# CQUAL User's Guide
# Version 0.991

Jeffrey S. Foster (University of Maryland, College Park)
et al (University of California, Berkeley)
cqual@cs.umd.edu

March 27, 2007

Many people have contributed to the development of CQUAL. CQUAL uses the front-end from David Gay's region compiler [GA01] to parse C programs. Martin Elsman and Alex Aiken worked on CARILLON [EFA99], an earlier version of this system written in SML/NJ, which used the type qualifier system of [FFA99] to find Y2K bugs in C programs. Umesh Shankar, Kunal Talwar, and David Wagner used the system to find format-string bugs in C programs [STFW01], in the process making a number of important usability improvements. Tachio Teruachi and Alex Aiken helped develop the flow-sensitive portion of CQUAL. John Kodumal and Rob Johnson implemented polymorphic recursive qualifier inference for CQUAL, and Rob also enhanced CQUAL for user/kernel pointer checking, as well as making numerous other improvements. Portions of this document are taken from [EFA99] and [STFW01].

**Warning:** See Appendix A for known limitations and bugs.

# Contents


**1 Introduction** **3**
- 1.1 cqual and PAM Installation . . . 3
- 1.2 A Small Example . . . 4
- 1.3 Running cqual . . . 4
- 1.4 Another Example: Structural Qualifiers . . . 6
- 1.5 A Flow-Sensitive Example . . . 7
- 1.6 *l*-values and *r*-values . . . 7

**2 Type Qualifiers** **8**
- 2.1 Qualifiers and Subtyping . . . 8
- 2.2 Qualified Types . . . 9
- 2.3 Structural Qualifiers . . . 10
- 2.4 Qualifier Inference . . . 10
- 2.5 Flow-Sensitive Type Qualifiers . . . 11
  - 2.5.1 Aliasing . . . 11
  - 2.5.2 Restrict . . . 12
  - 2.5.3 Confine . . . 12
- 2.6 Browsing Qualifier Inference Results with PAM . . . 13

**3 Applying Type Qualifiers to C** **13**
- 3.1 Names . . . 13
- 3.2 Source Code Considerations . . . 14
  - 3.2.1 Multiple Files . . . 14
  - 3.2.2 Pre-Processed Source . . . 14
  - 3.2.3 Flow-Sensitivity . . . 15
  - 3.2.4 Type Casts . . . 15
  - 3.2.5 Structures . . . 16
  - 3.2.6 Restrict . . . 17
- 3.3 Partial Order Configuration File . . . 17
- 3.4 Prelude Files . . . 19
- 3.5 Qualifier Polymorphism . . . 19
- 3.6 Deep Subtyping with const . . . 21
- 3.7 Functions with Variable Numbers of Arguments . . . 21
- 3.8 Old-Style Functions . . . 22
- 3.9 Operators . . . 22
- 3.10 iquals . . . 22

**4 PAM Mode** **22**
- 4.1 The Interface . . . 22
- 4.2 Changing the Analysis . . . 23
- 4.3 Customizing Colors . . . 23

**5 Reference** **23**
- 5.1 cqual . . . 23
- 5.2 iquals . . . 24
- 5.3 gcqual . . . 25
- 5.4 Partial Order Configuration File . . . 25

**A Limitations and Bugs** **27**

**B Copyright** **27**

# 1 Introduction

CQUAL is a type-based analysis tool for finding bugs in C programs. CQUAL extends the type system of C with extra user-defined *type qualifiers*. The programmer annotates their program in a few places, and CQUAL performs *qualifier inference* to check whether the annotations are correct. CQUAL presents the analysis results either on the command line or in an interactive EMACS buffer.

Earlier versions of CQUAL written in SML/NJ have been used to perform `const`-inference [FFA99] and to find Y2K bugs [EFA99]. The current version of CQUAL has been used to detect potential format-string vulnerabilities [STFW01] and to find locking bugs in the Linux kernel [FTA02]. The latest version of CQUAL adds support for polymorphic recursive qualifier inference and gated qualifier edges, which are used for structural qualifier constraints. This release also includes direct support for checking for user/kernel pointer errors.

## 1.1 cqual and PAM Installation

The latest version of CQUAL can be found at

```
http://cqual.sourceforge.net
```

To unpack CQUAL, execute the following commands:

```
gunzip cqual-0.991.tar.gz
tar xf cqual-0.991.tar
```

CQUAL will be unpacked into a directory `cqual-0.991`, which contains, among other things,

| | |
|---|---|
| `COPYRIGHT` | The copyright notice |
| `KERNEL-QUICKSTART` | Guide to using cqual to find user/kernel errors |
| `bin` | Some utilities |
| `config` | Sample CQUAL configuration files |
| `doc` | Documentation (contains this file) |
| `src` | Source code for CQUAL |
| `PAM-3` | The latest version of PAM |

The latest version of PAM can also be downloaded separately from

```
http://www.cs.berkeley.edu/~chrishtr/pam
```

.

To build CQUAL and PAM, simply `cd` into the directory, run `configure` and then run `make`:

```
cd cqual-0.991
./configure
make
```

If all goes well, the makefile will build two executables in the `src` directory: `cqual`, the type qualifier inference system, and `iquals`, a small tool for experimenting with qualifier constraints. The makefile will also build PAM and append some commands to your `.emacs` file so that you can run CQUAL using PAM.

While you don't need to run CQUAL in PAM mode, the analysis results are much easier to understand if you do. If you wish to run CQUAL solely from the command line, you will be able to see some but not all of the information you could in PAM mode. In particular, rather than being able to browse through the program on qualifier paths, you will be presented with paths corresponding to just the errors.

