is unreachable, and so we won't try to infer a type qualifier for it.

```
let node_of_type (t : typ) : int =
  match filterAttributes nodeStr (typeAttrs t) with
  | [(Attr(_, [AInt id]))] → id
  | [] → 0
  | _ → E.s (E.bug "%a: Malformed node id on %a" d_loc (!currentLoc) d_type t)
```

The visitor `typeNodeMarker` visits every `typ` node in the AST and assigns them unique node IDs by placing a `"Node"` attribute on them.

```
class typeNodeMarker (node_count : int ref) = object(self)
  inherit nopCilVisitor
  method vtype (t : typ) =
    let action t =
      if ¬(hasAttribute nodeStr (typeAttrs t)) then begin
        let attr = nodeAttr (!node_count) in
        incr node_count;
        typeAddAttributes attr t
      end else t
    in
    ChangeDoChildrenPost(t, action)
end
```

The function `addNodeMarks` invokes the `typeNodeMarker` visitor on a file and returns the largest ID assigned by the visitor. This will be the number of nodes we need in our graph.

```
let addNodeMarks (f : file) : int =
  let cntr = ref 1 in
  let vis = new typeNodeMarker cntr in
  visitCilFile vis f;
  !cntr
```

9.1.2 Build a graph

We'll use an adjacency list representation for the graph. We define a `node` type, which is just two lists, one with edges to us (`incoming`), and the other with edges from us (`outgoing`). Later, we'll add colors to the graph nodes.

```
type node = {
  mutable ncolors : colors;
  mutable incoming : int list;
  mutable outgoing : int list;
}
```

The function `newNode` creates a new node with no edges incoming or outgoing.

```
let newNode (id : int) : node =
  {ncolors = []; incoming = []; outgoing = []}
```

A graph is just an array of nodes. We can use an array because the number of nodes is fixed, and we can get that number from the `addNodeMarks` function above.

```
type graph = node array
```

The function `graphCreate` creates an array of `node`s with no edges.

```
let graphCreate (n : int) : graph = A.init n newNode
```

The function `graphAddEdge` adds a directed edge to the graph.

```
let graphAddEdge (g : graph) (from_node : int) (to_node : int) : unit =
  if ¬(L.mem to_node g.(from_node).outgoing) then
    g.(from_node).outgoing ← to_node :: g.(from_node).outgoing;
  if ¬(L.mem from_node g.(to_node).incoming) then
    g.(to_node).incoming ← from_node :: g.(to_node).incoming
```

OCaml Pitfall: In `graphCreate` you might be tempted to say: `A.make n {ncolors = []; ...}`. However this would be problematic. The array would be initialized, not with *copies* of the given `node` record, but rather with *references* to a *single* `node` because the record literal would be evaluated before being passed to `A.make`. On the other hand, the `newNode` function is called for each array element, allocating a fresh `node` for each one.

The function `typesAddEdge` places an edge in the graph from the node for type `from_type` to the node for type `to_type`.

```
let rec typesAddEdge (g : graph) (from_type : typ) (to_type : typ) : unit =
  let from_id = node_of_type from_type in
  let to_id = node_of_type to_type in
  graphAddEdge g from_id to_id
```

Before defining the visitor that will build the graph, we define a couple of functions that will make handling function calls a bit easier. The function `addEdgesForCallArgs` adds edges arising from the assignment of actuals to formals at function call sites.

```
let addEdgesForCallArgs (g : graph) (fe : exp) (aes : exp list) : unit =
  let fts = fe |> function_elements |> snd |> L.map snd3 in
  let ats = aes |> list_take (L.length fts) |> L.map typeOf in
  L.iter2 (typesAddEdge g) ats fts
```

The function `addEdgesForCallRet` adds edges for the assignment of return values at call sites.

```
let addEdgesForCallRet (g : graph) (fe : exp) (rlvo : lval option) : unit =
  match rlvo with
  | None → ()
  | Some rlv →
    let rt, _ = function_elements fe in
    typesAddEdge g rt (typeOfLval rlv)
```

The visitor `graphBuilder` adds edges to the graph. Edges encode constraints on type qualifiers. We place an edge from node A to node B when the type of A must be included in the type of B. Thus, if we have an assignment `lv = e` the type of `lv` has to be "big enough" to handle the type of `e`. That is, the type of `e` must be included in the type of `lv`. Then, we can just think of casts, parameter passing, and the `return` statements as different ways of doing assignments, and add edges for them the same way.

```
class graphBuilder (g : graph) (fd : fundec) = object(self)
  inherit nopCilVisitor
  method vinst (i : instr) =
    match i with
    | Set(lv, e, loc) →
      typesAddEdge g (typeOf e) (typeOfLval lv);
      DoChildren
  (* ... *)
  end
```

The function `functionBuildGraph` invokes the visitor `graphBuilder` on function `fd`, building the graph in the mutable array `g`.

```
let functionBuildGraph (g : graph) (fd : fundec) (loc : location) : unit =
  let vis = new graphBuilder g fd in
  ignore(visitCilFunction vis fd)
```

The function `fileBuildGraph` simply calls the functions we defined above, first for marking the AST with `"Node"` type attributes, and second building the graph. The function returns the resulting graph.

```
let fileBuildGraph (f : file) : graph =
  let g = f |> addNodeMarks |> graphCreate in
  functionBuildGraph g
  |> onlyFunctions
  |> iterGlobals f;
  g
```

9.1.3 Inferene Algorithm

The inference algorithm must be seeded somehow. In this example, the `nodeColorFinder` visitor takes any type qualifiers that the programmer added explicitly to the program, and puts them into our graph. However, in a different application, depending on the meanings of the type qualifiers, it may be possible to deduce some of the missing type qualifiers before the inference stage begins.

```
class nodeColorFinder (g : graph) = object(self)
  inherit nopCilVisitor
  method vtype (t : typ) =
    let id = node_of_type t in
    let c = T.colors_of_type t in
    g.(id).ncolors ← c;
    DoChildren
end
```

The function `findColorNodes` invokes the `nodeColorFinder` visitor on the file `f` with the graph `g`.

```
let findColoredNodes (f : file) (g : graph) : unit =
  let vis = new nodeColorFinder g in
  visitCilFile vis f
```

The function `colors_equal` ensures that every color of `c1` is somewhere in `c2` and that every color of `c2` is somewhere in `c1`. That is it checks that the lists have the same elements.

```
let colors_equal (c1 : colors) (c2 : colors) : bool =
  L.for_all (fun c → L.mem c c2) c1 ∧
  L.for_all (fun c → L.mem c c1) c2
```

Now that the graph has been seeded by a few type qualifiers, we can see where else in the graph they must flow. We'll accomplish this by doing the following. First, we put all the nodes in the graph into a queue. Then, for each node that we pop off the queue, we'll find the least-upper-bound of the qualifier on the node itself along with the qualifiers on the nodes for incoming edges; these are the types that it must include. If the least-upper-bound qualifier is different, we'll assign the new qualifier to the node, and then add the nodes on outgoing edges to the end of the queue. When the qualifier lattice has finite height, as it does here, we can be sure that the algorithm will terminate.

The function `enqueueNodes` places each of the nodes in our graph on a queue and returns the queue.

```
let enqueueNodes (g : graph) : int Q.t =
  let q = Q.create () in
  A.iteri (fun i _ → Q.add i q) g;
  q
```

The function `processNode` folds over the incoming edges of a node, computing the least-upper-bound of the types on the origin nodes. If the result is different from the starting type, it adds the new type to the graph and return `true`. Otherwise it returns `false`.

```
let processNode (g : graph) (id : int) : bool =
  let c' =
    L.fold_left (fun c id' → list_union c g.(id').ncolors)
      g.(id).ncolors g.(id).incoming
  in
  if ¬(colors_equal g.(id).ncolors c') then begin
    g.(id).ncolors ← c';
    true
  end else false
```

The function `processQueue` applies `processNode` to each node of the graph `g` on queue `q`. If `processNode` for some node returns true, it adds the destinations of the node's outgoing edges to the end of the queue.

```
let processQueue (g : graph) (q : int Q.t) : unit =
  while ¬(Q.is_empty q) do
    let id = Q.pop q in
    if processNode g id then begin
      L.iter (fun id' → Q.add id' q) g.(id).outgoing
    end
  done
```

The function `attr_of_color` builds a new type attribute from the color `c`.

```
let attr_of_color (c : T.color) : attribute = Attr(T.string_of_color c, [])
```

Once the inference algorithm is finished, we can add its results to the types in the program. If an inferred type qualifier disagrees with one that the programmer has added explicitly, we keep the programmer's annotation, but issue a warning.

The visitor `colorAdder` visits `typ` nodes in the AST. It reads the inferred color type using the `typ`'s `"Node"` attribute, and adds it to the `typ`, respecting the above policy.

```
class colorAdder (g : graph) = object(self)
  inherit nopCilVisitor
  method vtype (t : typ) =
    let oc = T.colors_of_type t in
    let ic = (t |> node_of_type |> A.get g).ncolors in
    if oc ≠ [] ∧ ¬(colors_equal ic oc) then DoChildren
    else if list_equal (=) ic oc then DoChildren else
    let nattr = L.map attr_of_color ic in
    ChangeTo (typeAddAttributes nattr t)
end
```

The function `addInferredColors` invokes the visitor `colorAdder` on the file `f` using the inference results in graph `g`.

```
let addInferredColors (f : file) (g : graph) : unit =
  let vis = new colorAdder g in
  visitCilFile vis f
```

Finally, we tie everything together with the function `tut9`, which is the entry point into the module. We must also remember to remove the node attributes we added for building the graph. We leave the inferred colors since these will need to be checked by a type-checking pass like the one in Chapter 7.

```
let tut9 (f : file) : unit =
  let g = fileBuildGraph f in
  let q = enqueueNodes g in
  findColoredNodes f g;
  processQueue g q;
  addInferredColors f g;
  eraseNodeMarks f
```

9.2 test/tut9.c

This test demonstrates type qualifer inference. In particular, the types of the parameters `r` and `g` of function `foo` will be inferred.

../test/tut9.c

```
#include <ciltut.h>

struct bar {
  int red * red r;
  int green g;
  int red green blue c;
};

void foo(int blue c, int r, int g)
{
  return;
}

int main()
{
  struct bar B;
  int red r = 50;

  B.r = &r;
  B.g = 50;
  B.c = AddColor(blue, 50);

  foo(B.c, *B.r, B.g);
  foo(B.g, r, r);

  return 0;
}
```

When we build this test by doing:

```
$ ciltutcc --enable-tut9 --save-temps -o tut9 test/tut9.c
```

We recieve a number of warnings because we have not erased our custom type qualifiers from the source. We did this so that we could examine the results of type qualifier inference. In particular, since we passed `--save-temps` to `ciltutcc`, we can look at the results in `tut9.cil.c`.

When we do so, we can note the inferred types of the parameters `r` and `g` of `foo`:

```
void foo(int blue c, int red r, int red green g) ...
```

Since the type of parameter `c` was declared explicitly, its type qualifier is not inferred to be anything else. Since only `red int`s were passed as parameter `r`, it is inffered to be a `red int`. Finally, since both `red` and `green int`s were passed as parameter `g`, it is inferred to be both `red` and `green`.

9.3 Exercises

1. Modify `typesAddEdge` to recursively descend into types, adding edges as appropriate. Since this is just the inference stage, it is okay to ignore typing errors unless the problem is somehow catastrophic.

