Carillon*—a System to Find Y2K Problems in C Programs

Martin Elsman, Jeffrey S. Foster, and Alexander Aiken
Computer Science Division, University of California, Berkeley
bane-software@cs.berkeley.edu

July 30, 1999

"Any man's death diminishes me, because I am involved in mankind; and therefore never send to know for whom the bell tolls; it tolls for thee."—John Donne

Abstract

Carillon is a simple, fast, and effective type-based system for finding Y2K errors in C programs. Carillon extends the standard C type system with a user-defined set of date-related *type qualifiers*. The user annotates date-related functions with the appropriate qualifiers, and Carillon checks the program for Y2K errors.

Carillon displays the results of the Y2K analysis in an interactive Emacs buffer. Program variables are colored according to the kind of Y2K information they may contain, and the user can click on program variables to see the exact Y2K type inferred by the analysis.

The system has been used successfully to verify Y2K readiness of programs and to locate Y2K errors.

1 Introduction

The Y2K problem, or the *millennium bug*, happens when a program represents years using only two digits. If the year 2000 is represented by the string `"00"`, then a program may not be able to tell the difference between the year 2000 and the year 1900. As we enter the new millennium, such bugs can lead to system crashes, or worse, a seemingly-working program that computes the wrong results.

Because most of the world's legacy code is written in COBOL, the commercial marketplace has focused on Y2K bugs in the business and financial applications written in COBOL. But these applications are not the only ones with potential damaging Y2K bugs. Control

*Carillon is also the name of the bells of the Sather Tower at University of California, Berkeley.

software for embedded systems and operating system software are examples of systems where dates play a critical rôle. Many such systems are written in the C programming language, and the commercial Y2K tools available for C are not as sophisticated as the COBOL tools. While the C Standard Library provides Y2K-safe date operations, it is common that programs manipulate dates directly as strings of characters—for instance, to interact with other programs. It is essential to establish that such systems run as expected when we enter the new millennium.

Carillon is an easy-to-use type-based system for finding Y2K problems in C programs and for showing that such problems do not exist. Carillon has the following important features:

- Carillon points the user to Y2K related problems and makes her concentrate on the parts of the program that manipulate dates. The system can analyze source files independently for quick and easy use. It also supports whole-program analysis for improved precision.

- Y2K safety is guaranteed up to casting; Carillon provides an overview of the Y2K unsafe casts in the analyzed program.

- Analysis of industrial-sized programs is supported, even with whole-program analysis. Whole-program analysis of a 57,000 line program (132,000 lines preprocessed) takes 137 seconds on a 300MHz Pentium II.

- Annotations are necessary only where dates are manipulated. Because Carillon provides type inference and qualifier polymorphism, relatively few annotations are needed.

- Carillon is easily integrated with other tools, such as compilers. Analysis results are presented to the user in an interactive Emacs buffer and can be browsed through using the mouse or the keyboard.

Carillon has been used to verify that RCS (Revision Control System) version 5.6.0.1 does not contain any Y2K errors. RCS is about 17,000 lines of C (41,000 lines of preprocessed C.) The experiment took only two hours: This time was spent partly on instrumenting the RCS Makefile to output preprocessed code, partly on annotating the main header file of RCS, and partly on solving type conflicts that were not Y2K errors.

Carillon has also been used to locate a millennium bug in CVS (Concurrent Version System) version 1.9. CVS is about 57,000 lines of C (132,000 lines of preprocessed C.) The millennium bug is fixed in CVS version 1.10.

There are three technical, research-related contributions from the development of Carillon:

- The system is a demonstration of how a program analysis can be composed from components of the Berkeley ANalysis Engine (BANE). BANE provides a complete infrastructure for developing program analysis applications, including language front-ends, efficient algorithms for solving different kinds of constraints, and a customizable user-interface called PAM (for Program Analysis Mode) for visualizing the results of a program analysis in Emacs. Using BANE, Carillon was developed in one month.

- The Carillon type system is a result of an ongoing effort at providing an *open type-system*, in which the user (or analysis implementor) can modify the typing rules for specific needs. This open type-system is based on the notion of qualifiers [FFA99].
- Carillon supports qualifier polymorphism, which decreases the number of required Y2K annotations. Qualifier polymorphism is an enhancement over other type-based tools for finding Y2K errors in COBOL programs [EHM+99].

In the next section, we give information about obtaining and installing Carillon. In Section 3, we show a first example of finding a Y2K bug in a C program. The type system that Carillon uses and various aspects of how to use Carillon is described in Sections 4 and 5. In Section 6, we give an example from the use of Carillon to verify Y2K readiness of RCS. Qualifier polymorphism and how Carillon can analyze multiple files at once is described in Sections 7 and 8. The PAM Emacs interface is documented in Section 9. Information about the authors and a conclusion is given in Sections 10 and 11.

2 Installation

For the installation, we assume some familiarity with Emacs and UNIX. Carillon requires GNU Emacs 20.2.1 or later.

Carillon is shipped as a gzipped tar-file, which can be downloaded from the web page

`http://bane.cs.berkeley.edu/carillon`

There are versions of Carillon for X86-Linux, Sparc-Solaris, and HPPA-HPUX. When you have downloaded the gzipped tar file, named `Carillon_X.tar.gz`, where `X` denotes the platform you are using, execute the commands

```
gunzip Carillon_X.tar.gz
tar xf Carillon_X.tar
```

These commands create a directory called `Carillon` with the following file and directories:

`copyright`	The copyright notice
`bin/`	Carillon executable
`emacs/`	Elisp code for displaying the analysis results in Emacs
`example/`	Example directory
`doc/`	Documentation

Now execute the commands

```
cd Carillon
./setup
```

The `setup` script generates a few lines of Emacs code to put in your `.emacs` file (see Section 2.1). The script also makes Carillon executable from the directory in which it is installed.