## 1.2 A Small Example

In this section we present a small example showing how to use CQUAL to find a potential format-string vulnerability in a C program. Consider the following program, which is included in the distribution as `examples/taint0.c`:

```
char *getenv(const char *name);
int printf(const char *fmt, ...);

int main(void)
{
    char *s, *t;
    s = getenv("LD_LIBRARY_PATH");
    t = s;
    printf(t);
}
```

This program reads the value of `LD_LIBRARY_PATH` from the environment and passes it to `printf` as a format string. If an untrusted user can control the environment in which this program is run, then this program may have a format-string vulnerability. For example, if the user sets `LD_LIBRARY_PATH` to a long sequence of `%s`'s, the program will likely seg fault.

By default CQUAL assumes nothing about the behavior of your program.[1] In order to start checking for bugs, we need to annotate the program with extra *type qualifiers*. For this example we will use two qualifiers. We will annotate untrusted strings as `$tainted`, and we will require that `printf` take `$untainted` data:

```
$tainted char *getenv(const char *name);
int printf($untainted const char *fmt, ...);

int main(void)
{
    char *s, *t;
    s = getenv("LD_LIBRARY_PATH");
    t = s;
    printf(t);
}
```

In CQUAL all user-defined qualifiers, which we will refer to *constant qualifiers* or *partial order elements*, begin with dollar signs. Notice that we only need to annotate `getenv` and `printf` with type qualifiers. For this example CQUAL will infer that `s` and `t` must also be `$tainted`, and hence will signal a type error: `$tainted` data is being passed to `printf`, which requires `$untainted` data. The presence of a type error indicates a potential format-string vulnerability.

## 1.3 Running cqual

Assuming that CQUAL is already installed as described above, you can run CQUAL on this example program to see what happens. From within EMACS type `M-x cqual` and press return. Enter the file name `examples/taint1.c` (assuming you are in the top-level `cqual` directory) and press return.

CQUAL analyzes the file and brings up a window listing the input files and the analysis results. In this case, CQUAL complains

```
/home/rtjohnso/projects/cquals/fields2/cqual/examples/taint1.c:7
incompatible types in assignment
```

[1] CQUAL comes with some default configuration files for checking for format-string vulnerabilities and user/kernel errors; more on this below.

The user interface used to display the analysis results is Program Analysis Mode (PAM), a generic interface for marking up programs in emacs [PAM]. Middle-clicking on a hyperlink with the mouse or moving the cursor over a hyperlink and pressing `C-c C-l` will follow that link. Error messages are linked to the position in the file where the error was generated.

If you middle-click on the error message link you will see a marked-up display of `taint1.c`. Identifiers are colored according to their inferred qualifiers. In the default configuration file, `$tainted` identifiers are colored red, `$untainted` identifiers are colored green, and identifiers that may contribute to a type error are colored purple.

Each marked-up identifier is also a hyperlink. Middle-clicking on an identifier will show you the type of the identifier, fully annotated with qualifiers. For example, middle-clicking on `t` should bring up a window showing

```
t:  &t ptr (t ptr (*t char))
```

The name of the identifier is shown to the left of the colon, and its inferred type is are shown to the right of the colon.

Here `t` has the type pointer to pointer to character. (We will explain the extra level of `ptr` in Section 1.6.) Notice that CQUAL writes types from left-to-right using `ptr` as a type constructor.

The three hyperlinked names in the type are *qualifier variables* (see Section 2). In this case the qualifier variable $*t$ (throughout this document we italicize qualifier variables) is colored purple because it has been inferred to be both `$tainted` and `$untainted`, an error.

Middle-clicking on a qualifier variable will show you the inferred value of the qualifier variable and the shortest path on which it was inferred to have its value. For example, if you click on $*t$, you should see the following result:

```
*t:  $tainted $untainted

$tainted <= *getenv_ret
         <= *s
         <= *t
         <= *printf_arg1
         <= $untainted
```

The first line tells us that $*t$ is both `$tainted` and `$untainted`, an error. The remaining lines show us an erroneous path. We see that $*t$ was tainted from $*s$, which was tainted from the return type of `getenv`. We also see that the error arises because $*t$ taints the parameter to `printf`, which must be untainted.

Middle-clicking on a `<=` will jump to the source location where that constraint was generated. Middle-clicking an a qualifier we compute the shortest path by which that qualifier was inferred to have its value. And shift-middle-clicking on a qualifier will jump to the source location where the identifier corresponding to that qualifier was defined.

You can also run CQUAL on the command line. If you do so with the appropriate configuration arguments then CQUAL will generate the same error messages, but you will be unable to interactively explore the analysis results:

```
bash-2.05a$ src/cqual -config config/lattice examples/taint1.c
Analyzing examples/taint1.c
examples/taint1.c:1 ``getenv'' used but not defined
examples/taint1.c:2 ``printf'' used but not defined
examples/taint1.c:7 incompatible types in assignment
*s: $tainted $untainted
examples/taint1.c:1      $tainted <= *getenv_ret
examples/taint1.c:7               <= *s
examples/taint1.c:8               <= *t
```

```
  examples/taint1.c:9              <= *printf_arg1
  examples/taint1.c:2              <= $untainted
```

Cqual also lists the globals that are used but not defined; see Section 3.2.1.

CQUAL comes with a standard *prelude file* that contains declarations of standard-library functions that have been annotated with `$tainted` and `$untainted`. See Section 3.4 for a discussion of prelude files, and Section 4.2 for instructions on how to invoke CQUAL in PAM mode with the standard prelude file.