9.4 Further Reading

As mentioned in earlier chapters, Deputy and CCured use type-qualifier inference to infer how pointers are used, and thus what sort of safety checks their use requires.

Additionally, SharC [1] is an extension to C's type system and language runtime that enforces programmer written type-qualifiers describing how data may be shared among threads. As an optimization it uses a flow-insensitive type-qualifier inference algorithm as a thread-escape analysis. In particular, if the compiler can determine that an object is thread-private, then there is no need to check accesses to it at runtime for data-races. The inference algorithm is seeded by the types of locations that are obviously thread-escaping, like global variables accessed in thread functions, or arguments to `pthread_create()`. Then, the relevant type-qualifiers flow across assignments as in the above tutorial. At the end of the inference algorithm, those locations without the type qualifier indicating thread-escape must be thread-private.

References

[1] Zachary Anderson, David Gay, Rob Ennals, and Eric Brewer. Sharc: checking data sharing strategies for multithreaded c. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 149–158, New York, NY, USA, 2008. ACM.

Chapter 10

Adding a New Kind of Statement

In this section, we will take a look at two different ways of adding a new statement to C *without* any changes to the lexer or parser. In particular, we will add a statement to C that profiles the cache miss rate and used memory bandwidth in a block of code. Adding the new syntax and replacing the new statement with calls into our runtime library will be the easy part. The more difficult task will be the implementation of the runtime library. In particular, we must arrange for each thread to maintain sufficient state to handle arbitrary nesting of the new statement.

10.1 `tut10.ml`

The two different ways of adding syntax involve further use of the same attribute syntax that we used for type qualifiers. In the first method, we transform a specially constructed `if` statement. In the second method we transform `Block` statements that have been annotated with the `__blockattribute__((...))` syntax supported by CIL. When using the transformed `if` statement, we'll be able to use a C preprocessor macro to make it look more like a standard C statement.
First, we define a string for the name of the new statement. Then, in `hasCacheReportAttrs` and `isCacheReportType`, we define functions for determining whether a type or set of attributes contains the the name of the new statement.

```
let cacheReportStr = "cache_report"
let hasCacheReportAttrs : attributes → bool = hasAttribute cacheReportStr
let isCacheReportType (t : typ) : bool = t |> typeAttrs |> hasCacheReportAttrs
```

The function `isCacheReportState` looks for two kinds of statements: either an `if` statement testing the constant 0 cast to a type with the `cache_report` attribute, or a `Block` with that attribute.

```
let isCacheReportStmt (s : stmt) : block option =
  match s.skind with
  | If(CastE(t,z),b,_,_) when z = zero ∧ isCacheReportType t → Some b
  | Block b when hasCacheReportAttrs b.battrs → Some b
  | _ → None
```

Now, if we make the following macro:

```
#define cache_report if((void * __attribute__((cache_report)))0)
```

We can write code like the following:

```
cache_report {
   ...
}
```

Alternately (or in addition), if we make the following macro:

```
#define CACHE_REPORT __blockattribute__((cache_report))
```

We can write code like the following:

```
{ CACHE_REPORT
   ...
}
```

The above syntactic constructs will be replaced by calls to two functions, one at the beginning of the block of code we want to profile, and one at the end.

We create instrumentation functions just as we did back in Chapter 4, so we just include the code here that builds the function call instructions.
The functions are defined in `ciltut-lib/src/tut10.c` and set up using the function `initCacheFunctions`. The function `makeCacheReportStmts` builds the function calls for source location `loc`, and returns a pair of statements containing the instructions.

```
let makeCacheReportStmts (loc : location) : stmt × stmt =
  let f, l = mkString loc.file, integer loc.line in
  i2s (Call(None,v2e cachefuns.cache_begin, [f; l;], loc)),
  i2s (Call(None,v2e cachefuns.cache_end, [f; l;], loc))
```

The `cacheReportAdder` visitor finds statements satisfying `isCacheReportStmt`, and brackets the statements nested within them inside of the calls to our runtime, which are given by `makeCacheReportStmts`.

```
class cacheReportAdder = object(self)
  inherit nopCilVisitor
  method vstmt (s : stmt) =
    let action s =
      match isCacheReportStmt s with
      | Some b → begin
        let bs, es = makeCacheReportStmts (get_stmtLoc s.skind) in
        let nb = mkBlock [bs; mkStmt(Block b); es] in
        s.skind ← Block nb;
        s
      end
      | None → s
    in
    ChangeDoChildrenPost(s, action)
end
```

The function `tut10` is the entry point to this module. It Initializes the runtime functions, and invokes the `cacheReportAdder` visitor on the file `f`.

```
let tut10 (f : file) : unit =
  initCacheFunctions f;
  let vis = new cacheReportAdder in
  visitCilFile vis f
```

Now we describe all of the hard work that takes place inside of the C functions `tut_cache_begin` and `tut_cache_end`. In addition we must explain the data structures used to maintain per-thread state, and the method for reading the hardware performance counters for cache misses and references.

10.2 `tut10.c`

This C source file contains the implementation of the runtime for the `cache_report` statement. It must accomplish three things. First, each thread must use Linux's `perf_event_open` system call to access the hardware performance counters for cache misses and references. Secondly, each thread must maintain a stack that mirrors the nesting structure of the `cache_report` statement. Finally, we must use the dynamic linker to override the C Library function `pthread_create` to ensure that new threads are set up to access the performance counters and to maintain the stack.

../ciltut-lib/src/tut10.c

```
#define _GNU_SOURCE   // Needed for RTLD_NEXT
#include <stdint.h>   // for uint64_t
#include <pthread.h>  // for pthread_create
#include <dlfcn.h>    // for RTLD_NEXT
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <ciltut.h>
```

We'll omit from the text most of the code for setting up the performance counters. You can find it in `ciltut_libc.c`. The important thing to note about this code is that `perf_get_cache_refs()` returns the cumulative number of cache references performed by the calling thread, and `perf_get_cache_miss()` returns the cumulative number of those references that missed the cache.

For each thread we'll keep a stack of records that each record the cumulative number of cache misses (`start_miss`), the cumulative number of cache references (`start_refs`), and the starting time (`start_time`) at the beginning of a `cache_report` statement.

../ciltut-lib/src/tut10.c

```
struct cache_stack_entry {
  uint64_t start_miss;
  uint64_t start_refs;
  uint64_t start_time;
};

#define MAX_CACHE_STACK_ENTRIES 256
struct cache_stack {
  struct cache_stack_entry s[MAX_CACHE_STACK_ENTRIES];
  int t;
};
```

Next, we set up the thread local storage for the stack, and a pointer to the top of the stack. Linux supports gcc's `__thread` storage modifier, but other OS's might not, for example Mac OSX. Thus, we'll just stick to using `pthread_setspecific` and `pthread_getspecific` for implementing thread local storage. The `CONSTRUCTOR` attribute on `init_CS_key` ensures that the stack is initialized in the first thread of the program. Next, we'll ensure that each new thread also calls `init_CS_key`.

../ciltut-lib/src/tut10.c

```
pthread_key_t CS_key;
static void init_CS()
{
  struct cache_stack *CS = calloc(1, sizeof(*CS));
  pthread_setspecific(CS_key, CS);
}

CONSTRUCTOR static void init_CS_key()
{
  pthread_key_create(&CS_key, &free);
  init_CS();
}

static struct cache_stack *get_CS()
{
  return (struct cache_stack *)pthread_getspecific(CS_key);
}
```

The next three items demonstrate how to override the C Library's `pthread_create` function. First, we set up a function pointer to store a reference to the original function (`pthread_create_orig`)—we'll need to call it from inside of our version. Second, wrap the call to the dynamic linker in some error checking code in `checked_dlsym`. Finally, we set up a constructor function, which runs before `main`, to call `checked_dlsym`, which returns a pointer to the original `pthread_create` call.

../ciltut-lib/src/tut10.c

```
int (*pthread_create_orig)(pthread_t *__restrict,
                           __const pthread_attr_t *__restrict,
                           void *(*)(void *),
                           void *__restrict) = NULL;

extern void *checked_dlsym(void *handle, const char *sym);

CONSTRUCTOR static void init_cache_stack()
{
  pthread_create_orig = checked_dlsym(RTLD_NEXT, "pthread_create");
}
```

The goal of wrapping `pthread_create` is to ensure that spawned threads set up the state needed for the `cache_report` statement. Therefore, we need to wrap every function passed to `pthread_create` in a function that performs the initialization before calling passed function. We accomplish this by defining a structure type for the function pointer and its argument. Then, in `tfunc_wrapper`, we initialize thread local storage for the cache stack, and initialize the performance counters for the new thread with `perf_init` before calling the function pointer on its argument.

../ciltut-lib/src/tut10.c

```
struct pthread_closure {
  void *(*fn)(void *);
  void *arg;
};

static void *tfunc_wrapper(void *arg)
{
  struct pthread_closure *c = (struct pthread_closure *)arg;
  void *(*fn)(void *) = c->fn;
  void *a             = c->arg;
  void *res           = NULL;

  free(c);

  init_CS();
  perf_init(gettid());
  res = fn(a);
  perf_deinit();

  return res;
}
```

With the thread function wrapper set up in `tfunc_wrapper`, we can now set up our version of `pthread_create`, which allocates a closure for the thread routine and its argument before making the call to `pthread_create_orig`.

../ciltut-lib/src/tut10.c

```
int pthread_create(pthread_t *__restrict thread,
                   __const pthread_attr_t *__restrict attr,
                   void * (*start_routine)(void *),
                   void *__restrict arg)
{
  struct pthread_closure *c = malloc(sizeof(struct pthread_closure));
  int res;

  c->fn  = start_routine;
  c->arg = arg;

  res = pthread_create_orig(thread, attr, &tfunc_wrapper, c);
  if (res != 0) {
    printf("pthread failed\n");
    fflush(stdout);
    free(c);
  }

  return res;
}
```

Finally, we can write the functions, which we inserted into the code with our CIL transformation. In `tut_cache_begin` we read the performance counters for cache references and misses and the current time, and write these values into the top entry of the stack. In `tut_cache_end`, we read the performance counters and get the current time again, calculate the differences from the starting values, and output the cache miss rate as a percentage of the total number of references. Since we also know the elapsed time, we can also calculate the memory bandwidth.

../ciltut-lib/src/tut10.c

```
void tut_cache_begin(char const *f, int l)
{
  struct cache_stack *cs = get_CS();

  cs->s[cs->t].start_miss = perf_get_cache_miss();
  cs->s[cs->t].start_refs = perf_get_cache_refs();
  cs->s[cs->t].start_time = tut_get_time();
  cs->t++;

  return;
}
```

In `tut_cache_end`, we read the performance counters and get the current time again, calculate the differences from the starting values, and output the cache miss rate as a percentage of the total number of references. Since we also know the elapsed time, we can also calculate the memory bandwidth.