2.1 Customizing Emacs

Before you can use Carillon, you need to add to your `.emacs` file the Emacs code that the `setup` script writes to the file `emacs/personal.el` during setup. This file contains (after running the `setup` script) the following lines, with the difference that the directory path `/home/mael` is modified for your environment:

```
(setq load-path (append (list "/home/mael/Carillon/emacs/pam"
                              "/home/mael/Carillon/emacs/pam/elib")
                        load-path))
(autoload 'pam-analyze-file "pam-3" "Carillon Version 1.0" t)
(setq pam-default-analysis '("/home/mael/Carillon/bin/carillon"
                             "-config"
                             "/home/mael/Carillon/examples/config.d"
                             "-prelude"
                             "/home/mael/Carillon/examples/prelude.i"))
(fset 'carillon 'pam-analyze-file)
```

2.2 Changing the PAM Colors

Carillon comes with a set of predefined colors used to display analysis information. These colors are designed to work well with a grey background, and you may need to change them to suit other color schemes. You can customize the PAM colors by adding the following lines to your `.emacs` file and changing the colors to whatever your prefer.

```
(custom-set-faces
   '(pam-color-1 ((t (:foreground "Red" :underline t))) t)
   '(pam-color-2 ((t (:foreground "Blue" :underline t))) t)
   '(pam-color-3 ((t (:foreground "Turquoise" :underline t))) t)
   '(pam-color-4 ((t (:foreground "Green" :underline t))) t)
   '(pam-color-5 ((t (:foreground "Violet" :underline t))) t)
   '(pam-color-6 ((t (:foreground "GreenYellow" :underline t))) t)
   '(pam-color-7 ((t (:foreground "Magenta" :underline t))) t)
   '(pam-color-8 ((t (:foreground "Thistle" :underline t))) t)
   '(pam-color-mouse ((t (:foreground "White"
                          :background "Grey" :underline t))) t))
```

The use of colors 1-8 is determined by a configuration file passed to Carillon (see Section 5.5). The last color, `pam-color-mouse`, is the color with which hyperlinks are highlighted when the mouse pointer is moved on top of them. A copy of this code can be found in `emacs/pam_colors.el`.

Another possibility is to change the Emacs background color by typing M-x set-background-color. We recommend selecting "White" to make the default PAM colors most readable.

3 First Example

In this section, we demonstrate how Carillon can be used to find a Y2K error in a C program. Consider the following program, found in `examples/simple1.c`:

```
int printf(const char * format, ...);
void pr_year(char * year) {
   printf("The year is 19%s", year);
}
int main() {
   pr_year("99");
   pr_year("2000");     /*1*/
   return 0;
}
```

Here the programmer's intention is that the function `pr_year` is applied to strings that consist of two digits, representing a year after the year 1900. As we can see in line `/*1*/`, the function `pr_year` is not applied to only two-digit years, but also to the four-digit year `"2000"`. The problem here is that years are represented differently in different parts of the program.

Our tool does not assume anything about the functions or strings that appear in a program—after all, `"99"` could represent the year 1999, the programmer's age, or the expected temperature in degrees Fahrenheit.

Instead of guessing which strings represent dates, Carillon requires that the programmer provides information about her intentions with *qualifier annotations*. In this case, we add annotations to mark two-digit years and four-digit years (see `examples/simple2.c`):

```
int printf(const char * format, ...);
void pr_year(char * $YY year) {
   printf("The year is 19%s", year);
}
int main() {
   pr_year((char * $YY)"99");
   pr_year((char * $YYYY)"2000");     /*2*/
   return 0;
}
```

The annotation on the parameter of the `pr_year` function indicates that it may take only a two-digit year as an argument. Carillon assumes that the type of all string literals is `char *$NONYEAR`, that is, strings by default do not contain dates. Because the strings `"99"` and `"2000"` in this case do contain dates, we cast their types to `char *$YY` and `char *$YYYY`, respectively.

Notice that only those parts of a program that manipulate dates need be annotated. For our small example that was most of the program, but in practice, almost all of a program can remain unannotated.

3.1 Finding the Y2K Error

Assuming that Carillon is already installed, as described in Section 2, you can now run Carillon on the example program. From within Emacs, type `M-x carillon` and enter the string `"examples/simple2.c"` (assuming you are in the directory where Carillon is installed) when asked for the file to analyze. The system analyzes a prelude file `examples/prelude.i` and then the file `examples/simple2.c`, which uses the `printf` function declared both in the file `examples/simple2.c` and in the prelude file. Carillon displays the result in an Emacs buffer.

As we expect, Carillon complains with an error message:

```
/home/mael/Carillon/examples/simple2.c:9.4-9.11
 Error during analysis of ``pr_year((char *$YYYY ) "2000")''.
 The qualifier $YY does not match the qualifier $YYYY.
```

If you click the middle mouse-button on the highlighted portion of the error message (called an *overlay*), Carillon will move the cursor to the location in the program where the error occurs. If you would rather use the keyboard to select an overlay instead of using the mouse, you can place the cursor over the overlay and type `C-c C-l`. Identifiers in the program are highlighted with colors that classify what qualifiers appear in the type of a given identifier. If you click on a highlighted identifier, Carillon shows the type of the identifier in the mini-buffer. If the type does not fit in the mini-buffer, the system shows the type in a dedicated buffer. For instance, to understand the type error in the program, observe that the type of `pr_year` is a function from `($YY ptr(num))` to `void` and that the type of the argument to `pr_year` in line `/*2*/` is `$YYYY ptr(num)`.