## 1.4 Another Example: Structural Qualifiers

In the previous example, we saw how format-string vulnerabilities can occur when a program uses untrusted data in certain positions, namely as format strings. Strings are particularly simple, because they're flat sequences of characters. In programs that have trusted and untrusted data structures and pointers, the rules are more complicated, and we need to extend our type qualifiers with additional *structural* constraints. As an example of such a system, consider the Linux operating system kernel, which copies data structures between user space (which we will annotate with `$user` and consider untrusted) and kernel space (which we will annotate with `$kernel` and consider trusted).

The following program (available as `examples/user0.c`) has a user/kernel pointer bug, meaning that data copied from user space is improperly trusted, and hence a malicious local user could breach security:

```
unsigned long copy_from_user(void $user * $kernel to, void * $user from, unsigned long n);
$$a _op_deref ($$a *$kernel x);

struct msg {
  char *buf;
};

void dev_ioctl(long arg)
{
  struct msg m;
  char c;

  copy_from_user(&m, (void*)arg, sizeof (m));
  c = m.buf[0];
}
```

After the call to `copy_from_user`, the contents of `m` are under user control. Thus `m.buf` is under user control, and hence should be sanity checked before being dereferenced. The statement "`c = m.buf[0];`" unsafely dereferences `m.buf`, and hence is an error. The annotations capture these rules:

- The annotation "`void $user * $kernel to`" indicates that the first argument to `copy_from_user` must be a kernel pointer, but that its contents are under user control.
- The annotation "`$$a _op_deref ($$a *$kernel x);`" declares that only kernel pointers can be dereferenced. For more on this annotation, see Section 3.9.

Moreover, once we infer that `m` has the `$user` qualifier, we also infer, via structural constraints, that its `m.buf` must also be annotated with `$user`. For `$user` and `$kernel`, CQUAL enforces *structural* rules like, "If a structure is `$user`, then so are all of its fields," and "A `$user` pointer can only point to `$user` data." For more on structural constraints see Section 2.3.

For more on checking for user/kernel pointer errors in the Linux kernel, see `KERNEL-QUICKSTART`.

## 1.5 A Flow-Sensitive Example

The qualifiers `$tainted` and `$untainted` are *flow-insensitive*, meaning that a variable's taintedness does not change during program execution. I.e., if `x` is inferred to be `$tainted`, then it is `$tainted` everywhere.

Sometimes this flow-insensitive restriction makes it difficult to apply type qualifiers to certain checking problems. For example, if we want to use qualifiers to keep track of state changes, then we need *flow-sensitivity*, i.e., we need qualifiers that can change as the state changes. For example, consider the following program, which can be found in `examples/lock.c`:

```
typedef int lock_t;

lock_t lock;

int main(void)
{
  lock = ($unlocked lock_t) 0;
  lock;
  lock = ($locked lock_t) 1;
  lock;
  lock = ($unlocked lock_t) 0;
  lock;
}
```

In this case, we want to use qualifiers `$locked` and `$unlocked` to keep track of whether this thread last left `lock` in the locked or unlocked state.

In order to analyze this example, we need to tell CQUAL that `$locked` and `$unlocked` should be modeled flow-sensitively. That's already taken care of in the default configuration files, so to try out this example type `M-x cqual` within EMACS, press return, and then enter the file name `examples/lock.c` and press return again.

As before, CQUAL analyzes the program. This time there are no type errors. If you click on the file name, CQUAL will display the source code colored according to the inferred qualifiers. In this case, CQUAL colors `lock` green wherever it is unlocked, and red wherever it is locked.

If you click on the various occurrences of `lock`, you can see its type and its qualifiers. Notice that name of the qualifier `lock` points to changes after an assignment. Initially it is *lock*, then it is *lock*@0, and so on.

**Warning:** This version of cqual has been enhanced with support for polymorphic recursion, gated qualifier edges for structural constraints, and better handling of type casts. These features, however, are not supported by the flow-sensitive type qualifier inference system, and as a result flow-sensitive CQUAL may fail on some programs. For the time being, we recommend using version 0.98 of CQUAL for flow-sensitive analysis.

## 1.6 *l*-values and *r*-values

In C there is an important distinction between *l*-values, which correspond to memory locations, and *r*-values, which are ordinary values like integers. In the C type system, *l*-values and *r*-values are given the same type. For example, consider the following code:

```
int x;
x = ...;
...  = x;
```

The first line declares that `x` is a location containing an integer. On the second line `x` is used as an *l*-value: it appears on the left-hand side of an assignment, meaning that the location corresponding to `x` should be updated. On the third line `x` is used as an *r*-value. Here when we use `x` as an *r*-value we are not referring

to the location `x`, but to `x`'s contents. In the C type system, `x` is given the type `int` in both places, and the syntax distinguishes integers that are *l*-values from integers that are *r*-values.

CQUAL uses a slightly different approach in which the types distinguish *l*-values and *r*-values. In CQUAL, `x` is given the type `ptr(int)`, meaning that the name `x` is a location containing an integer. When `x` is used as an *l*-value its type stays the same—in CQUAL, the left-hand side of an assignment is always a `ptr` type. When `x` is used as an *r*-value the outermost `ptr` is removed, i.e., `x` as an *r*-value has the type `int`. CQUAL is implemented in this way both because it makes the implementation cleaner in a number of ways and because it makes `const` easier to understand [FFA99].

In more concrete terms, if you click on an identifier `a` that can be used as an *l*-value you will see `a`'s type as an *l*-value, i.e., with an extra `ptr` at the top-level. For most purposes you can safely ignore this extra level of indirection.