../ciltut-lib/src/tut10.c

```
void tut_cache_end(char const *f, int l)
{
  uint64_t final_miss, final_refs, final_time;
  uint64_t net_miss,   net_refs,   net_time;
  double miss_rate;
  double bandwidth;
  struct cache_stack *cs = get_CS();

  final_miss = perf_get_cache_miss();
  final_refs = perf_get_cache_refs();
  final_time = tut_get_time();

  net_miss = final_miss - cs->s[cs->t - 1].start_miss;
  net_refs = final_refs - cs->s[cs->t - 1].start_refs;
  net_time = final_time - cs->s[cs->t - 1].start_time;

  miss_rate = (double)net_miss/(double)net_refs;
  bandwidth = (double)(net_miss * 1000000000ULL)/(double)net_time;
  printf("%s:%d Miss rate was: %f, Bandwidth was %f\n",
         f, l, miss_rate, bandwidth);
  fflush(stdout);

  cs->t--;
  return;
}
```

The program in file `test/tut10.c` spawns as many threads as there are cores on the machine, and writes into a large array. The loop that writes the array is wrapped in a `cache_report` statement. We can build and run the test as follows:

```
$ ciltutcc --enable-tut10 -o tut10 test/tut10.c
$ ./tut10
...  # debug messages from the pthread_create wrapper
test/tut10.c:26 Miss rate was:  0.133511, Bandwidth was 1176936.786361
test/tut10.c:26 Miss rate was:  0.135422, Bandwidth was 1164002.872631
test/tut10.c:26 Miss rate was:  0.136743, Bandwidth was 1174543.676575
test/tut10.c:26 Miss rate was:  0.126649, Bandwidth was 1073938.382925
```

On my machine, there are four cores, and the cache line size of the last-level cache is 64-bytes. Thus, the reported miss rate of approximately 1/8 makes sense since we are going sequentially through the array accessing each of the 8-byte `uint64_t`s only once or twice.

10.3 Exercises

1. Given the example code in this chapter. It is straightforward to add statements that report on other performance counters, both hardware and software. Look in `linux/perf_event.h`, and add another statement that profiles a different performance counter.

2. We might also profile values that are specific to a particular application. Notice that, since we know how to translate attribute parameters into C expressions, we could just as well have allowed arguments to be passed to the profiling statement we added, e.g. `cache_report(e) {}`. Extend this statement to accept the address of an integer value, whose value is profiled instead of a performance counter.

3. In `ciltut-lib/src/ciltut_libc.c`, reading the performance counters is implemented only for Linux. Implement `perf_get_cache_miss` and `perf_get_cache_refs` for another OS.

10.4 Further Reading

This technique for adding a new statement is exactly the same as the one used by the Shelters [1] extension to C, which adds an `atomic {}` statement. The statement is replaced by calls into the the Shelters language runtime, which enforces mutual exclusion in multithreaded programs for selected objects touched inside of the statement.

References

[1] Zachary Anderson and David Gay. Composable, nestable, pessimistic atomic statements. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 865–884, New York, NY, USA, 2011. ACM.

Chapter 11

Program Verification

In this tutorial, we will use the Why3 [3] verification framework to prove things about C code. In particular we will generate verification conditions (VCs) from preconditions, postconditions, and loop invariants given by an annotation syntax that we will add to C using function type attributes and block attributes. Suppose a function f is annotated with precondition *pre* and postcondition *post*. We ask Why3 to prove the validity of $pre \rightarrow VC(f, post)$.

An introduction to the generation of verification conditions for imperative languages can be found in the textbook by Winskel [6]. In this example we will use the *backwards* method of VC generation. This will preclude a straightforward handling of C constructs like `goto`, `break`, and `continue` statements. For these, a *forwards* method of VC generation is more suitable. However, the backwards method is able to handle all the other features of C including `while` loops, `for` loops, `if` statements, and `switch` statements.

This module allows expression preconditions, postconditions, and loop invariants in C code with syntax extensions embodied in the following example:

```
void (pre(p1) post(p2) f)(...)  {
   while(c) { invariant(c, p3, v1, ..., vn)
   }
}
```

Here, `p1`, `p2`, and `p3` are propositions that may include universal quantifications and implications. Universal quantification is written as `forall(v1,...,vn,p)` where `v1` through `vn` are the variables being quantified over, and `p` is the formula being quantified over. Implication is written as `implies(p1,p2)`, where `p1` is the antecedent and `p2` is the consequent. In the loop invariant annotation, `c` is the loop termination condition, `p3` is the loop invariant proposition, and `v1` through textttvn are the loop variant variables. Except for `forall` and `implies`, C expression syntax is used for the propositions.

11.1 tut11.ml

This module is organized as follows. First, we define types and functions for initializing and maintaining the state needed for translation of the VC into the `Term` language of Why3. Then, we define functions for calculating the VC for the elements of the CIL AST and translating them into the Why3 `Term` language. We begin with attribute parameters, needed for the precondition, postcondition, and loop invariant syntax that we add to C. This is followed by the translation of expressions into Why `Term`s, which is similar. Next, using the Why3 `Term`s, we define functions to generate the VC for instructions, statements, blocks, and functions. Finally, we define a function that wraps up the generated VC into a proof goal that we pass on to one of Why3's provers.

```
module W = Why3          (* We introduce aliases for the Why3 module and some of its sub-modules. *)
module T = W.Term                                  (* The Why3 library for constructing terms. *)
module Th = W.Theory                                   (* The Why3 library of theories. *)
```

We use the `ops` record type to store function symbols of the Why3 theories that we'll employ. In particular we'll use its theories of integer arithmetic, computer division, and maps.

```
type ops = {
  iplus_op : T.lsymbol;                                          (* Integer addition *)
  iminus_op : T.lsymbol;                                         (* Integer subtraction *)
  itimes_op : T.lsymbol;                                         (* Integer multiplication *)
  idiv_op : T.lsymbol;                                           (* Computer integer division *)
  imod_op : T.lsymbol;                                           (* Computer integer modulus *)
  lt_op : T.lsymbol;                                             (* Integer comparisons *)
    (* ... *)
  get_op : T.lsymbol;                                            (* get and set for maps *)
  set_op : T.lsymbol;
}
```

We use the type `wctxt` to store context for Why3. The first four fields, which we omit here, store the internal configuration of Why3, the theories it will use, and information about the back-end prover it will use. The `ops` field stores the symbols described above. The `memory` field stores a variable of Why3 type (`map int int`) that we'll use to model memory accesses. The `vars` field is a map from strings to Why3 symbols for the variables that are mentioned in the Why3 `Term` being constructed.

```
type wctxt = {
(*...*)
  mutable ops : ops;
  mutable memory : T.vsymbol;
  mutable vars : T.vsymbol SM.t;
}
```

The `initOps` function extracts the Why3 symbols for the operations from the Why3 theories that we are using.

```
let initOps (it : Th.theory) (dt : Th.theory) (mt : Th.theory) : ops =
  {iplus_op = Th.ns_find_ls it.Th.th_export ["infix +"];
   iminus_op = Th.ns_find_ls it.Th.th_export ["infix -"];
(*...*)
  }
```

The function `initWhyCtxt` initializes the Why3 context and directs it to use the prover specified by the string `p`. It uses the Why3 API to read its configuration, load its plug-ins, find the specified prover, and load the theories we'll need into the `wctxt`.

```
let initWhyCtxt (p : string) (pv : string) : wctxt =
(* ... *)
```

As with the other attributes in previous chapters, we introduce global variables for string constants for the attribute syntax.

```
let invariantAttrStr = "invariant"
let postAttrStr = "post"
let preAttrStr = "pre"
let tut11_attrs = [invariantAttrStr; postAttrStr; preAttrStr;]
```

The functions `term_of_int` and `term_of_int64` convert OCaml integers (which we'll extract from the CIL AST) into Why3 terms.

```
let term_of_int (i : int) : T.term = i |> string_of_int |> T.t_int_const
let term_of_i64 (i : int64) : T.term = i |> Int64.to_string |> T.t_int_const
```

We'll use the functions `make_symbol` and `freshvar_of_ap` to make fresh symbols for the variables bound by quantifiers that appear in attribute parameters.

```
let make_symbol (s : string) : T.vsymbol =
  T.create_vsymbol (W.Ident.id_fresh s) W.Ty.ty_int
let freshvar_of_ap (ap : attrparam) : string × T.vsymbol =
  match ap with
  | ACons(n, []) → n, make_symbol n
  | _ → Em.s(Em.error "Names only")
```

The function `term_of_attrparam` converts an attribute parameter into a Why3 term. For variable references, it looks up symbols in the `vars` field of the context. For binary and unary operations, it uses the operations defined in the `ops` field.

For memory references it uses the `get` and `set` operations and the `memory` field of the context to generate the appropriate Why3 terms. In particular, memory is modeled as one large array indexed

by integers and storing integers. Thus, the C expression `*a` is translated to `M(a)`, and the `i`'th field of array `a` is translated to `M(a + i)`.

Since attribute parameters will be used to express function pre- and post-conditions, and loop invariants, we also interpret syntax for universal quantification. The `forall` attribute parameter is parameterized by the quantified formula and the free variables of this formula. The free variables are added to the context, the attribute parameter for the quantified formula is translated to a Why3 term, and then we construct a Why3 term for the qualifier.

```
let rec term_of_attrparam (wc : wctxt) (ap : attrparam) : T.term =
  match ap with
  | AInt i → term_of_int i
  | ACons(n,[]) → T.t_var (SM.find n wc.vars)
  | ACons("forall",apl) → term_of_forall wc apl
  | ACons("implies",[a;c]) → term_of_implies wc a c
  | AUnOp(uo, ap) → term_of_apuop wc uo ap
  | ABinOp(bo, ap1, ap2) → term_of_apbop wc bo ap1 ap2
  | AStar ap → term_of_star wc ap
  | AIndex(base, index) → term_of_index wc base index
(* The rest are unimplemented. See the exercises at the end of the section. *)
  | _ → Em.s(Em.error "Attrparam is not a term: %a" d_attrparam ap)
```

The function `term_of_forall` builds a Why3 term for our universal quantifier annotation. The list of parameters to a `forall` ends with the formula we are quantifying over. We copy this last element of `apl` into `fat`. The rest of the elements of `apl` are the variables that we are quantifying over. We translate those strings into fresh Why3 symbols and store the list in `vl`. Next, we add the new symbols to the Why3 context after keeping a copy of the old map. Then, we translate `fat` in the new context, restore the old context, and then finally build the Why3 term for the quantifier.

```
and term_of_forall (wc : wctxt) (apl : attrparam list) : T.term =
  let fat = apl |> L.rev |> L.hd in
  let vl = apl |> L.rev |> L.tl |> L.map freshvar_of_ap in
  let oldm = wc.vars in
  wc.vars ← L.fold_left (fun m (n,v) → SM.add n v m) oldm vl;
  let t = term_of_attrparam wc fat in
  wc.vars ← oldm;
  T.t_forall_close (L.map snd vl) [] t
```

The function `term_of_implies` translates the antecedent and consequence, then constructs and returns the Why3 implication term.

```
and term_of_implies (wc : wctxt) (a : attrparam) (c : attrparam) : T.term =
  let at = term_of_attrparam wc a in
  let ct = term_of_attrparam wc c in
  T.t_implies at ct
```

The functions `term_of_apuop` and `term_of_apbop` translate `AUnOp` and `ABinOp` attribute parameters into Why3 terms.

```
and term_of_apuop (wc : wctxt) (u : unop) (ap : attrparam) : T.term =
  let te = term_of_attrparam wc ap in
  match u with
  | Neg → T.t_app_infer wc.ops.iminus_op [(term_of_int 0);te]
  | LNot → T.t_equ te (term_of_int 0)
  | BNot → Em.s (Em.unimp "Attribute BNot: ~%a\n" d_attrparam ap)
and term_of_apbop (wc : wctxt) (b : binop) (ap1 : attrparam) (ap2 : attrparam) : T.term =
  let te1 = term_of_attrparam wc ap1 in
  let te2 = term_of_attrparam wc ap2 in
  match b with
  | PlusA | PlusPI | IndexPI → T.t_app_infer wc.ops.iplus_op [te1; te2]
  | MinusA | MinusPI | MinusPP → T.t_app_infer wc.ops.iminus_op [te1; te2]
(* ... *)
  | _ → Em.s (Em.error "term_of_bop failed: %a %a %a\n"
              d_attrparam ap1 d_binop b d_attrparam ap2)
```