Carillon also complains with a warning, which you can see if you click on the overlay `Unsafe Var-Arg Apps` in the `Carillon Results` buffer:

```
There was 1 instance of a variable-length argument function
being unsafely applied to a year-involved argument:

/home/mael/Carillon/examples/simple2.c:2.4-2.10
  Argument ``year'' to function ``printf''
  has type  $YY ptr(num)
```

The declaration of `printf` in the prelude file specifies a type only for its first parameter. Hence, to be safe, Carillon gives warnings if qualifiers other than `$NONYEAR` occur in the types of arguments passed for `...` in an application of a variable-length argument function. In this case, the argument to the `printf` function can safely be cast to `char *$NONYEAR`, which makes the warning disappear (file `examples/simple3.c`).

The prelude file is used to give library functions more refined types than those given in the programs themselves. The different types for the `printf` function given by the declarations in the prelude file and in the file `examples/simple1.c` is one example.

Carillon does not try to correct the possible Y2K errors that it finds. Instead, it is the responsibility of the programmer to modify the program so that years are used consistently.

4 The Type System

Carillon is a type-based analysis tool. As described in the previous example, the programmer annotates her program by adding type qualifiers to the program. Carillon finds Y2K bugs by performing type checking.

Carillon extends the C type system in four ways:

1. The programmer can use an extensible set of user-defined type qualifiers (e.g., `$YY` and `$YYYY`) in addition to the C qualifiers `const` and `volatile`.

2. Certain operations on data that contain years are disallowed. These restrictions are enforced by requiring certain types to be qualified as `$NONYEAR`.

3. Qualifier polymorphism allows functions to have different types for each use. See Section 7.

4. Multiple files can be type checked together, thereby enforcing type consistency across files. See Section 8.

Carillon assumes that the input program is a type-correct C program. While Carillon will detect many C type errors, it does not display as much useful information about C type errors as a C compiler.

The remainder of this section describes the extensions noted above and discusses some of the limitations of Carillon.

4.1 Qualifiers and Qualified Types

We begin by introducing the types that Carillon uses to perform its analysis.

An *identifier* is a C identifier not starting with `_`. A *type qualifier* is a token `$`*id*, where *id* is an identifier. Thus, `$NONYEAR`, `$YY`, and `$YYYY` are examples of qualifiers. The qualifier `$NONYEAR` is built into Carillon, while other qualifiers must be described in a *configuration file* that is read by Carillon. See Section 5.5 for more information about the configuration file and how colors are associated with the qualifiers.

A *qualifier variable* is a token `$_`*id*, where *id* is an identifier. Thus, `$_1`, `$_q`, and `$_q1` are examples of qualifier variables. Qualifier variables are used for introducing functions with polymorphic function types. See Section 7.

Carillon allows type qualifiers and qualifier variables to appear in any position where C allows the qualifiers `const` and `volatile`. For example, `(char * $YYYY)` is a type representing strings containing a four-digit year. Similarly, the declaration

```
char * $YYYY f(char * $YY a);
```

declares `f` to be a function that takes as argument a string containing a two-digit year and returns a string containing a four-digit year. Sometimes we differentiate between *Carillon types*, which may contain user-defined qualifiers and qualifier variables, and C types, which may not.

As the reader may have noticed in Section 3, the types displayed by Carillon are slightly different from the usual C types. Carillon uses and displays types given by the following grammar:

type	::=	⟨ *qual* ⟩ `num`	any numeric type
	|	`void`	void
	|	⟨ *qual* ⟩ `ptr(` *type* `)`	pointers and arrays
	|	`(` *type** `) ->` *type*	functions
	|	⟨ *qual* ⟩ { (*label* : *type*)* }	structures and unions

Here *qual* ranges over qualifiers and qualifier variables and *label* ranges over structure field-names. There are several important things to notice. First, all numeric C types (e.g., `int`, `char`, and `float`) are represented with the same Carillon type, `num`. Second, both pointers and arrays are represented by `ptr`.[1] Third, both structures and unions are represented the same way. This treatment of unions is consistent with C, in which unions can be used to make unsafe casts. Finally, `num` types, `ptr` types, and structure types can appear with a qualifier or a qualifier variable. For the purpose of finding date errors in programs that represent dates as strings, one can ignore qualifiers that appear in other than string types. However, the richer syntax can be useful for finding other abstraction violations.

4.2 Carillon Type Rules

Carillon's type system forces date strings to be used consistently. In an assignment `a = b`, Carillon requires the types of `a` and `b` to match. In a call `f(x)`, `x` must match the type of `f`'s formal parameter.

Unlike C, Carillon assumes that unspecified type qualifiers may be anything. For example, consider the following code:

```
char * $YY s1;
char * s2;
char * s3;
s2 = s1;
s1 = s3;
```

Because `s2` and `s3`'s types contain no year-related qualifier, Carillon assumes that any qualifier could appear (more formally, Carillon automatically inserts a qualifier variable). Hence the system infers that both `s2` and `s3` must have type `char * $YY`.

It is this *type inference* process, in which Carillon computes the necessary type annotations inferred by the program structure, that makes the system easy to use by minimizing the number of programmer-supplied annotations. As one might imagine, such a system can be useful for more than just date strings; see Section 11 for a discussion.