# 2 Type Qualifiers

CQUAL is a type-based analysis tool. As described above, to use CQUAL the programmer annotates their program with extra type qualifiers. CQUAL type checks the program and warns the programmer about any inconsistent type qualifier annotations, which indicate potential bugs.

In the rest of this section we discuss what type qualifiers are and how CQUAL checks for inconsistent qualifier annotations. Section 3 describes how CQUAL applies these ideas to C.

## 2.1 Qualifiers and Subtyping

CQUAL extends the type system of C to work over *qualified types*, which are the combination of some number of type qualifiers with a standard C type. We allow type qualifiers to appear on every level of a type. Here are some examples of qualified types:

| | |
|---|---|
| `int` | Integer |
| `$locked lock_t` | Acquired lock |
| `ptr($untainted char)` | Pointer to untainted character |
| `$user ptr(char)` | User-level pointer to character |

In general, the rules for checking that type qualifiers are valid can be arbitrary, and indeed, the source code of CQUAL can be modified to support qualifiers with arbitrary meanings. The key insight behind CQUAL, however, is that many kinds of type qualifiers naturally induce a *subtyping* relationship on qualified types. The notion of subtyping most commonly appears in object-oriented programming. In Java, for example, if $B$ is a subclass of $A$ (which we will write $B < A$), then an object of class $B$ can be used wherever an object of class $A$ is expected.

For example, consider the following program, which uses the `$tainted` and `$untainted` qualifiers introduced above:

```
void f($tainted int);
$untainted int a;
f(a);
```

In this program, `f`, which expects tainted data, is passed untainted data. This program should type check. Intuitively, if a function can accept tainted data (presumably by doing more checks on its input), then it can certainly accept untainted data.

Now consider another program:

```
void g($untainted int);
$tainted int b;
g(b);
```

In this program, `g` is declared to take an `$untainted int` as input. Then `g` is called with a `$tainted int` as a parameter. This program should fail to type check, since tainted data is being passed to a function that expects untainted data.

Putting these two examples together, we have the following subtyping relation:

$$\texttt{\$untainted int} < \texttt{\$tainted int}$$

As in object-oriented programming, if $T_1 \leq T_2$ (read $T_1$ is a subtype of $T_2$), then $T_1$ can be used wherever $T_2$ is expected, but not vice-versa. We write $T_1 < T_2$ if $T_1 \leq T_2$ and $T_1 \neq T_2$.

On the other hand, consider `$locked` and `$unlocked`. It is an error for a lock to be in both the `$locked` and `$unlocked` state, so these qualifiers are in the discrete partial order: Neither `$locked` $\not<$ `$unlocked` nor `$unlocked` $\not<$ `$locked`. (Alternately, we could add a third qualifiers $\top$ and have `$locked` $< \top$ and `$unlocked` $< \top$.)

## 2.2 Qualified Types

CQUAL needs to know not only how integer types with qualifiers relate but also how qualifiers affect pointer types, pointer-to-pointer types, function types, and so on. Fortunately, well-known results on subtyping tell us how to extend the subtyping on integers to other data types.

The programmer supplies CQUAL with a configuration file describing a partial order of type qualifiers (see Section 3.3 for the file format). Right now cqual supports any partial order that is a lattice (a lattice is a partial order where for each pair of elements $x$ and $y$, the least upper bound and greatest lower bound of $x$ and $y$ both always exist. For example, the qualifiers `$tainted` and `$untainted` with the partial order `$untainted` $<$ `$tainted` form a lattice.) CQUAL also supports the discrete partial orders, and any of the three-point partial orders. Other partial orders may or may not work correctly.

Given the partial order configuration file, CQUAL extends the partial order on qualifiers to a subtyping relation on qualified types. We have already seen one of the subtyping rules:

$$\frac{q_1 \leq q_2}{q_1\ \texttt{int} \leq q_2\ \texttt{int}}$$

This is a natural-deduction style inference rule, read as follows: If $q_1 \leq q_2$ in the partial order ($q_1$ and $q_2$ are qualifiers), then $q_1$ `int` is a subtype of $q_2$ `int` (note the overloading of $\leq$). For our example, it means that `$untainted int` $\leq$ `$tainted int`. The same kind of rule applies to any primitive type (`char`, `double`, etc.).

For pointer types, we need to be a little careful. Naively, we might expect to use the following rule for pointers:

$$\frac{q_1 \leq q_2 \qquad \tau_1 \leq \tau_2}{q_1\ \texttt{ptr}(\tau_1) \leq q_2\ \texttt{ptr}(\tau_2)}\ \text{(Wrong)}$$

Here the type $q_1$ `ptr`$(\tau_1)$ is a pointer to type $\tau_1$, and the pointer is qualified with $q_1$. Unfortunately, this turns out to be unsound, as illustrated by the following code fragment:

```
tainted char *t;
untainted char *u;

t = u;                 /* Allowed by (Wrong) */
*t = <tainted data>;   /* tainted data written into untainted *u */
```

According to (Wrong), the first assignment `t = u` typechecks, because `ptr($untainted char)` is a subtype of `ptr($tainted char)`. But then after the assignment `*t` is an alias of `*u`, yet they have different types. Therefore we can store `$tainted` data into `*u` by going through `*t`, even though `*u` is supposed to be untainted.

This is a well-known problem, and the standard solution, which is followed by CQUAL, is to use the following rule:

$$\frac{q_1 \leq q_2 \qquad \tau_1 = \tau_2}{q_1\ \texttt{ptr}(\tau_1) \leq q_2\ \texttt{ptr}(\tau_2)}$$

Here we require $\tau_1 = \tau_2$, which intuitively means that any two objects that may be aliased must be given exactly the same type. In particular, if $\tau_1$ and $\tau_2$ are decorated with qualifiers, the qualifiers must themselves match exactly, too. This equality, while sound, is sometimes too conservative in practice. Section 3.6 describes how `const` can be used to weaken the equality to an inequality.