The function `term_of_star` translates a memory referene in an attribute parameter into a Why3 `get` operation of the address in the memory map of the Why3 context.

```
and term_of_star (wc : wctxt) (a : attrparam) : T.term =
  let at = term_of_attrparam wc a in
  let mt = T.t_var wc.memory in
  T.t_app_infer wc.ops.get_op [mt;at]
```

The function `term_of_index` is similar to `term_of_star`. First, though, we have to calculate the address by adding the index to the base pointer.

```
and term_of_index (wc : wctxt) (base : attrparam) (index : attrparam) : T.term =
  let bt = term_of_attrparam wc base in
  let it = term_of_attrparam wc index in
  let addr = T.t_app_infer wc.ops.iplus_op [bt; it] in
  let mt = T.t_var wc.memory in
  T.t_app_infer wc.ops.get_op [mt; addr]
```

The function `oldvar_of_ap` finds the Why3 symbol for a variable in the context.

```
let oldvar_of_ap (wc : wctxt) (ap : attrparam) : T.vsymbol =
  match ap with
  | ACons(n, []) → SM.find n wc.vars
  | _ → Em.s(Em.error "Names only")
```

Now that we can translate attribute parameters into Why3 terms, we can also interpret the syntax for specifying loop invariants. In particular, a loop invariant gives three things. First, it specifies the loop's termination condition. This could also be read out of the program, but it simplifies the code a bit to have it available here. Second, it gives the loop invariant itself. Finally, it gives a list of variables that vary in the loop. This list could also be read out of the program, but getting the list here from the programmer avoids us having to write a visitor. If the programmer forgets to list a variable, the worst that could happen is that the proof will fail.

The function `inv_of_attrs` translates a loop invariant annotation into the three Why3 terms mentioned above.

```
let inv_of_attrs (wc : wctxt) (a : attributes)
                 : T.term × T.term × T.vsymbol list
  =
  match filterAttributes invariantAttrStr a with
  | [Attr(_, lc :: li :: rst)] →
    term_of_attrparam wc lc,
    term_of_attrparam wc li,
    L.map (oldvar_of_ap wc) rst
  | _ → Em.s(Em.error "Malformed invariant attribute: %a" d_attrlist a)
```

The function `cond_of_function` extracts the Why3 terms for the pre- and post-condition annotations on C function definitionss.

```
let cond_of_function (k : string) (wc : wctxt) (fd : fundec) : T.term option =
  match filterAttributes k (typeAttrs fd.svar.vtype) with
  | [Attr(_, [ap])] → Some(term_of_attrparam wc ap)
  | _ → None
let post_of_function = cond_of_function postAttrStr
let pre_of_function = cond_of_function preAttrStr
```

Now that we have written functions to translate the new syntax elements into Why3 terms, we can calculate the VC of the postcondition. We proceed by first translating CIL expressions into Why3 terms. This is similar to our handling of attribute expressions with two exceptions. First, there are no qualifiers. Second, we treat the results of comparisons and the boolean operations (e.g. $\wedge$, $\vee$) as integer valued, using 1 for true and 0 for false. These can be converted back to boolean values for `if` statements and loop conditions as necessary.

The `iterm_of_bterm` function performs the conversion from a boolean valued Why3 term to an integer valued Why3 term. `bterm_of_iterm` performs the reverse conversion. `term_of_exp`, assisted by `term_of_uop` and `term_of_bop`, translates a CIL expression into a Why3 term. Their definitions are very similar to the analogous functions for attribute parameters, so they are omitted in the text.

```
let iterm_of_bterm (t : T.term) : T.term = T.t_if t (term_of_int 1) (term_of_int 0)
let bterm_of_iterm (t : T.term) : T.term = T.t_neq t (term_of_int 0)
let rec term_of_exp (wc : wctxt) (e : exp) : T.term = (* ... *)

and term_of_uop (wc : wctxt) (u : unop) (e : exp) : T.term = (* ... *)

and term_of_bop (wc : wctxt) (b : binop) (e1 : exp) (e2 : exp) : T.term = (* ... *)
```

With expressions translated to terms, we can now begin calculating the VC. We are using the backwards method of VC construction, but we accomplish this by traversing the program in the forwards direction, building up a continuation that will then be applied to the function's postcondition. Therefore, both `term_of_inst` and `term_of_stmt` return functions that take as an argument the term generated by *the next* instruction or statement, and yield the term modified according to *the current* instruction or statement.

This method of VC generation is not very efficient. In particular it prevents the hashcons optimizations inside of the Why3 library from being applied until the continuation we build up is finally applied. It would be more sensible, especially for dealing with large programs, to use a forwards method of VC construction.

The function `term_of_inst` generates Why3 `let` bindings for assignments. These `let` bindings automatically take care of any necessary variable renaming. Currently, the translation does not handle writes to `struct` fields. Accomplishing this will be discussed in one of the exercises.

```
let term_of_inst (wc : wctxt) (i : instr) : T.term → T.term =
  match i with
  | Set((Var vi, NoOffset), e, loc) →
    let te = term_of_exp wc e in
    let vs = SM.find vi.vname wc.vars in
    T.t_let_close vs te
      (* Also the case for a memory write *)
  | _ → Em.s (Em.error "term_of_inst: We can only handle assignment")
```

The function `term_of_stmt` generates the continuation for calculating the VC for a statement. For instruction statements, it folds over the list of instructions. For `if` statements it generates a Why3 `if-then-else` term. It recursively descends into block statements, and treats return statements as no-ops. Loop statements are more complex. Handling loops is described in detail below.

```
let rec term_of_stmt (wc : wctxt) (s : stmt) : T.term → T.term =
  match s.skind with
  | Instr il → L.fold_right (fun i t → (term_of_inst wc i) t) il
  | If(e,tb,fb,loc) → term_of_if wc e tb fb
  | Loop(b,loc,bo,co) → term_of_loop wc b
  | Block b → term_of_block wc b
  | Return(eo, loc) → (fun t → t)
```

```
(* The other cases are unimplemented *)
  | _ -> Em.s(Em.error "No support for try-finally, or try-except")
```

For the C `if` statement, the function `term_of_if` we creates a Why3 `if-then-else` term. We convert the condition of the `if` statement (`e`) to a term. The expression translation code gives an integer term, which we convert to a boolean term using `bterm_of_iterm`. Next, we obtain the VC continuations for the true and false blocks, and finally construct the VC continuation for the `if` term.

```
and term_of_if (wc : wctxt) (e : exp) (tb : block) (fb : block) : T.term -> T.term =
  let te = e |> term_of_exp wc |> bterm_of_iterm in
  let tbf = term_of_block wc tb in
  let fbf = term_of_block wc fb in
  (fun t -> T.t_if te (tbf t) (fbf t))
```

For a loop statement, we use the following translation:

$$VC(loop(inv, c, b), t) = inv \wedge \forall_{x_1,\ldots,x_n}(inv \Rightarrow (c \Rightarrow VC(b, inv)) \wedge (\neg c \Rightarrow t))$$

Where inv is the loop invariant, c is the termination condition, b is the body of the loop, t is the postcondition for the loop, and $x_1, \ldots, x_n$ are the loop variants. Furthermore, $VC(b, inv)$ is the verification condition of the loop invariant with respect to the body.

This asserts, essentially, that the invariant is true before the first iteration, that each non-terminal iteration maintains the invariant, and that when the loop exits, the postcondition is true.

CIL transforms all loops into the form:

```
while(1) {
   if(!cond) break;
   {
      ...
   }
}
```

where the innermost block statement is the original loop body. In the syntax extensions to C used for this tutorial, the loop invariant goes on this loop body block. Therefore, we must take apart the loop a bit to reach the invariant. Then, we must generate the continuation calculating the VC for the loop body, excluding the `break` statement (since we can't handle it correctly). Finally, we generate the continuation for the VC for the loop statement, which employs the loop invariant, and quantifies over the loop variant variables, including the memory.

The function `term_of_loop` performs the above translation.

```
and term_of_loop (wc : wctxt) (b : block) : T.term → T.term =
  let test, body = L.hd b.bstmts, L.tl b.bstmts in
  let body_block = body |> L.hd |> force_block in
  let bf = term_of_block wc (mkBlock (body_block.bstmts @ (L.tl body))) in
  let ct, li, lvl = inv_of_attrs wc body_block.battrs in
  let lvl' = wc.memory :: lvl in
  (fun t → t
    |> T.t_if ct (bf li)                                  (* if c then VC(b,inv) else t *)
    |> T.t_implies li                                     (* inv => previous line *)
    |> T.t_forall_close lvl' []                           (* ∀x1,...,xn (previous line) *)
    |> T.t_and li)                                        (* inv ∧ previous line *)
```

The function `term_of_block` folds over the statements of a block, processing the last statement first, to generate a continuation for the VC.

```
and term_of_block (wc : wctxt) (b : block) : T.term → T.term =
  L.fold_right (term_of_stmt wc) b.bstmts
```

The function `vsymbols_of_function` collects the Why3 symbols for the formal parameters to a function in addition to the symbol for the memory.

```
let vsymbols_of_function (wc : wctxt) (fd : fundec) : T.vsymbol list =
  fd.sformals
  |> L.map (fun vi → vi.vname)
  |> sm_find_all wc.vars
  |> L.append [wc.memory]
```

If there is a precondition, the function `pre_impl_t` returns a continuation that generates a term in which the precondition implies the VC generated for the function body and the term. Otherwise, it simply gives the continuation that generates the VC for the function body and the term.

```
let pre_impl_t (wc : wctxt) (fd : fundec) (pre : T.term option) : T.term → T.term =
  match pre with
  | None → term_of_block wc fd.sbody
  | Some pre → (fun t → T.t_implies pre (term_of_block wc fd.sbody t))
```

Finally the function `vcgen` generates a function that will take the function postcondition as the argument and produce the verification condition for the function. It does this by quantifying over the memory and formal parameters as given by `vsymbols_of_function`, with the VC continuation for the function body given by `pre_impl_t`.

```
let vcgen (wc : wctxt) (fd : fundec) (pre : T.term option) : T.term → T.term =
  (fun t → T.t_forall_close (vsymbols_of_function wc fd) [] (pre_impl_t wc fd pre t))
```

The function `validateWhyCtxt` Adds the term `p` to the Why3 context as a proof goal, and invokes the external prover. In this example the prover is given a timeout of two minutes, and the result is echoed to the terminal. One might also choose to use a proof assistant such as Coq [2] as the back-end to Why3, in which case, we could enter into an interactive proof session.