In addition to enforcing consistency for assignments and function calls, Carillon's type systems forces certain types to be qualified by `$NONYEAR`. Intuitively, the kind of errors we are interested in for the Y2K problem are abstraction violations. Years are represented concretely by strings (type `char *`), but they should be manipulated only by certain routines.

[1] Internally Carillon uses the C types to correctly handle multidimensional arrays and such.

Carillon has three new kinds of type rules to enforce this abstraction:

- String literals are given the type `(char * $NONYEAR)`. This rule ensures that no string literal is mistakenly considered a year. Thus, the programmer must cast strings containing years to the appropriate type, as in Section 3.
- Pointer dereferencing and `struct` field-access require the type of the argument to be qualified as `$NONYEAR`. For example, Carillon assumes that if the programmer dereferences a `char *`, she is manipulating the string directly rather than through an abstraction. It is the responsibility of the programmer to inspect the code to verify it is safe and to insert explicit qualifier casts to bypass the type system, if necessary.
- The type of `&` expressions are qualified as `$NONYEAR`. Moreover, pointer types involved in arithmetic operations, such as addition and subtraction, are also qualified as `$NONYEAR`. Again, these qualifiers prevent explicit manipulation of dates.

Moreover, as we have seen in Section 3, Carillon gives warnings if the types of arguments passed for `...` in an application of a variable-length argument function contain qualifiers other than `$NONYEAR`. Such applications could potentially be unsafe.

Finally, many programs use standard library functions such as `strcmp`, `strcpy`, and `printf` to manipulate strings. Carillon needs to know the types of these functions in order to correctly analyze the program—specifically, Carillon needs to know what effects these functions have on strings. Thus, Carillon comes with a file of standard declarations for library functions (file `examples/prelude.i`). See Section 8.1 for more discussion.

5 Using the System

In this section, we describe in more detail how to use Carillon for analyzing industrial-sized programs (i.e., programs larger than the example program of Section 3.)

5.1 Modifying a Makefile

As mentioned earlier, Carillon parses only preprocessed C code. Large C programs are usually maintained and compiled using the program `make`, which reads a `Makefile` to determine what recompilations are necessary to compile and link the program. Here we describe how to modify the `Makefile` to also generate preprocessed C code.

We make use of two programs `remblanks` and `remquals`, which are included with Carillon. The `remblanks` program removes superfluous blank lines from a program. The `remquals` program removes all qualifiers and qualifier variables (i.e., tokens starting with `$`) from a program. Both programs read characters from `stdin` and output characters to `stdout`. The programs are located in the `bin` directory.

We use file names of the form *file*`.i` to denote preprocessed files. Here is a `make` rule for compiling a C file *file*`.c` into an object file *file*`.o` and in the process creating a preprocessed file *file*`.i`.

```
.c.o:
        $(CC) -E $< | remblanks > $*.ii
        remquals < $*.ii > $*.i
        $(CC) $(CFLAGS) -c -o $*.o $*.i
        mv -f $*.ii $*.i
```

The first line in this rule preprocesses the C file, removes superfluous blank lines using the `remblanks` program,[2] and stores the result in a temporary file *file*`.ii`. The second line uses `remquals` to remove all qualifiers and qualifier variables from the preprocessed code. The third line performs the actual compilation. The last line moves the preprocessed file (with qualifier annotations) to *file*`.i`.

Once a `Makefile` has been instrumented to create preprocessed files and preprocessed files have been generated for each C file in the program, Carillon can be used to analyze the program. Instructions on how to analyze a program with multiple files are given in Section 8.

5.2 Dealing with Error Messages

Carillon issues three kinds of error messages: parse errors, C type errors, and Y2K errors. Carillon issues parse errors in a buffer in Emacs. In the case that one or more parse errors are found, Carillon does not try to type check the program. The user must correct possible parse errors before the program can be analyzed properly by Carillon.

In traditional C compilers, C type checking is performed after parsing, thereby verifying that the program source is indeed a valid C program. Carillon is not as good at finding C type errors as a C compiler, mostly because the structure of the types that Carillon uses is simple (see Section 4.1). In general, before analyzing a program with Carillon, the program should be checked for possible parse and type errors with a C compiler.

The C type errors that Carillon does detect, include those that cause a mismatch between Carillon types (e.g., between `num` and `void`). Carillon also detects if a function is applied to fewer arguments than it specifies.

The most interesting kind of error messages are those caused by type qualifier mismatches. These kinds of errors indicate a potential Y2K error as illustrated in Section 3. To correct these kinds of errors it is essential that the programmer has a basic understanding of the Carillon type system (see Section 4). In Section 6, we shall see an example where a series of error messages are safely eliminated by bypassing the Carillon type system, through the use of explicit casts.

Carillon issues an error message if one of the following rules is violated:

> **Rule 1.** The set of function definitions for an identifier in a program must have identical types.

> **Rule 2.** A type inferred for a function definition must be identical to the first declaration of that function in each file that declares the function.

[2] We assume here that the `bin` directory is included in the users `PATH` environment-variable for accessing `remblanks` and `remquals` without specifying the paths.

The first rule allows for multiple definitions of functions, which provides support for the GNU `__inline__` extension. Carillon prints a warning message when a function is defined more than once.

Each of the rules can be violated either because of a type qualifier mismatch or because of a mismath in the structure of the types involved (e.g., a `num` type is matched against a function type.)

In Section 8, we refine the two rules to allow for declarations and definitions of polymorphic functions.