For function types, we use the following standard rule:

$$\frac{q \le q' \qquad \tau_1' \le \tau_1 \quad \cdots \quad \tau_n' \le \tau_n \qquad \tau \le \tau'}{q\ \texttt{fun}\ (\tau_1, \ldots, \tau_n)\ \to\ \tau \le q'\ \texttt{fun}\ (\tau_1', \ldots, \tau_n')\ \to\ \tau'}$$

Here the type $q\ \texttt{fun}\ (\tau_1, \ldots, \tau_n)\ \to\ \tau$ is a function, qualified by $q$, with argument types $\tau_1$ through $\tau_n$ and result type $\tau$.

## 2.3 Structural Qualifiers

In Section 1.4, we saw how some qualifiers come with well-formedness conditions on the types they decorate. We write $\vdash_{wf} \tau$ to mean that type $\tau$ is well-formed; CQUAL requires that all types in the program be well-formed. Each qualifier partial order comes with a set of well-formedness conditions. For example, a partial order may specify that its qualifiers flow from outer to inner pointer constructors or vice-versa. Formally, CQUAL enforces the rule

$$\frac{\vdash_{wf} \tau \qquad \begin{cases} c \le q \to c \le q' & \text{if } c\text{'s p.o. flows down pointers} \\ q \le c \to q' \le c & \text{if } c\text{'s p.o. flows down pointers} \\ c \le q' \to c \le q & \text{if } c\text{'s p.o. flows up pointers} \\ q' \le c \to q \le c & \text{if } c\text{'s p.o. flows up pointers} \end{cases}}{\vdash_{wf} q\ \texttt{ptr(}q'\ \tau\texttt{)}}$$

This rule states that a pointer-to-pointer-to-$\tau$ type is well-formed if type $\tau$ is well-formed. Additionally, for any qualifier $c$ of a partial order whose qualifiers flow "down" pointers, constraints on the outer pointer type propagate to the inner pointer type, and similarly for the case when qualifiers flow "up" pointers. For example, the `$user` qualifier from Section 1.4 flows down pointers.

CQUAL also provides support for two other well-formedness conditions. In a similar manner as above, the user may specify that qualifiers propagate from pointers-to-aggregates (structures and unions) to the corresponding pointers-to-fields or vice-versa, and separately, the user may specify that qualifiers propagate from aggregates (structures and unions) to their fields or vice-versa.

## 2.4 Qualifier Inference

Given the partial order configuration file, CQUAL extends the qualifier partial order to a subtyping relation among qualified types as described above. The next problem is to determine whether a program is type correct or not, i.e., whether the qualifier annotations are valid.

CQUAL checks a program's correctness by performing *qualifier inference*. Rather than requiring the programmer to specify type qualifiers on every type in the program, using CQUAL the programmer can sprinkle a few qualifier annotations through the program, and CQUAL will infer the remaining qualifiers. It is this qualifier inference process that makes CQUAL easy to use.

CQUAL begins by adding fresh qualifier variables to every level of every type in the program. A qualifier variable stands for an unknown qualifier. For any explicit qualifier annotations in the program, CQUAL generates the appropriate constraint on the corresponding qualifier variable (see Section 3.3). Next CQUAL walks over the program and generates constraints between qualified types. For example, for an assignment `x = y`, CQUAL generates the constraint that the type of `y` is a subtype of the type of `x`. For a function call `f(x)`, CQUAL generates the constraint that the type of `x` is a subtype of the type of the formal parameter of `f`.

Applying the subtyping rules from above, these constraints between types yield constraints between qualifiers (variables and partial order elements). More formally, we are left with a set of constraints of the form $q_1 \le q_2$, where each $q_i$ is either a qualifier variable or a partial order element.

## 2.5 Flow-Sensitive Type Qualifiers

(If you are only interested in flow-insensitive type qualifiers, such as used in the tainted/untainted analysis or the user/kernel analysis, you may safely skip this section.)

The qualifier system described so far is *flow-insensitive*. For example, if we declare `x` to be an integer, then the contents of `x` is assigned a single type $q$ `int` for the whole program execution. For example, in

```
/* x has type q int */
x = ...;
/* x still has type q int */
```

the contents of `x` (here we are ignoring the $l$-value/$r$-value distinction) has the same qualifier $q$ before and after the assignment. For checking some properties, such as keeping track of the state of locks, we need *flow-sensitive* type qualifiers.

CQUAL supports flow-sensitive type qualifier inference, as described in [FTA02]. Each qualifier partial order may either be flow-insensitive, the default, or it may be flow-sensitive, as declared in the partial order configuration file (Section 3.3).

The flow-sensitive analysis consists of two separate passes over the source code. In the first pass, CQUAL performs flow-insensitive alias analysis and effect inference. This pass is done at the same time as flow-insensitive qualifier inference. In the second pass, CQUAL uses the results of the first pass to help perform flow-sensitive qualifier inference.

During flow-sensitive analysis, qualifiers on variables may change after an assignment:

```
/* x has type q int */
x = ...;
/* x now has type q' int */
```

### 2.5.1 Aliasing

Not every assignment in C is a simple variable assignment of the form shown above—updates can also occur indirectly, through pointers. CQUAL performs a unification-based, flow-insensitive alias analysis to compute an approximation to the aliasing behavior of the program. The alias analysis computes, for each pointer-valued expression $e$ in the program, the set of *locations* (either stack variables or heap memory) to which $e$ may point. The basic rule of the alias analysis is that given an assignment between pointers `x = y`, we unify (equate) the locations to which `x` and `y` can point. Formally, if `x` points to location $\rho_x$ and `y` points to location $\rho_y$, then upon seeing the assignment `x = y` we require $\rho_x = \rho_y$.