Alternately, there are a range of options for what could be done here. We could do anything from simply directing Why3 to emit all of the proof obligations for a program for examination off-line, to halting compilation and emitting an error if a proof obligation is not discharged during compilation. The right choice likely depends on the stage of development the code is in, not to mention the goals of the t

```
let validateWhyCtxt (w : wctxt) (p : T.term) : unit = (*...*)
```

The function `processFunction` initializes the Why3 context with fresh variables for the local variables and formal parameters of the function before checking to see if it has any postconditions. If so, it tries to find a precondition, generates the verification condition for the postcondition, and finally invokes `validateWhyCtxt` on the resulting term.

```
let processFunction (wc : wctxt) (fd : fundec) (loc : location) : unit =
  wc.vars ←
    L.fold_left (fun m vi → SM.add vi.vname (make_symbol vi.vname) m)
      SM.empty (fd.slocals @ fd.sformals);
  match post_of_function wc fd with
  | None → ()
  | Some g →
    let pre = pre_of_function wc fd in
    let vc = vcgen wc fd pre g in
    validateWhyCtxt wc vc
```

The function `tut11` is the entry point for this module. It initializes the Why3 context and then iterates over all functions in the file.

```
let tut11 (f : file) : unit =
  let wc = initWhyCtxt (!Ciltutoptions.prover) (!Ciltutoptions.prover_version) in
  iterGlobals f (onlyFunctions (processFunction wc));
        eraseAttrs f
```

11.2 test/tut11.c

This C source file contains an example function that we will use to demonstrate the features developed in `tut11.ml`. In particular we will verify that a function will successfully initialize an integer array to contain the number 4 at each entry.

../test/tut11.c

```
# include <ciltut.h> // For the pre, post and invariant annotations.
```

The function `arr_init` loops over the given array setting each element to 4. The precondition to the function states that the parameter `n` must be positive. The postcondition states that each element of the array is 4. The loop invariant states that the loop index stays in bounds, and that the array up to the value of the loop index is initialized to be 4.

../test/tut11.c

```
void (pre(n > 0)
      post(forall(j,implies(j>=0 && j < n,*(a+j)==4)))
      arr_init)(int *a, int n)
{
  int i;

  for (i = 0; i < n; i++)
  { invariant(i != n,
              i >= 0 && i <= n && forall(j, implies(j>=0 && j<i, *(a+j) == 4)),
              i)
    a[i] = 4;
  }

  return;
}
```

The `main` function simply invokes the `arr_init` function on a local array. The annotations we have added have no runtime effects.

../test/tut11.c

```
int main()
{
  int arr[5];

  arr_init(&arr[0], 5);

  return 0;
}
```

We can compile this example, and verify the correctness of `arr_init` by saying:

```
$ ciltutcc --enable-tut11 --prover=Alt-Ergo -prover-version=0.94 -o tut11 test/tut11.c
```

11.3 Exercises

1. Use a chain of Why3 `if-then-else` terms to calculate the VC for C's `switch` statement.
2. Handle structure fields.
3. Verify an array sorting function.
4. Extend the translation of attribute parameters and terms to handle not only integer arithmetic, but also floating point arithmetic. To achieve this, the relevant theory must be added to the Why3 context, and the operations from these theories must be added to the `ops` type.
5. Extend the checking here to also verify that function preconditions are satisfied.

Project: Create a tool that finds loop invariants given a post-condition and pre-condition for the loop. There are several approaches you might take. You could execute the program symbolically at the same time as it executes concretely (which we'll see how to do in Chapter 15), and assume to be symbolic invariants relationships that are concretely true at runtime. Alternately, you could repeatedly construct trial invariants out of every relationship among program variables that are live in the loop.

11.4 Further Reading

There are many approaches to, and implementations of, program verification. Here is a dated and incomplete list. This is a big area with many ongoing projects, so ask your local PL professor to point you in the right direction.

- VC generation and checking: Boogie [1]
- Software model checking: BLAST [4]
- Explicit path model checking: CUTE [4]

References

[1] Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '05, pages 82–87, New York, NY, USA, 2005. ACM.

[2] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer Verlag, 2004.

[3] Jean-Christophe Filliâtre. Verifying two lines of C with Why3: an exercise in program verification. In *Verified Software: Theories, Tools and Experiments (VSTTE)*, Philadelphia, USA, January 2012.

[4] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '02, pages 58–70, New York, NY, USA, 2002. ACM.

[5] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[6] Glynn Winskel. *The formal semantics of programming languages - an introduction.* Foundation of computing series. MIT Press, 1993.

Chapter 12

Comments

CIL has a very basic mechanism for tracking comments. In this chapter we'll see how to use it. CIL only sees comments when they are maintained in the output of the preprocessor. The preprocessor may be instructed to maintain comments in its output by using the `-C` switch to `ciltutcc`. Then, the comments in the source are collected by CIL's parser and placed in the array `Cabshelper.commentsGA`. `commentsGA` is a `GrowArray.t` of triples of type (`Cabs.cabsloc` × `string` × `bool`). The `cabsloc` is the source location of the comment. The `string` is the comment itself, and the `bool` is set aside for application bookkeeping.

12.1 tut12.ml

In this example, we'll visit the AST and print out comments nearby instructions and statements, taking care to only print each comment once. This will be accomplished by extracting the source location from instructions and statements and then doing a binary search on the array of comments. The array of comments lives in the `Cabshelper` module. The locations of the comments are defined in terms of the `Cabs.cabsloc` record type.

```
module GA = GrowArray
module A = Cabs
module CH = Cabshelper
```

First, we'll need a few utility functions, some of which are hidden away in `Tututil` (e.g. functions for ordering source locations and comments). `prepareCommentArray` filters out comments not from source file `fname`, and sorts the results according to source location.

```
let prepareCommentArray (cca : comment array) (fname : string) : comment array =
  cca |> array_filter (fun (cl, _, _) → fname = cl.A.filename)
      |> array_sort_result comment_compare
```

The funcion `commentsAdjacent` returns the indexes of at most two comments that are immediately

adjacent to the given source location. The indexes are into the the array returned as the second element of the return value.

```
let commentsAdjacent (cca : comment array) (l : location)
                     : int list × comment array =
  if l = locUnknown then [], cca else
  let cca = prepareCommentArray cca l.file in
  (cca |> array_bin_search comment_compare (comment_of_cilloc l)), cca
```

The function `commentsBetween` returns a list of indexes into the array returned as the second element of the return value. The indexes indicate the comments lying between source locations `l1` and `l2`. If the exact location is not in the comments array, the binary search function returns the two closest elements. Therefore `commentsBetween` returns the highest of the lower bounds, and the smallest of the upper bounds, so that only the indexes for the comments between the two locations are returned.

```
let commentsBetween (cca : comment array) (l1 : location) (l2 : location)
                    : int list × comment array
  =
  if l1 = locUnknown then commentsAdjacent cca l2 else
  if l1.file ≠ l2.file then commentsAdjacent cca l2 else begin
  let cca = prepareCommentArray cca l1.file in
  let ll = array_bin_search comment_compare (comment_of_cilloc l1) cca in
  let hl = array_bin_search comment_compare (comment_of_cilloc l2) cca in
  let l, h =
    match ll, hl with
    | ([l] | [_;l]), h :: _ → l, h
    | _ → E.s(E.bug "bad result from array_bin_search")
  in
  (Array.init (h − l + 1) (fun i → i + l) |> Array.to_list), cca
  end
```

The function `markComment` searches a `comment array` for an exact source location, and marks the third element of the tuple for that location as `true`, indicating in this example that the comment has been printed by `printComments`.

```
let markComment (l : A.cabsloc) (cca : comment array) : unit =
  Array.iteri (fun i (l',s,b) →
    if compare l l' = 0 then cca.(i) ← (l',s,true)
  ) cca
```

The function `printComments` prints the comments from the array `cca'` indicated by the indexes in `il` and marks the comments as having been printed in `cca`. The location `l` is used to indicate the source location being inspected by an instance of the `commentVisitorClass` that triggered the call to `printComments`.

```
let printComments (cca : comment array) (l : location)
                  ((il, cca') : int list × comment array) : location =
  L.iter (fun i → let c = cca'.(i) in
    if ¬(thd3 c) then begin
      markComment (fst3 c) cca;
      E.log "%a: Comment: %a -> %s\n"
        d_loc l d_loc (cilloc_of_cabsloc (fst3 c)) (snd3 c)
    end
  ) il;
  if il ≠ []
  then il |> L.rev |> L.hd |> Array.get cca' |> fst3 |> cilloc_of_cabsloc
  else l
```

The `commentVisitorClass` visitor visits the AST, printing comments nearby instructions and statements.

```
class commentVisitorClass (cca : comment array) = object(self)
  inherit nopCilVisitor
  val mutable last = locUnknown
  method vinst (i : instr) =
    last ← i |> get_instrLoc
             |> commentsBetween cca last
             |> printComments cca (get_instrLoc i);
    DoChildren
  method vstmt (s : stmt) =
    last ← s.skind |> get_stmtLoc
                   |> commentsBetween cca last
                   |> printComments cca (get_stmtLoc s.skind);
    DoChildren
end
```

The function `tut12` is the entry point for this module. First it copies the `GrowArray.t` of comments into an array. Then, it instantiates the `commentVisitorClass` visitor, and runs it over the `Cil.file` passed as an argument.

```
let tut12 (f : file) : unit =
  let cca = array_of_growarray CH.commentsGA in
  let vis = new commentVisitorClass cca in
  visitCilFile vis f
```

12.2 test/tut12.c

../test/tut12.c

```
/* With this test, we'll see if CIL's parser successfully captures comments */

int main ()
{
  int x = 1; // line comment x
  int y = 4; // line comment y
  int z;

  /* so far so good */
  z = x + y;

  /* after the instr */
  return z;
}
```

When we invoke `ciltutcc` on `tut12.c` as follows:

```
$ ciltutcc --enable-tut12 -C -o tut12 test/tut12.c
```

We get the following output:

```
test/tut12.c:7:  Comment:  test/tut12.c:3 -> With this test, we'll see if CIL's parser
successfully captures comments
test/tut12.c:7:  Comment:  test/tut12.c:7 -> line comment x
test/tut12.c:12:  Comment:  test/tut12.c:8 -> line comment y
test/tut12.c:12:  Comment:  test/tut12.c:11 -> so far so good
test/tut12.c:15:  Comment:  test/tut12.c:14 -> after the instr
```

12.3 Further Reading

Tan et al. have proposed that comparing the Natural Language semantics of comments with the Programming Language semantics of nearby code can reveal inconsistencies that could indicate the existence of bugs [1, 2].

References

[1] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /* iComment: Bugs or bad comments? */. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP07)*, October 2007.

[2] Lin Tan, Ding Yuan, and Yuanyuan Zhou. Hotcomments: How to make program comments more useful? In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems (HotOS07)*, May 2007.

Chapter 13

Whole-program Analysis

CIL has a simple mechanism for allowing whole-program analysis. This mechanism is invoked when the `--merge` switch is passed to `ciltutcc`. First, in the compilation phase, instead of compiling source code to an object file with the back-end compiler, the emitted `.o` file will contain the pre-processed source. Then, in the link stage, `ciltutcc` parses the `.o` files, and uses the CIL `Mergecil` module to combine the separate source files into a single `Cil.file`. More details of this process can be found in the official CIL documentation.