Carillon does not issue a warning if an identifier is declared more than once in a file, even if the identifier is declared with different types. For each file, Carillon uses the first declaration of an identifier for all its succeeding uses.

5.3 Cast Control and Warnings

Carillon propagates type information correctly only up to casting. Thus, it is important that all the casts in a program are safe. When a program has been analyzed, Carillon shows a list of casts involving qualifiers other than `$NONYEAR`; click on the `Cast Control` overlay in the `Carillon Results` buffer to see a list of overlays, each of which is linked to an unsafe cast in the program.

Carillon issues warnings if there are any implicit casts to or from a type containing qualifiers other than `$NONYEAR` (e.g., `$YY`, `$YYYY`). There are two places such implicit casts can occur. The first is when an argument is passed for ... in the application of a variable-length argument function. The second is when using `union`'s with types containing qualifiers other than `$NONYEAR`. In both cases, Carillon produces a list of warning messages. After a program has been analyzed, one can click on the overlays `Unsafe Var-Arg Apps` and `Unsafe Unions`—in the `Carillon Results` buffer—to see the warnings.

5.4 Year-Involved Functions

Another way to view where years are propagated in an analyzed program is to click on the `Year-Involved Functions` overlay in the `Carillon Results` buffer. Carillon then lists overlays pointing to the definitions of functions for which qualifiers other than `$NONYEAR` occur in their types.

5.5 The Configuration File

The Carillon type system uses a configuration file to define user specified type qualifiers. For the examples shown in this document, the following configuration file suffices (file `examples/config.d`):

```
$YYYY color "pam-color-4";
$YY color "pam-color-5";
$RCSYEAR color "pam-color-6";
```

This configuration file introduces the qualifiers `$YYYY`, `$YY`, and `$RCSYEAR`, and binds them to the colors `"pam-color-4"`, `"pam-color-5"`, and `"pam-color-6"`, respectively. The qualifier `$NONYEAR` is built-in and associated with the color `"pam-color-3"`. The colors are used for visualizing the identifiers in an analyzed program. An identifier whose type contains only one kind of qualifier is colored with its associated color. When two or more different qualifiers occur in the type of an identifier, then the color `"pam-color-2"` is used for the overlay. Finally, when no qualifier occurs in the type of an identifier then `"pam-color-1"` is used for the overlay. See Section 2.2 for information about modifying the mapping of the names `"pam-color-1"` thorough `"pam-color-8"` into actual colors in Emacs.

The possibility of extending the set of qualifiers is useful for analyzing programs that use many different representations of years, such as four digit years and *windowing* years (i.e., years represented by two digits, but offset by a number so that a fixed set of years before and after year 2000 are representable.)

6 A Second Example—RCS Years

We now present a more elaborate example extracted from the C sources of the Revision Control System (RCS) software package. RCS was originally written to work with only two-digit years but was then modified so that files created with RCS before year 2000 work correctly when used with RCS after year 2000. This new version of RCS (version 5.5 or later) has been successfully checked for Y2K errors with Carillon.

RCS uses several different internal representations of dates. Y2K errors may occur wherever string representations of dates are manipulated or transformed into other date representations. Because RCS initially worked with strings containing only two-digit years and because it is crucial that new versions of RCS are backward compatible, RCS gives meaning to strings with two- and four-digit years as follows:

- Years before 2000 can be represented using two digits or four digits.
- Years after 2000 must be represented using four digits.

So for example, the year 1999 can be represented both as the string `"99"` and as the string `"1999"`, whereas the year 2000 must be represented as the string `"2000"`. We associate this meaning of strings containing years with a new qualifier `$RCSYEAR` (this qualifier is already present in the configuration file `examples/config.d`.) For convenience, we extend the notion of `$RCSYEAR` strings and `$YYYY` strings to denote also strings containing dates, where the year part of the date is an `$RCSYEAR` string or a `$YYYY` string, respectively.

Consider the following example code (file `examples/rcs1.c`), which is extracted from the RCS sources and annotated with qualifiers (the code is also modified slightly for the presentation):

```
int printf(const char * $NONYEAR format, ...);
int sprintf(char * str, const char * format, ...);

char * $YYYY date2str(char * $RCSYEAR date, char * $NONYEAR datebuf) {
```

```
  char *p = date;
  while (*p++ != '.')
    ;
  sprintf(datebuf,
          "19%.*s/%.2s/%.2s" + (date[2]=='.' ?  0  :  2),
          (int)(p-date-1), date, p, p+3
          );
  return datebuf;
}
int main(void) {
  char *today = (char * $RCSYEAR)"99.05.12";
  char *nextyear = (char * $RCSYEAR)"2000.05.12";
  char *datebuf = "\0         ";
  printf("today is %s\n", date2str(today,datebuf));
  printf("nextyear is %s\n", date2str(nextyear,datebuf));
  return 1;
}
```

Here `main` formats and prints the dates 1999.05.12 and 2000.05.12, but internally, the date 1999.05.12 is represented as the string `"99.05.12"`. The `date2str` uses a buffer to reformat an `$RCSYEAR` date as a `$YYYY` date.