By using the results of alias analysis, CQUAL can track the effect of indirect updates, e.g.,

```
y = &x;
/* x has type q int */
*y = ...;
/* x now has type q' int */
```

Because the alias analysis is flow-insensitive, sometimes it will produce unexpected results. For example, the analysis will assume that `y` points to $\rho_x$, the location of `x`, no matter where the assignment `y = &x` actually occurs. The alias analysis does not track null pointers, and hence does not check for null pointer dereference statically.

In the above example, `y` pointed to exactly one location $\rho_x$. In this case we say that $\rho_x$ is *linear*, and the flow-sensitive analysis allows *strong updates* on $\rho_x$ (at assignments to $\rho_x$ it is given a new qualifier).

But what if the alias analysis determines that `y` may point to more than one location, say both `x` and `z`? Then the alias analysis will say that `y` points to $\rho_x$ where $\rho_x = \rho_z$ ($\rho_x$ is the location of `x` and $\rho_z$ is the location of `z`). In this case, we say that $\rho_x$ is non-linear, because it may represent more than one location.

Then at the assignment `*y = ...` we can't distinguish which location we are updating. Thus the flow-sensitive inference gives $\rho_x$, which stands for both `x` and `z`, the type $q''$ `int` after the assignment with constraints $q \leq q''$ and $q' \leq q''$. Intuitively, this means that the qualifier of location $\rho_x$ is either $q$ or $q'$. This is called a *weak update*.

### 2.5.2 Restrict

Clearly weak updates can cause the analysis to lose precision. CQUAL supports two language constructs to expose the alias analysis to programmer control. The idea behind both constructs is to introduce a lexical scope in which a non-linear location can locally be treated as linear.

In an idealized syntax, the `restrict` construct has the form

```
restrict x = e1 in e2
```

In our C notation, this construct will be written using the ANSI C qualifier `restrict` [ANS99]:

```
{
  T *const restrict x = e1;
  e2;
}
```

(The `const` needs to be there so that `x` is not modified within the scope of the declaration.)

In this construct, `e1` is a pointer to some location $\rho$. The name `x`, which can be used during evaluation of `e2`, is initialized to `e1`, but it is given a fresh location $\rho_x$. At the beginning and end of `restrict`, $\rho$ and $\rho_x$ have the same type.

The key property of `restrict` is that within `e2`, the location $\rho_x$ may be accessed, but the location $\rho$ may not. Outside of `e2` the reverse is true: $\rho$ may be accessed, but $\rho_x$ may not. This property is enforced automatically by CQUAL.

Since the accesses within `e2` go through location $\rho_x$, notice that the flow-sensitive qualifier inference may be able to treat $\rho_x$ as a linear location even if $\rho$ is non-linear. When the scope of `e2` ends the analysis may need to weakly update $\rho$.

For example, suppose we want to lock and then unlock a single array element:

```
spin_lock(&foo[i].lock);
...
spin_unlock(&foo[i].lock);
```

Then because the alias analysis does not distinguish array elements, both the lock acquire and release will be weak updates, and the analysis will conclude that `foo[i].lock` is both locked and unlocked, which is an error in the supplied partial order configuration file.

But we can use `restrict` to introduce a new name, and hence a new location, for `foo[i].lock`:

```
{
  spinlock_t *const restrict l = &foo[i].lock;
  spin_lock(l);
  ...
  spin_unlock(l);
}
```

Assuming no other array elements are used within this scope, `l` can be strongly updated to first be locked and then be unlocked. When the scope ends, the analysis will do a weak update from the final state of `l` (unlocked) to the state of the array.

The name `restrict` is deliberately chosen to correspond to the ANSI C qualifier; see [FA01] for a discussion.

### 2.5.3 Confine

While `restrict` can be used to locally recover strong updates, sometimes it is inconvenient, as it requires the programmer to come up with a new name. CQUAL also includes a construct `confine` that allows expressions to be restricted without introducing a new name. The syntax is

```
confine (e1) s2
```

Here `e1` is an expression that occurs within statement `s2`. As with `restrict`, the expression `e1` must evaluate to a pointer to some location $\rho$. Within `s2`, the analysis treats occurrences of `e1` as pointing to a fresh location $\rho'$. As before, location $\rho$ may only be accessed outside of `s2`, and location $\rho'$ may only be accessed within `s2`. At the beginning and end of `confine`, $\rho$ and $\rho'$ have the same type.

The key to making this sound is that `e1` must not contain any side-effects, and the value of `e1` must not change during evaluation of `s2`. As before this is checked automatically by CQUAL.

Going back to the last example in the previous section, with `confine` we can more conveniently annotate the program as

```
confine (&foo[i].lock) {
  spin_lock(&foo[i].lock);
  ...
  spin_unlock(&foo[i].lock);
}
```

In this case CQUAL will check that neither `foo` nor `i` changes during the evaluation of the `...`'s, and that none of the other aliases of `foo[i].lock` is changed in the scope of `confine`.

For further explanation of `restrict` and `confine`, see [FTA02, AFKT03]. As described in [AFKT03], CQUAL also supports automatic inference for `restrict` and `confine`, but only experimentally, and not currently as part of the regular system.