13.1 tut13.ml

In this tutorial, we'll see how to compute a whole-program call graph. The code in thie module is very simple since there is already a pretty good module for computing call graphs in `cil/src/ext/callgraph.ml`. Additionally, we'll use the `ocamlgraph` library to output a `.dot` file, which can be used to generate an image of the call-graph.

```
module H = Hashtbl
module CG = Callgraph
```

This module uses the `Ocamlgraph` library, which must be installed to build the code for this tutorial. Here, we use its functorial interface to define a module for functions on a graph type that mirrors the graph we get back from the `Callgraph` module.

```
module SG = Graph.Imperative.Digraph.ConcreteBidirectional(struct
(* ... *)
end)
```

We'll also need to define a module that extends the graph module above with functions to define properties of vertices used by Dot to draw the graph. We'll just use the defaults. These functions can be modified to add more information to the graph.

```
module D = Graph.Graphviz.Dot(struct
(* ... *)
end)
```

The `graph_of_callgraph` functions converts a CIL call-graph into an `Ocamlgraph` graph that we can use to generate a `.dot` file.

```
let graph_of_callgraph (cg : CG.callgraph) : SG.t =
  let g = SG.create () in
  H.iter (fun s n → SG.add_vertex g n) cg;
  H.iter (fun s n →
    Inthash.iter (fun i n' →
      SG.add_edge g n n'
    ) n.CG.cnCallees
  ) cg;
  g
```

The function `tut13` is the entry point for this module. It computes the call-graph, converts it to a graph for the `Ocamlgraph` graph library, and passses it to the graph library function that produces the `.dot` file.

```
let tut13 (f : file) : unit =
  let o = open_out !Ciltutoptions.tut13out in
  f |> CG.computeGraph |> graph_of_callgraph |> D.output_graph o;
  close_out o
```

13.2 Example

The difficult part of arranging for whole-program analysis is the more complicated compilation process. Here are two source files that we'll use to generate one call-graph:

In the first file, we'll declare an `extern` function `bar` and define a function `foo` that calls it.

../test/tut13a.c

```
extern int bar(int x);

int foo(int x)
{
  return bar(x);
}
```

In the second file, we'll make an `extern` declaration for the function `foo`, and define the function `bar` that in turn calls `foo`. The `main` function simply calls `bar`. (Obviously, this program is a nonsense example.)

../test/tut13b.c

```
extern int foo(int x);

int bar(int x)
{
  return foo(x);
}

int main()
{
  bar(1);
  return 0;
}
```

Now we can build this program with the whole program analysis by executing the following commands:

```
$ ciltutcc --merge -o tut13a.o -c test/tut13a.c
$ ciltutcc --merge -o tut13b.o -c test/tut13b.c
$ ciltutcc --merge --enable-tut13 --tut13-out tut13.dot -o tut13 tut13a.o tut13b.o
```

Then, we can generate a graph from the `.dot` file as follows:

```
$ dot -Tpdf tut13.dot -o tut13.pdf
```

to produce the graph in Figure 13.1. Which is the call-graph for the whole program.

When doing whole program analysis, it is also necessary to override `ar`. For example, the command:

```
$ ar r tut13.a tut13a.o
```

Should be replaced with the command:

```
$ ciltutcc --merge --mode=AR r tut13.a tut13a.o
```

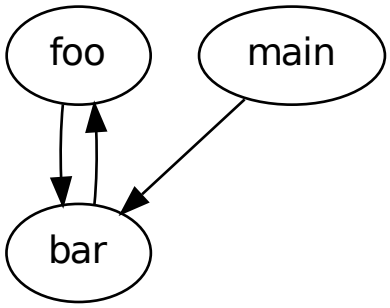

Figure 13.1: The call-graph for `tut13a.c` and `tut13b.c`

13.3 Exercises

1. Extend CIL's call-graph analysis to correctly handle function pointers by using the `Ptranal` module defined in `cil/src/ext/pta/ptranal.ml`. Note the function `Ptranal.resolve_funptr`.

13.4 Further Reading

The Capriccio [**?**] project uses a whole-program call-graph analysis to estimate the amount of stack space needed by threads in C programs. Then, in server programs that use multithreading to hide IO latency, lightweight user-level threads can be used to concurrently process a massive number of incoming client requests. The threads can be very lightweight thanks to the call-graph analysis putting an upper bound on the amount of stack space required.

Chapter 14

Implementing a simple DSL

When writing C code, one of the tasks that annoys me a lot is writing code to handle command line arguments. Thus, in this chapter we'll use CIL to put together a simple DSL for describing command line arguments. The module below will analyze a description of the arguments provided by the programmer, and generate code for parsing the command line, and checking the arguments for consistency. In particular, the goal is for the programmer to give a description like the one below at global scope:

```
argument(int, intarg) {
    .short_form = "i",
    .help_text = "An integer argument (>0)",
    .format = "%d",
    .def = 1,
    .requires = arg_assert(intarg > 0),
    .has_opt = ARG_HAS_OPT,
};
```

We can provide this syntax using CIL's attribute syntax along with judicious use of pre-processor macros. The macros, which are defined in `ciltut-include/ciltut.h`, declare variables for storing the command line arguments and their options, along with a structure type with fields for storing the properties of the argument. The code above is essentially an initializer list for those structure fields. The CIL code below will generate code to parse the command line arguments according to the options specified in the structure field using the `getopt_long` C Library call.

14.1 tut14.ml

First, we'll declare types for representing requirements for the command line arguments along with some utility functions. Then, we'll write functions for extracting the requirements from the syntax mentioned above. Finally, we'll use CIL's interpreted constructors to generate the loop that calls

`getopt_long` and uses `sscanf` to parse any options the arguemnts might have.

14.1.1 Specification Extraction

The `argument` type is where we'll collect the information the programmer gives us about the command line arguments.

```
type argument = {
  mutable argName : string;                    (* The name of the argument, i.e. --<name> *)
  mutable argType : typ;                                (* The type of the argument *)
  mutable argShort : string;                  (* The short form of the argument, i.e. -n *)
  mutable argHelp : string;                      (* The help message for the argument *)
  mutable argFmt : string;                        (* The format specifier for sscanf *)
  mutable argDef : exp;                          (* The default value of the argument *)
  mutable argReq : exp;                           (* An assertion about the argument *)
  mutable argOpt : bool;                        (* Whether the argument is optional *)
  mutable argVi : varinfo;                     (* The global varialbe for the argument *)
  mutable argGot : varinfo;       (* Global variable indicating whether the argument was given *)
  mutable argOption : bool;                  (* Whether the argument as a parameter *)
}
```

The function `makeArgument` makes a fresh `argument` initialized with dummy values.

```
let makeArgument () = (* ... *)
```

Now that we've defined the `argument` type, we can write utility functions for extracting information from type attributes. In particular, structure types that should be interpreted as argument specifications are annotated with the `ciltutarg` type attribute. Also, we extract the parameter to the `ciltut_assert` attribute on the type of the `requires` field of the argument structure to be used as an assertion that must be true of the argument, e.g. `intarg > 0` in the example above.

As usual, we begin by defining some global variables for the string constants, along with a list of the strings.

```
let argStr = "ciltutarg"
let assertStr = "ciltut_assert"
let mandatoryStr = "mandatory"
let attr_strings = [argStr; assertStr; mandatoryStr;]
```

The functions `hasArgAttr`, `hasAssertAttr`, and `hasMandatoryAttr` return true if their arguments have the indicated attributes.

```
let hasArgAttr : attributes → bool = hasAttribute argStr
let hasAssertAttr : attributes → bool = hasAttribute assertStr
let hasMandatoryAttr : attributes → bool = hasAttribute mandatoryStr
```

The functions `isArgType`, and `isMandatoryType` return true when their argument types have the indicated attributes.

```
let isArgType (t : typ) : bool = t |> typeAttrs |> hasArgAttr
let isMandatoryType (t : typ) : bool = t |> typeAttrs |> hasMandatoryAttr
```

The function `getAssertAttr` returns the attribute parameter when the type has a `"ciltut_assert"` attribute, and otherwise throws an exception.

```
let getAssertAttr (t : typ) : attrparam =
  match filterAttributes assertStr (typeAttrs t) with
  | [Attr(_, [ap])] → ap
  | _ → E.s (E.error "Malformed %s attribute: %a" assertStr d_type t)
```

The function `string_of_exp` extracts a string literal from an expression and returns it as an OCaml string. If the expression is `NULL`, it returns the empty string. If the expression is not a string, it raises an exception. We'll use it to extract structure fields for the `argument` type.

```
let rec string_of_exp (e : exp) : string =
  match e with
  | Const(CStr s) → s
  | z when z = zero → ""
  | CastE(_,e) → string_of_exp e
  | _ → E.s (E.error "Expected string literal: %a" d_exp e)
```

The function `name_of_argname` extracts the name of the argument from the name of the instance of the argument description structure.

```
let name_of_argname (s : string) : string =
  if S.length s < 8 then
    E.s (E.error "Invalid argument name: %s" s);
  S.sub s 8 (S.length s - 8)
```

Since we're using attributes to express the assertions about arguments, we'll need a few functions from Chapter 8 that we'll use to compile the attribute parameters. The function `req_of_exp` extracts the `attrparam` for an argument assertion from a `CastE` expression, and compiles it using `T.exp_of_ap`. The context `c` holds all of the global variables, so the assertion can also refer to other command line arguments.

```
module T = Tut8
let req_of_exp (c : T.ctxt) (loc : location) (e : exp) : exp =
  match e with
  | CastE(t, z) when z = zero →
    t |> getAssertAttr |> T.exp_of_ap c loc
  | _ → one
```

The command line arugment specification is the in form of a static initializer on a global variable. We'll iterate through the list of global variable declarations looking for global variable definitions (i.e. GVars) whose types have the `ciltutarg` attribute. The function `handle_field` looks at one field of the static initializer on the GVar, and uses it to fill in a field of the `argument` type we defined above.

```
let handle_field (c : T.ctxt) (loc : location) (a : argument)
                 (off : offset) (ini : init) (t : typ) ()
                 : unit
  =
  match off, ini with
  | Field(f, NoOffset), SingleInit e → begin
    match f.fname with
    | "short_form" → a.argShort ← string_of_exp e
    | "help_text" → a.argHelp ← string_of_exp e
    | "def" → a.argDef ← e;
                    a.argType ← f.ftype;
                    a.argVi ← makeGlobalVar a.argName a.argType
    | "requires" → a.argReq ← req_of_exp c loc e
    | "format" → a.argFmt ← string_of_exp e
    | "has_opt" → a.argOption ← e = one
    | _ → E.s(E.bug "malformed arg struct")
  end
  | _ → E.s(E.bug "Unexpected initializer in argument_of_global")
```

The function `argument_of_global` looks at one global, `g`, to determine whether it is a command line argument sepcification, i.e. whether it is a GVar with `isArgType` returning true. If so, it makes a new `argument`, fills in a few of the fields that can be determined immediately, like the name of the argument, and calls `iterCompound`. `iterCompound` iterates over the fields of the static initializer using the function `handle_field` that we just defined. `iterCompound` is defined in `tututil.ml`. It is based on `foldLeftCompound` from the CIL library.

```
let argument_of_global (c : T.ctxt) (g : global) : argument list =
  match g with
  | GVar(vi, {init = Some(CompoundInit(t,ini))}, loc)
    when isArgType vi.vtype → begin
    let a = makeArgument () in
```

```
    a.argName ← name_of_argname vi.vname;
    a.argGot ← makeGlobalVar (a.argName^"got") intType;
    a.argOpt ← ¬(isMandatoryType vi.vtype);
    iterCompound ~implicit:false ~doinit:(handle_field c loc a)
                 ~ct:vi.vtype ~initl:ini;
    [a]
  end
  | _ → []
```

Now that we can extract an argument specification from an arbitrary global, we can just iterate over all the globals to collect a list of all the arguments. The function `gatherArguments` performs this iteration and returns the list of found argument specifications.

```
let gatherArguments (f : file) : argument list =
  let c = T.context_for_globals f in
  f.globals
  |> L.map (argument_of_global c)
  |> L.concat
```

14.1.2 Code Generation

With the argument specifications successfully gathered, we can now generate the code that actually parses the command line. As mentioned above, we'll use the `getopt_long` function from the C Library. This process will involve three steps. First, we'll generate assignments to initialize the global variables for the options to any default values given by the programmer. Second, we'll generate a loop that calls `getopt_long` and interprets its results. Finally, we'll generate checks to make sure that the arguments are consistent with any assertions that the programmer gave.