Although the code behaves as intended, it imposes several challenges to the Carillon type system. Carillon issues the following error messages when the program is analyzed:

```
/home/mael/Carillon/examples/rcs1.c:3.13-3.24
 Error during analysis of ``*p++!='.'''.
 The qualifier $NONYEAR does not match the qualifier $RCSYEAR.

/home/mael/Carillon/examples/rcs1.c:5.6-5.13
 Error during analysis of ``sprintf(datebuf, "19%.*s/%.2s/%.2s"+
   date[2]=='.' ? 0 : 2, (int ) p-date-1, date, p, p+3)''.
 The qualifier $NONYEAR does not match the qualifier $RCSYEAR.

/home/mael/Carillon/examples/rcs1.c:9.6-9.21
 Error during analysis of ``return datebuf;''.
 The qualifier $YYYY does not match the qualifier $NONYEAR.
```

The first two error messages are caused by the Carillon pointer-dereferencing type rule, which requires the type of the argument to a pointer-dereferencing construct to be qualified as `$NONYEAR`. Carillon issues the first error message because `p` has type `(char * $RCSYEAR)` because it is assigned to `date`, but `p` is dereferenced in line 3. Similarly, Carillon issues the second error message because `date` is dereferenced in line 5. The third error message is issued because `datebuf`, which has type `char * $NONYEAR`, is associated with the return type `char * $YYYY` of `date2str` in line 9.

Now, before we can safely bypass the type system and cast `datebuf` to type `char * $YYYY` in line 9, we must be sure that the body of `date2str` behaves as intended. It is up to the

programmer to convince herself that the code is correct. Here is a version of `date2str` with casts inserted to bypass the Carillon type system (file `examples/rcs2.c`):

```
char * $YYYY date2str(char * $RCSYEAR date, char * $NONYEAR datebuf) {
  char *p = (char * $NONYEAR)date;
  while (*p++ != '.')
    ;
  sprintf(datebuf,
          "19%.*s/%.2s/%.2s" + (((char * $NONYEAR)date)[2]=='.' ? 0 : 2),
          (int)(p-date-1), date, p, p+3
          );
  return (char * $YYYY)datebuf;
}
```

With these annotations, Carillon now issues three warnings, because arguments with qualifiers other than `$NONYEAR` in their types are passed for `...` in the application of `sprintf`. It is left as an exercise to the reader to eliminate these warnings (a solution is given in `examples/rcs3.c`), but do not forget to convince yourself that the casts you insert are safe. After eliminating the warnings, the inserted casts turn up in the `Cast Overview` (see Section 5.3).

7 Qualifier Polymorphism

Carillon makes it possible to avoid a range of annotations by allowing function identifiers to be associated with so-called polymorphic types. Polymorphism in Carillon makes it possible to give different qualified types to different uses of a function, in such a way that type information is still propagated safely.

The benefit provided by polymorphism is best illustrated with an example. Consider the following annotated declaration of the `strcpy` function from the C Standard Library:

```
char * $NONYEAR strcpy(char * $NONYEAR s1, char * $NONYEAR s2);
```

Recall that `strcpy` copies the string `s2` to the string `s1` and returns `s2` as a result. The qualifier annotations ensure that no year-qualified string is copied to a `$NONYEAR`-qualified string (or vice versa) without notifying the user of the problematic copying. Now, consider a program containing the two statements

```
strcpy(text, "The year is ");
strcpy(year, (char * $YYYY)"1999");
```

where `text` and `year` are declared with type `char *`. Although the first application of `strcpy` leads to no type error—provided the type of `text` is `char * $NONYEAR`—the second application of `strcpy` does lead to a type error. The type error requires the user to first refine the declaration of `year` to be of type `char * $YYYY`, and second, to cast the first and second arguments to `strcpy` to be of type `char * $NONYEAR`.

We can avoid this problem by introducing *qualifier polymorphism* into the Carillon type system. Intuitively, we want to allow `s1`, `s2`, and the result of `strcpy` to have any qualifiers in their types as long as all three have the same qualifiers. We achieve this with the following declaration (in `prelude.i`):

```
char * $_a strcpy(char * $_a s1, char * $_a s2);
```

Here `strcpy` is declared to be *polymorphic* in the qualifier variable `$_a`, meaning that if we replace `$_a` consistently with any qualifier, then we will have a valid type for `strcpy`. For example, we can replace `$_a` by the qualifier `$YYYY` to see that `strcpy` can have the type

```
char * $YYYY, char * $YYYY -> char * $YYYY
```

We can replace `$_a`by `$YY` to see that `strcpy` *also* has the type

```
char * $YY, char * $YY -> char * $YY
```

Thus, with this declaration it is possible to apply `strcpy` in different contexts with arguments of different qualified types. Consider again a program with the two statements

```
strcpy(text, "The year is ");
strcpy(year, (char * $YYYY)"1999");
```

where `text` and `year` are declared with type `char *`, but now in the context of the polymorphic declaration of `strcpy`. Although `strcpy` is applied to arguments of different qualified types, this time, the statements do not lead to a type error. The first application constrains `text` to be qualified as `$NONYEAR`, because Carillon gives the type `char * $NONYEAR` to the string literal in the first application of `strcpy`. Moreover, the second application of `strcpy` constrains `year` to be qualified as `$YYYY`. Notice that the type of `strcpy` is not polymorphic in the underlying C type; Carillon allows types to be polymorphic only in qualifiers.