### 2.6 Browsing Qualifier Inference Results with PAM

CQUAL represents constraints between qualifiers as a directed graph whose nodes are qualifier variables and partial order elements. For each constraint $q_1 \leq q_2$, there is an edge from $q_1$ to $q_2$ in the graph. After all the constraints have been generated, CQUAL solves the constraints and, if there is an error, performs a second pass over the constraint graph to isolate the most useful error messages to display to the user. For example, if CQUAL ever generates the constraint `$tainted` $\leq$ `$untainted` then it will signal an error.

If you run CQUAL with PAM, then once CQUAL has completed qualifier inference you will be able to browse the inference results. There are a few important things to know about this browsing interface.

When displaying a source program, CQUAL colors each identifier according to its inferred qualifiers. Currently, CQUAL colors an identifier `x` by computing the colors of all partial order elements reachable in the constraint graph from any of `x`'s qualifier variables. If there is one such color, CQUAL uses it to color `x`. If there is more than one such color, CQUAL colors `x` purple. CQUAL colors individual qualifiers similarly. When computing colors, CQUAL does not include the qualifiers on fields of structures and unions, or on argument or result types of functions.

When you click on a qualifier variable $q$, CQUAL tries to show you how $q$'s color was inferred. If no partial order elements are reachable from $q$ in the constraint graph, CQUAL prints `No qualifiers`. Otherwise CQUAL displays the shortest path from any partial order element to $q$, and from $q$ to any partial order element.

The shortest path algorithm really works best with lattices, and it should also work with the discrete partial orders. Your luck with other partial orders may vary.

## 3 Applying Type Qualifiers to C

### 3.1 Names

As described in Section 2.4, CQUAL introduces qualifier variables at every position in a type.

Qualifier variables are named after the corresponding program variable. For an identifier `x`, the outermost qualifier on `x`'s type is given the name $x$. The names of qualifiers on nested `ptr` types are constructed by

appending $'$ to the name of the qualifier from the outer type. For example, given the declaration `char *x`, the $l$-value `x` is given the type $\&x$ `ptr(`$x$ `ptr(` $*$ $x$ `char))`.

The $i^{\text{th}}$ argument (starting with one) of function `f` has associated qualifier variable *f_argi*, and the return value of function `f` has qualifier variable *f_ret*.

When parsing a C program, CQUAL assumes that any identifier beginning with a dollar sign (`$`) is a type qualifier (e.g., `$tainted`, `$untainted`). Constant qualifiers appearing in a program must be declared in the partial order configuration file (Section 3.3). Qualifier variables are not normally added to the program explicitly, except in the case of polymorphism (Section 3.5).

## 3.2 Source Code Considerations

CQUAL accepts standard pre-processed source code and performs most C type checking. Currently error messages from the parser and standard C type checker are not displayed in PAM mode. If you wish to view the parser error messages in PAM mode, switch to the `*pam-results-buf*` buffer.

### 3.2.1 Multiple Files

CQUAL can analyze single files at a time or whole programs at once. Recall that CQUAL assigns fresh qualifier variables to every level of every type in the program. In particular, if a function `f` is declared with no explicit type qualifiers and is not defined anywhere, CQUAL assumes that the body of `f` places no constraints on `f`'s type qualifiers.

Thus, in general, it is best to run CQUAL on a whole program rather than on individual files, unless you are careful to fully annotate the types of every declared function. For example, suppose we have two source files `file1.c` and `file2.c`:

```
file1:  char *foo(void);
        void bar(void) {
           char *s = foo();
           ...
        }
```

```
file2:  char *foo(void) {
          $tainted char *t;
          return t;
        }
```

Because `foo`'s type has no explicit qualifiers, we will only discover that `s` is tainted if we analyze both files together.

In order to help avoid this problem, CQUAL generates a list of functions that are declared but not defined. In PAM mode this list is available by clicking on `Undefined Globals`. If a function is declared in a prelude file (Section 3.4) then is it not added to the undefined globals list.

To specify multiple input files to CQUAL, simply list them on the command line. When invoking CQUAL in PAM mode you can only enter one file. In this case CQUAL will expand the input file name using `glob`, which allows you to specify a set of input files using wildcards (for example, you can analyze `foo/*.c`).

### 3.2.2 Pre-Processed Source

CQUAL is designed to run on pre-processed source code. If you invoke CQUAL from the command line then you can use standard pre-processed source from `gcc -E`. The standard output from gcc -E can be processed by cqual, and the line numbers of error messages will be correct with respect to the original, non-pre-processed source code. PAM mode currently ignores `#line` directives in the preprocessed source. Thus, when using PAM, you will be browsing the pre-processed source and the line numbers of errors will, of course, be the line numbers of the pre-processed code. In practice this is rarely a problem.

The wrapper script, `gcqual`, makes cqual behave more like a regular compiler: it accepts non-pre-processed source files, preprocesses them, and calls cqual. It is the preferred interface to cqual. For more info, see Section 5.3. The program `remblanks`, provided in the `bin` directory, strips out all `#line` directives. The perl script `remquals`, also provided in the `bin` directory, strips out all identifiers beginning with a dollar sign, i.e., anything that might be a type qualifier.

### 3.2.3 Flow-Sensitivity

In CQUAL, a set of qualifiers forming a partial order can be declared to be flow-sensitive in the partial order configuration file (Section 3.3). Flow-sensitive analysis is an additional step, so to enable flow-sensitive analysis you also need to run CQUAL with the `-fflow-sensitive` option. If you do not need flow-sensitivity, you should probably not use `-fflow-sensitive` and you should probably comment `restrict` out of your partial order configuration file, because having either of these will cause CQUAL to consume more resources.