Initializing the global varialbes and checking the programmer provided assertions is reletively easy, but initializing the `option` strucutres and generating the loop for `getopt_long` is more involved. First, we'll need a few utility functions:

The function `field_of_option` gives us the CIL `lval` for the field `n` of the `i`'th `option` structure in the array of `option`s based at `o`. We'll use it in generating `Set` instructions that initialize the array of `option`s needed for `getopt_long`.

```
let field_of_option (o : varinfo) (ot : typ) (i : int) (n : string) : lval =
  (Var o),
  Index(integer i, Field(fieldinfo_of_name ot n, NoOffset))
```

The functions `has_arg_of_argument` and `int_code_of_argument` transform bits of the argument specification into CIL expressions that we'll use to initialize fields of the `option` structures.

```
let has_arg_of_argument (a : argument) : exp =
  if a.argOption then one else zero
let int_code_of_argument (a : argument) : exp =
  a.argShort.[0] |> int_of_char |> integer
```

The function `initialize_options` takes a function for generating an `lval`, (e.g. `field_of_option`), an index, a field name, and an `argument`. Using the `argument` it generates instructions for initializing the fields.

```
let initialize_options (foo : int → string → lval)
                       (i : int) (a : argument)
                       : instr list
  =
  [Set(foo i "name", mkString a.argName, locUnknown);
   Set(foo i "has_arg", has_arg_of_argument a, locUnknown);
   Set(foo i "val", int_code_of_argument a, locUnknown)]
```

The function `create_long_options` generates code that allocates an `option` array, and fills it in. It returns the `varinfo` for the array, the call to `malloc`, and the assignments for the initializations.

```
let create_long_options (f : file) (main : fundec) (al : argument list)
                        : varinfo × instr × instr list
  =
  let malloc = findOrCreateFunc f "malloc" (mallocType f) in
  let ot = findType f.globals "option" in
  let o = makeTempVar main (TPtr(ot, [])) in
  let foo = field_of_option o ot in
  let size = integer((L.length al + 1) × ((bitsSizeOf ot)/8)) in
  let mcall = Call(Some(var o), v2e malloc, [size], locUnknown) in
  let inits = al |> A.of_list
                 |> A.mapi (initialize_options foo)
                 |> A.to_list
                 |> L.concat
  in
  o, mcall, inits
```

With an array of `option`s allocated and initialized, we can now generate the code for the loop that calls `getopt_long`. Since generating this code using type-constructors would be tedious and difficult to read, we'll use CIL's interpreted constructors instead. To do this, we need to build a string to pass to `Formatcil.cStmts`.

Before doing that, though, we'll define a few utility functions for obtaining expressions that will be parameters to the interpreted constructor. The function `create_short_options` generates an expression for the string to be used as the parameter to `getopt_long` that defines the short versions of the arguments.

```
let create_short_options (al : argument list) : exp =
  let short_arg_of_arg (a : argument) : string =
    a.argShort^(if a.argOption then ":" else "")
  in
  al |> L.map short_arg_of_arg
     |> S.concat ""
     |> mkString
```

The function `getMainArgs` just extracts the `varinfo`s for `argv` and `argv` from the list of formal parameters.

```
let getMainArgs (main : fundec) : varinfo × varinfo =
  match main.sformals with
  | argc :: argv :: _ → argc, argv
  | _ → E.s (E.error "Must give main argc and argv")
```

The function `string_of_short_arg` gives the integer code for the character used as the short form of an argument. This is needed since we can't write string or character literals in the interpreted constructor. The function `string_of_arg_opt` returns the string "1" if the argument has an option, and the string "0" otherwise.

```
let string_of_short_arg (a : argument) : string =
  a.argShort.[0] |> int_of_char |> string_of_int
let string_of_arg_opt (a : argument) : string =
  if a.argOption then "1" else "0"
```

The function `create_def_int_string` creates a string for a C `else` statement to be used for setting an argument of integer type without any options to 1.

```
let create_def_int_string (a : argument) : string =
  if isIntegralType a.argType ∧ ¬(a.argOption)
  then "else {%l:"^a.argVi.vname^"l = 1;}"
  else ""
```

The function `create_if_str` creates a string for the C code that processes a particular `argument`. The return of `getopt_long` is given in the variable `c`. If `c` is equal to the character code for the short version of the argument, then we check to see if the argument has an option, which we parse using `sscanf`. In case the argument is for a boolean flag, we include the string obtained with `create_def_int_string`. Finally, we set the global variable that indicates that the argument was given.

```
let create_if_str (a : argument) : string =
  "if (c == "^(string_of_short_arg a)^") {"^
     "if ("^(string_of_arg_opt a)^") { if(%e:optarg) {"^
       "%l:scan(%e:optarg,%e:"^a.argVi.vname^"fmt,%e:"^a.argVi.vname^"addr);"^
     "}}"^
    (create_def_int_string a)^
   "%l:"^a.argGot.vname^" = 1;"^
   "}"
```

The `if` statements created with `create_if_str` from the `arguments` form the body of the loop in which we repeatedly call `getopt_long`. The function `create_opt_loop_str` generates the string of C code for the loop. It calls `getopt_long` before proceeding into the `if` statements generated by `create\_if_str`. If the return of `getopt_long` is -1, we break out of the loop.

```
let create_opt_loop_str (al : argument list) : string =
  "while(1) {"^
     "int c;"^
     "c = %l:gol(%e:argc, %e:argv, %e:sstr, %e:lopts, (void * )0);"^
     "if (c == -1) break;"^
     (al |> L.map create_if_str |> S.concat " else ")^
   "}"
```

Finally, in the function `makeArgStmts` we can call `Formatcil.cStmts` to generate the CIL statements for argument processing. The calls to `findOrCreateFunc` and `findGlobalVar` retrieve `varinfos` from the CIL file that we'll need to pass as parameters to the interpreted constructor. Here, we'll also call the functions that generate the code for allocating and initializing the `option` array.

```
let makeArgStmts (f : file) (main : fundec) (al : argument list) : stmt list =
  let gol = findOrCreateFunc f "getopt_long" intType in
  let scan = findOrCreateFunc f "sscanf" intType in
  let optarg = findGlobalVar f.globals "optarg" in
  let so = create_short_options al in
  let o, m, i = create_long_options f main al in
  let argc, argv = getMainArgs main in
  (L.map i2s (m :: i)) @
  Formatcil.cStmts (create_opt_loop_str al)
   (fun n t → makeTempVar main ~name:n t) locUnknown
   ([("argc", Fe(v2e argc));
     ("argv", Fe(v2e argv));
     ("sstr", Fe so);
     ("lopts", Fe (v2e o));
     ("gol", Fl(var gol));
     ("scan", Fl(var scan));
     ("optarg",Fe(v2e optarg))]@
     (L.map (fun a → (a.argVi.vname^"fmt"), Fe (mkString a.argFmt)) al)@
```

```
    (L.map (fun a → (a.argVi.vname^"addr"),Fe (AddrOf(var a.argVi))) al)@
    (L.map (fun a → (a.argVi.vname^"l"), Fl(var a.argVi)) al)@
    (L.map (fun a → (a.argGot.vname), Fl(var a.argGot)) al))
```

The function `initArgs` generates statements that initialize the global variables for arguments with their default values.

```
let initArgs (al : argument list) : stmt list =
  L.map (fun a → i2s(Set(var a.argVi, a.argDef, locUnknown))) al
```

The function `printHelp` generates statements using `printf` that print the help text for the arguments to standard out. The help text gives the short version, the long version, the default value, and tells whether the argument is mandatory. The statements generated by `printHelp` will be used in case some checks on the runtime argument values fails.

```
let printHelp (f : file) (al : argument list) : stmt list =
  let print = findOrCreateFunc f "printf" intType in
  let s s = mkString s in
  [i2s(Call(None,v2e print,[s "Improper arguemnts\n"],locUnknown))]@
  (L.map (fun a →
    let af = if a.argFmt ≠ "" then a.argFmt else "%d" in
    let fmt = if a.argOpt
      then s ("\t-%s,--%s\t%s (default="^af^")\n")
      else s "\t-%s,--%s\t%s (mandatory)\n"
    in
    let args = if a.argOpt
      then [fmt;s a.argShort;s a.argName;s a.argHelp;a.argDef]
      else [fmt;s a.argShort;s a.argName;s a.argHelp]
    in
    i2s (Call(None,v2e print,args,locUnknown))
  ) al) @
  [mkStmt (Return(Some mone, locUnknown))]
```

The function `makeArgChecks` generates an `if` statement that checks two things. First, it checks that all mandatory arguments have been given. Second, it checks that all assertions stipulated by the programmer on the arguments are true. If either of these fails, the statements given by `printHelp` are used to print the help messages, and exit from the program with a failure code.

```
let makeArgChecks (f : file) (al : argument list) : stmt =
  let got_arg a = if a.argOpt then one else v2e a.argGot in
  let bexp, mexp =
    L.fold_left (fun b a → BinOp(LAnd, b, a.argReq, intType)) one al,
    L.fold_left (fun b a → BinOp(LAnd, b, (got_arg a), intType)) one al
  in
  mkStmt (If(BinOp(LOr,UnOp(LNot,mexp,intType),UnOp(LNot,bexp,intType),intType),
             mkBlock (printHelp f al), mkBlock[],
             locUnknown))
```

If we have found the main function, then the function `processFunction` inserts the code for processing command line arguments at the top of the function.

```
let processFunction (f : file) (al : argument list)
                    (fd : fundec) (loc : location) : unit =
  if fd.svar.vname = "main" then
    fd.sbody.bstmts ← (initArgs al) @
                      (makeArgStmts f fd al) @
                      [makeArgChecks f al] @
                      fd.sbody.bstmts
```

Finally, the function `tut14` gathers the argument specifications using `gatherArguments` before calling `processFunction`, which generates the argument parsing code.

```
let tut14 (f : file) : unit =
  f |> gatherArguments
    |> processFunction f
    |> onlyFunctions
    |> iterGlobals f;
  eraseAttrs f
```

14.2 test/tut14.c

This C code demonstrates the language extension we wrote for generating code to process command line arguments. It will accept one argument called `boolarg`, and another called `intarg`.

../test/tut14.c

```
#include <unistd.h>
#include <getopt.h>
#include <ciltut.h> // This is where the argument macro is #define'd
```

The `boolarg` argument can be passed with `--boolarg` or `-b`, and its value can be accessed in the program through the global variable `boolarg` of type `int`. When not given, the default value is 0 or false.

../test/tut14.c
```
argument(int, boolarg) {
  .short_form = "b",
  .help_text  = "A boolean argument",
};
```

The program also accepts an argument called `intarg` that specifies a particular integer. Setting the `has_opt` field to `ARG_HAS_OPT` indicates that the argument takes an option, which is parsed using the given format specifier. It can be passed with `--intarg n` or `-i n`, and then the value `n` will be accessable in the program through the variable `intarg`. We have also stated a requirement that the given number must bigger than 0.