In the following, we use *qv* to range over qualifier variables. *Polymorphic types* in Carillon, which are also called *type schemes*, are given by the following grammar:

scheme	::=	`All(` qv^* `).` *type*	Polymorphic type scheme
	|	*type*	Non-polymorphic type scheme

In a type scheme *scheme* of the form `All(` $qv_1, \cdots, qv_n$ `).` *type*, the qualifier variables $qv_1, \cdots, qv_n$ are called the *generalized* qualifier variables of the type scheme, and *type* is called the *body* of the type scheme. We require that all the qualifier variables that appear in the body of a type scheme are generalized. Thus, in the declaration of `strcpy` above the qualifier variable `$_a` is implicitly generalized. We say that a type *type*′ is an *instance* of the type scheme *scheme* if there exists a mapping S (called a *substitution*) from the qualifier variables $qv_1, \cdots, qv_n$ to types $type_1, \cdots, type_n$, such that $type' = S(type)$. Here the notation $S(type)$ means the type *type*, with each qualifier variable *qv* in the domain of S being substituted with $S(qv)$. The instance relation extends to type schemes as follows. A type scheme `All(` $qv_1, \cdots, qv_n$ `).` *type* is an instance of another type scheme *scheme* iff *type* is an instance of *scheme*. Two type schemes are *equal* if they can be made identical by systematic renaming of generalized qualifier variables.

Carillon allows functions to be both defined and declared with polymorphic types. To continue our example, here is an implementation of `strcpy` (file `examples/strcpy.c`):

```
char * $_a strcpy(char * $_a s1, const char * $_a s2) {
   char * p = (char * $NONYEAR)s1;
   for ( ; *p++ = *((char * $NONYEAR)s2)++ ; ) ;
   return s1;
}
```

Notice that we have cast the uses of `s2` and the first use of `s1` to be qualified as `$NONYEAR`, which allows the type of `s1` and `s2` to be `char * $_a`.

Carillon finds that the definition we have given for `strcpy` is consistent with the polymorphic declaration of `strcpy`. In fact, Carillon will complain if the type scheme given by a previous declaration of a function is not an instance of the type scheme inferred for the definition of the function.

Moreover, when a type scheme is formed for a function definition, Carillon requires that the type scheme is *closed*, meaning that all qualifier variables appearing in the types of the arguments and result of a function definition are generalized. For the system to be sound, Carillon requires that none of the qualifier variables occurs in the type of any identifier in a scope outside of the function definition. This restriction makes the following code erroneous (file `examples/wrong.c`):

```
char * s;
void wrong(char * $_q a) {
   s = a;
   return;
}
s = (char * $YYYY) "1999";
```

Carillon complains with the error message

```
/home/mael/Carillon/examples/wrong.c:2.7-2.12
 Failed to close function type for ‘wrong’. Type variable ‘$_q’ could
 not be generalized.
```

In the example, the identifier `s` is constrained by the assignment in `wrong` to be of type `char * $_q`, because `s` is given this type, and clearly we cannot generalize `s`'s type after analyzing `wrong` because we have not yet discovered that `s` must be of type `char * $YYYY`.

8 Multiple Files

Carillon can analyze one file at a time or multiple files at once. To make Carillon analyze multiple files at once, enter a directory path when asked for a file to analyze. Carillon then analyzes all `.i` files in the directory (or `.c` files, if no `.i` files are present.) By analyzing multiple files at once, Carillon has a better chance of finding type inconsistencies in the program. In particular, the types (or type schemes) that Carillon infers for the definitions in one file are used for the uses of these identifiers in other files (instead of the perhaps less descriptive declarations of these identifiers.)

The two basic type-consistency rules that are enforced across files were given in Section 5.2. Here we refine the rules so as to allow declarations and definitions of functions with polymorphic types. The first rule is refined to hold for type schemes:

> **Rule 1′** The set of function definitions for an identifier in a program must have identical type schemes.

Notice that this rule holds across files: It is an error if a function is defined in different files in a program with different type schemes. Recall, that for each file, all declarations for an identifier except the first are discarded. The second rule is refined to the following:

> **Rule 2′** If an identifier *id* is defined in some file with type scheme *scheme*, then for each file that declares *id*, the type scheme provided by the first declaration of *id* in the file must be an instance of *scheme*.

We illustrate the second rule with an example. Assume a program with the two files `dec.c` and `def.c` (directory `examples/defdec/`):

```
dec.c        char * $_q1 first(char * $_q1 a, char * $_q2 b);

def.c        char * $_q1 first(char * $_q1 a, char * $NONYEAR b) {
                return a;
             }
```

Here the second rule is violated because the type scheme provided by the declaration in `dec.c` is not an instance of the type scheme provided by the definition in `def.c`. Carillon issues the following error message:

```
Error occurred in declaration of ‘first’. Identifier ‘first’ is
defined with type
   All($_q1).($_q1 ptr(num), $NONYEAR ptr(num)) -> $_q1 ptr(num)
which is inconsistent with the type
   All($_q2,$_q1).($_q1 ptr(num), $_q2 ptr(num)) -> $_q1 ptr(num)
with which it was declared.
```

The two rules have the important property that whether the analysis succeeds is independent of the order in which the files in a program are analyzed.

With Carillon, types containing `struct`s must match across files. This property is essential for the safety of the type system. Consider the following example consisting of the two files `struct1.c` and `struct2.c` (directory `examples/struct/`):

```
struct1.c    struct date {char * y;};
             struct date d = {(char * $YY)"99"};

struct2.c    int printf(const char * $NONYEAR format, ...);
             struct d {char * y;};
             extern struct d d;
```

```
int main(void) {
   printf(d.y);
   return 0;
}
```

Because `printf` requires a `$NONYEAR` qualified string to be passed for its first argument and because the identifier `d` is defined with a `$YY` qualified string element, Carillon issues the following error message:

```
/home/mael/BANE/CQual/examples/struct/struct2.c:4.3-4.9
 Error during analysis of ``printf(d.y)''.
 The qualifier $NONYEAR does not match the qualifier $YY.
```

Thus, Carillon detects that the program is in danger of printing a two digit year.

8.1 Libraries and the Prelude File

Because Carillon works only on preprocessed C code, declarations for all library functions that are used in a file are already present in the file that uses the identifiers. However, because the code for such library functions is often not available, it is sometimes necessary to provide declarations that annotate the types of library functions with appropriate qualifiers. In particular, string manipulation functions, such as those found in `string.h`, must be restricted so their arguments have `$NONYEAR` qualified types. One example of such a function is the `atoi` function from the C Standard Library, which in Carillon is declared by

```
int atoi(const char $NONYEAR s);
```

This declaration is given in the default prelude file `examples/prelude.i`, which is analyzed before any other program file. The declaration of `atoi` in the prelude file does not conflict with the declaration of `atoi` in the library, which is declared identically but without the `$NONYEAR` qualifier. (Recall that any missing qualifiers are assumed to be qualifier variables.) Another class of identifiers that are declared in the prelude file are those library functions that are polymorphic in their type qualifiers. One example of such a function is `strcpy` from Section 7.

For each identifier declared in the prelude file with type scheme *scheme*, Carillon requires that the first declaration of this identifier in a file provides a type scheme that is an instance of *scheme*. The identifiers declared in the prelude file cannot immediately be used by another file without first being declared in this file, but when declared, the type scheme provided by the declaration in the prelude file is used. See Section 3 for an example involving the `printf` function.

For a particular application, it may be necessary to extend the prelude file to describe more library functions. It is also possible to choose between different prelude files by modifying the settings in your personal `.emacs` file (see Section 2.1.)

9 The Emacs Interface

The bulk of Carillon is a program, written in Standard ML, which analyzes C files and communicates the results to Emacs. Emacs then displays the result of the analysis to the user via Program Analysis Mode (PAM). PAM is also the name of the software that implements the communication layer between the Standard ML program and Emacs.

Once Carillon has analyzed the files of a program, the user can browse the analysis results using the mouse or the keyboard. Here is a list of commands that are supported by PAM:

`C-c C-l` selects the overlay pointed at by the cursor (same as selecting the overlay with the middle mouse-button.)

`C-c C-f` analyzes a file or a directory.

`C-c C-r` exits PAM and kills all PAM buffers.

10 The Authors

If you have any questions or comments related to Carillon, please do not hesitate to contact the authors. You can use the email address `bane-software@cs.berkeley.edu`.

We would like to thank Henning Niss and Chris Harrelson for providing the current version of the Program Analysis Mode (PAM).

11 Conclusion

Several other tools are available for finding Y2K problems in COBOL programs. One example is Hafnium's commercial product AnnoDomini, which is a tool for finding Y2K errors in IBM OS/VS COBOL programs. Like Carillon, AnnoDomini is based on a type system for detecting inconsistent uses of years [EHM+99]. We know of no systems other than Carillon for finding Y2K problems in C programs.

Two related tools are Lackwit [OJ97] and LCLint [EGHT94, Eva96]. Based on ML type inference, Lackwit can be used to detect abstraction violations in C programs. LCLint is a tool that uses, among other techniques, an extended set of C type qualifiers to find bugs in C programs. Carillon integrates the use of qualifiers and polymorphism to a degree that makes it useful to analyze even very large programs for Y2K readiness. Carillon has been used effectively to locate a Y2K bug in CVS (Concurrent Version System) version 1.9, which is about 57,000 lines of C (132,000 lines preprocessed).

In this document, we have presented Carillon, a system to find Y2K problems in C programs. The difficulties of establishing the Y2K readiness of software are largely caused by programs that break type abstraction barriers. An automatic tool, combined with appropriate information from a programmer (in form of qualifier annotations), is highly desirable for finding Y2K errors in programs and for establishing that such errors do not occur. Carillon provides just such a tool.

There are other examples of conversion problems where Carillon can help to detect abstraction violations, including conversion of programs to the use of unicode characters and conversion of programs to use the new Euro currency instead of native European currencies.

References

[EGHT94] David Evans, John Guttag, Jim Horning, and Yang Meng Tan. Lclint: A tool for using specifications to check code. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (SFSE'94)*, December 1994.

[EHM+99] Peter Harry Eidorff, Fritz Henglein, Christian Mossin, Henning Niss, Morten Heine Sørensen, and Mads Tofte. Annodomini: From type theory to year 2000 conversion tool. In *26th Symposium on the Principles of Programming Languages (POPL'99)*, January 1999.

[Eva96] David Evans. Static detection of dynamic memory errors. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'96), Philadelphia, PA*, May 1996.

[FFA99] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'99)*, pages 192–203, May 1999.

[OJ97] Robert O'Callahan and Daniel Jackson. Lackwit: A program understanding tool based on type inference. In *International Conference on Software Engineering (ICSE'97)*, May 1997.

A Limitations

In this appendix, we list some of the limitations of Carillon:

- Initializers are required to match the structure of the type of the variable being initialized. In most cases it is straightforward to modify the program text to meet this requirement. For example, the following code

  ```
  struct { char *s; int x } foo f = {"abc", 3, "def", 4};
  ```

  should be rewritten as

  ```
  struct { char *s; int x } foo f = {{"abc", 3}, {"def", 4}};
  ```

- Some GNU C extensions are supported, but not all; for instance, the `__typeof` operator is not supported.

- Carillon cannot parse functions returning function pointers. To get around this problem, one can use a `typedef` to define an identifier for the return type. This identifier can then be used for the return type of the function.

B Copyright Notice