../test/tut14.c
```
argument(int, intarg, mandatory) {
  .short_form = "i",
  .help_text  = "An integer argument (>0)",
  .format     = "%d",
  .requires   = arg_assert(intarg > 0),
  .has_opt    = ARG_HAS_OPT,
};

int main(int argc, char *argv[])
{
  printf("%d %d\n", boolarg, intarg);
  return 0;
}
```

This example can be compiled by saying:

```
$ ciltutcc --enable-tut14 -o tut14 test/tut14.c
```

Then we can run `tut14` and supply the correct arguments:

```
$ ./tut14 -i 4 --boolarg
1 4
```

But now let's see what happens when we get it wrong:

```
$ ../tut14 --intarg -5
Improper arguemnts
   -b,-boolarg A boolean argument (0)
   -i,-intarg An integer argument (>0) (mandatory)
```

Chapter 15

Automated Test Generation

In this tutorial we will use CIL to instrument programs with calls to an SMT solver such that running annotated functions will generate a high-coverage set of test inputs for those functions. For example, if a function definition is annotated as follows:

```
void (autotest foo)(int input a, int input b) {...}
```

then a set of `(a,b)` pairs will be generated such that as many conditionals in `foo` as possible have both true and false branches taken when run over the set of pairs.

This approach works as follows. The `autotest` functions are instrumented such that they run both concretely (as usual) and symbolically (in terms of `input`s) when called. When run symbolically, the `autotest` runtime builds a conjunction of true conditionals for each executed branch. This conjunction is called the *path condition*. If we were to send this path condition to our SMT solver, it would generate a model that would drive the program down the same path, taking the same branches, and generating the same path condition. However, if we change the sense of one of the conjuncts in the path condition, the SMT solver will generate a model that should drive the program down a different path, in particular taking the branch corresponding to the conjunct whose sense was was changed in a different direction than before.

Consider the example `autotest` function in Figure 15.1. Using random `a`'s and `b`'s, it would be difficult to cause the the true branch of either of the conditionals to be taken. Suppose, however, that we run the function concretely with the call `foo(0,0)`, and also run the program symbolically in terms of inputs `a0` and `b0`.

The symbolic execution generates the path condition given by Figure 15.2. If we negate one of the disequalities, for example if we replace (d != 700) with (d == 700), and ask an SMT solver to give a model (that is, an assignment of the free variables `a0` and `b0`) that causes the modified path condition to be true, then we will have obtained inputs to `foo` that cause the second branch to be taken. This process is repeated until as many conditionals as possible in `foo` have had both branches taken by the set of generated inputs.

```
void (autotest foo)(int input a, int input b) {
  int c, d, e;
  c = a * b;
  d = a + b;
  e = c - d;
  if (e == 14862436) explode();
  if (d == 700) explode();
  return;
}
```

Figure 15.1: Example

```
let a = a0 in
let b = b0 in
let c = a * b in
let d = a + b in
let e = c-d in
(e != 14862436) &&
(d != 700)
```

Figure 15.2: Path Condition

15.1 Background

This approach goes by many names including *directed automated random testing*, *concolic testing*, *whitebox fuzzing*, and *smart fuzzing*. Further, researchers have created a number of tools implementing the approach, and variations on it, for a number of different languages. These include DART [3], CUTE [4], CREST [1], and PEX [5].

In particular, CREST also uses CIL as compiler front-end, and Yices [2] for the SMT solver as we will here, and is much more complete than the implementation in this tutorial. Therefore, as a starting point for further investigation of automated test generation based on CIL, CREST is likely to be a more appropriate choice. However, one possible advantage to this simpler tutorial implementation is that it uses OCaml rather than C++ to implement the calls to the SMT solver by using the features of the OCaml runtime that allow OCaml calls to be made from C code.

Since these more complete tools exist, for the purposes of this tutorial we'll make some simplifying assumptions. In particular, this implementation will handle only scalar values, and regular and null-terminated arrays of scalar values. That is, `struct` and `union` types are not handled. Also, Only functions annotated `autotest` and `instrument` will be instrumented for symbolic execution. If a non-instrumented function is called from within an `autotest` function, only its concrete return value will be used in the path condition. In other words, inputs will not be generated to explore functions not annotated `autotest` or `instrument`. Finally, our path-exploration algorithm will give up when the SMT solver is unable to generate a new model for any of the available branches whose sense could be flipped. A more complete implementation would avoid getting stuck in this case.

15.2 Organization

A bit more code than in previous tutorials is required to implement these features, so instead of listing and commenting on all of it, we'll take a short tour through a few select functions, types, and modules to get an idea of how the code works, and the high-level ideas behind it.

15.2.1 Instrumentation

The code using CIL to instrument a program with calls to the SMT solver is in source file `src/tut15.ml`. Before carrying out the instrumentation however, we use CIL's `Simplify` module to break down complex expressions and l-values. A full description of its effects can be found in the CIL documentation. For now, it suffices to point out that expressions are simplified to the extent that all binary and unary operations operate only on constants or l-values. This is achieved by the `Simplify` module by introducing additional temporary variables and assignments.

The instrumentation calls notify the automated testing runtime of a number of important events: assignments, conditionals, function calls and returns, and entering and leaving an `autotest` function.

For assignments and conditionals, the calls are passed both the addresses and values of the operands and results. Including the concrete values allows the symbolic execution to under-approximate the concrete execution when the SMT solver lacks a theory for some operation performed by the program. In particular, instead of representing the operation symbolically, the SMT solver can under-approximate the program's behavior by using the concrete values. This under-

approximation may prevent the SMT solver from producing a new path through the code, but strategies for achieving high coverage exist in this case, and are implemented by the above mentioned tools.

To handle function calls, the mapping of actual parameters to formal parameters is handled like assignments. To handle function return values, a call is made so that the value can be remembered by the runtime, and then assigned at the call site.

A loop is wrapped around each call of an `autotest` function. This loop notifies the runtime when an `autotest` function is completed. If the runtime is able to produce an input that may explore a new path, the `autotest` function is called again. Otherwise, the loop terminates. This loop is constructed in the `makeTestLoop` function by way of an interpreted constructor.

15.2.2 `autotest` Runtime

The runtime library for this tutorial is divided into three parts. First, we maintain a symbolic memory that maps concrete addresses to expressions in terms of inputs. Second, at the end of an `autotest` function, we translate these expressions into expressions of the Yices SMT solver in order to obtain a path condition. Finally, we maintain a set of path conditions from which we may choose a path to modify for submission to Yices in the hope of obtaining inputs that will explore a new path. The source for the runtime library can be found in `ciltut-lib/src/concolic`. File names mentioned below are relative to this path.

The file `concolic_exp.ml` implements our internal expression representation. We define our own type for expressions instead of translating directly to Yices expressions so that we leave open the possibility of using other SMT solvers in the future. The file `concolic_ctxt.ml` implements the mapping from memory addresses to our symbolic expressions. It also maintains state for Yices, and includes a function `yices_of_exp` that translates our symbolic expressions into Yices expressions. In `yices_of_exp` the bit shifting operations are not translated into symbolic Yices expressions. This is because the Yices theory for bitvectors is only capable of reasoning about shifts by constant amounts. Therefore, `yices_of_exp` simply uses the concrete result of the operations.

Finally, the file `concolic_paths.ml` implements the path exploration algorithm. The global record `pState` maintains a list of already explored paths along with a list of conditions for the current path. It also maintains two mappings. The `branchHT` hash table maps branch identifiers to the state of a branch. A branch identifier is a combination of an id assigned at compile time to each conditional, and a count of the number of times a conditional has been executed on a single concrete run. The count is kept so that the possibly many executions of the same static conditional are not improperly conflated by the path exploration algorithm. This is important, for example, so that each time a loop guard is executed, it is considered a different conditional. The state of a branch is either `NeitherTaken`, `TrueTaken`, `FalseTaken`, or `BothTaken`. The second hash table, `branchCounts`, keeps track of how many times a conditional has been executed, so that branch id's may be properly assigned.

When a conditional is executed, its id is determined by checking its static id and looking it up in `branchCounts`. Then, the condition is appended to `pathCond`. When an `autotest` function ends, the expressions in `pathCond` are translated to Yices expressions, and the resulting path condition is appended to `paths`, and `branchHT` is updated. Then, `paths` is searched for a path condition having

the property that flipping one of its conjuncts should cause Yices to generate a model that explores an unexplored branch, as recorded by `branchHT`. This search and conjunct flipping is carried out by the `genNextPath` function. If no such path is found, a value is returned indicating that the loop around the `autotest` function should terminate. If Yices does find a model, the values are stored such that they may be requested in the `autotest` loop body, and stored in local variables so that they may be given as input in the next iteration.

15.3 test/tut15.c

This file shows how the features in this tutorial can be used. It defines two functions, one that does arithmetic using its arguments, and another that does a string comparison. These two functions are annotated `autotest` so that the test generation runtime will execute them repeatedly until each conditional has had both branches taken. The string comparison function is annotated `instrument` so that it will be executed symbolically when called from an `autotest` function.

../test/tut15.c

```
# include <ciltut.h>
```

`string_compare` has the behavior of `strcmp`. We include the code for it here explicitly so that it can be instrumented and explored by the test generation runtime when called from `g` below. On each iteration the conditionals are considered distict from the conditionals executed in previous iterations. The result is that input strings of various lengths will be generated until the SMT solver is no longer able to generate a model.

../test/tut15.c

```
int (instrument string_compare)(char *a, char *b)
{
        int i = 0;

        while (1) {
                if (a[i] > b[i]) return 1;
                if (a[i] < b[i]) return -1;
                if (a[i] == 0 && b[i] == 0) break;
                i++;
        }

        return 0;
}
```

The function `f` is annotated as an `autotest` function. The test generation runtime will take the given inputs from the text of the program and use them to generate the first path through the function. For this function, the first input is 0 for `a` and 0 for `b`. With this input the false branch of the conditional is taken. When the path condition is negated and passed to Yices, an `a` and `b` are returned such that `(a * b) - (a + b) == 14862436`.

../test/tut15.c

```
uint64_t (autotest f)(int input a, int input b)
{
        if ((a * b) - (a + b) == 14862436) {
                return 1;
        }
        else return 0;
}
```

The function `g` compares the string given by its argument with the string `"autotest"`. The type of the argument `s` has the annotation `inputnt` indicating that it is a null-terminated array. This indicates to the test generation runtime that it must generate valid null-terminated strings when concocting new inputs for `g`.

../test/tut15.c

```
uint64_t (autotest g)(char *inputnt s)
{
        return string_compare(s, "autotest");
}

int main ()
{
        uint64_t res;
        char cheese[1024] = "cheese";

        res = f(0,0);
        res = g(cheese);

        return res;
}
```

This example can be compiled by invoking the following command:

```
$ ciltutcc --enable-tut15 -o tut15 test/tut15.c
```

Then invoking the resulting program, `tut15`, produces the following output:

```
autotest f:
   a <- 0
   b <- 0
   a <- 2
   b <- 14862438
autotest g:
   s <- cheese
   s <- 0
```

```
s <- a
s <- au
s <- av6
s <- aut
s <- auu0
s <- auto
s <- autp6
s <- autot
s <- autou0
s <- autote
s <- autotp8
s <- autotes
s <- autotew6
s <- autotest
s <- autotesu
```

References

[1] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 443–446, Washington, DC, USA, 2008. IEEE Computer Society.

[2] Bruno Dutertre and Leonardo de Moura. A fast linear-arithmetic solver for dpll(t). In *Proceedings of the 18th international conference on Computer Aided Verification*, CAV'06, pages 81–94, Berlin, Heidelberg, 2006. Springer-Verlag.

[3] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[4] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[5] Nikolai Tillmann and Jonathan De Halleux. Pex: white box test generation for .net. In *Proceedings of the 2nd international conference on Tests and proofs*, TAP'08, pages 134–153, Berlin, Heidelberg, 2008. Springer-Verlag.

Index