CQUAL's alias analysis is based on the C types, hence casts can introduce unsoundness into the alias analysis. E.g., given an assignment `x = (void *) y`, we do not assume that `x` and `y` point to the same location. As with qualifiers, the locations of structure fields are shared across instances of the same struct (Section 3.2.5). Thus if you cast a pointer to a structure to `void *` and then back to its original type, the locations of the fields, and their qualifiers, will be preserved.

CQUAL adds two extra forms to C to make flow-sensitive type annotations a bit easier:

- `change_type(e, T);` is a statement that updates the type of $l$-value `e` to have type `T`. This statement is equivalent to the assignment `e =` ⟨*something-of-type-T*⟩`;`, except that you don't need to come up with an expression for the right-hand side, only the type.
- `assert_type(e, T);` is a statement that checks whether the $r$-value of `e` has type `T`. Alternately, instead of using `assert_type` you can declare a variable `x` to have type `T` and try to initialize it to `e`.

The flow-sensitive analysis is monomorphic, hence any polymorphic qualifier declarations are ignored during flow-sensitive analysis. The `-fcasts-preserve` flag also is not implemented for flow-sensitive analysis.

### 3.2.4 Type Casts

By default CQUAL tries to keep track of qualifiers even in the presence of type casts. For example, consider the following program:

```
$tainted char **s;
void *t;

t = (void *) s;
```

When `s` is used as an $r$-value at the cast, it is used with the type $s$ `ptr(`$*s$ `ptr(`$**s$ `char))`. The $r$-value of `t` has the type $t$ `ptr(`$*t$ `void)`. Because of the typecast, normally CQUAL will generate the constraint $s \leq t$ but it will generate no constraints between deeper qualifiers of `s` and `t`.

In some cases, however, qualifiers should propagate through type casts. Each qualifier partial order can optionally be marked as being *preserved through casts*. (*Note: This option is unavailable with flow-sensitive qualifiers.*) If CQUAL analyzes the above program with `$tainted`'s partial order $Q$ marked as cast-preserved, then CQUAL will generate the constraints $*s \cap Q = **s \cap Q = *t \cap Q$. In other words, the `$tainted` qualifier and all other qualifiers in $Q$ will flow from $**s$ to $*s$ and to $*t$. CQUAL does not preserve qualifiers on structure fields, function arguments, or function result types at casts even with a cast-preserved qualifier, because this introduces too much imprecision.

Currently each structure or union with a distinct tag that is cast to a *void* or other primitive type is associated separately. (Primitive types interact with pointers just like they had type `void *`.) In other words, if both `struct foo *a` and `struct bar *b` are stored in `void *c`, then no constraints between `a` and `b`'s fields are generated, even if the two structures are related in some way.

In order to provide an escape mechanism from this behavior, if you cast to a type containing a qualifier $c$, then other qualifiers in $c$'s partial order will not propagate through that cast even if the partial order is cast-preserved. It is highly recommended that you do not enable `casts-preserve` for `const`, since many C programs will fail to type check if `const` propagates through casts.

**Warning.** Preserving qualifiers across casts still does not guarantee soundness. For example, consider the following code:

```
char *x, *y;
int a, b;

a = (int) x;      (1)
b = a;            (2)
y = (char *) b;   (3)
```

For line (1), CQUAL generates the constraints $*x = x = a$. For line (2), CQUAL generates the constraint $a \leq b$. And for line (3), CQUAL generates the constraints $b = *y = y$. Notice that we have $*x \leq *y$ but we do not have $*y \leq *x$.

Finally, CQUAL handles casts on aggregate types in another manner. Consider the following program:

```
struct foo *s, *u;
void *t;

t = (void *) s;
u = (struct foo *) t;
```

Now `s` has type $s$ `ptr(` $*s$ `struct foo)`, where `struct foo` itself has fields. In CQUAL, every *void* type has an associated set of aggregates it represents. Upon analyzing the first assignment statement, CQUAL generates the constraint $s \leq t$ as before, and it also adds `struct foo` to `t`'s set of aggregates. Then at the second assignment statement the constraint $t \leq u$ is generated as usual, and additionally `t`'s `struct foo` information is extracted and the appropriate constraints are generated with `u`'s fields.

### 3.2.5 Structures

In CQUAL each aggregate (i.e. struct or union) is modeled as having its own collection of fields with their own type qualifiers. Thus, for example, the following code type-checks:

```
struct buf
{
  char *data;
  int len;
};
void main (void)
{
  struct buf a, b;
  a.data = ($tainted char *)0;
  b.data = ($utainted char *)0;
}
```

For efficiency reasons, CQUAL unifies the fields of aggregates that interact (e.g. by assignments between structs or pointers-to-structs). Thus there is no subtyping or polymorphism when dealing with fields of aggregates.

When analyzing multiple files, CQUAL will match up structure types from different files field-by-field, and it will complain if a structure is declared differently in different files.

Finally, structure initializers are not always handled correctly. CQUAL requires that the shape on the right-hand side of an initializer match the shape of the type being initialized. For example, CQUAL won't understand the following code

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

unless it is rewritten as

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

### 3.2.6 Restrict

As described in Section 2.5.2, CQUAL uses the `restrict` qualifier to help improve the precision of flow-sensitive qualifier inference. Moreover, occurrences of `restrict` are checked by CQUAL. This means that if you analyze a program that already contains `restrict` (for example, newer versions of the standard C library headers), CQUAL will attempt to check its uses of `restrict`. Often these uses of `restrict` will fail to typecheck, usually because they are not annotated with `const` and because the alias analysis is not precise enough. Warnings about `restrict` qualifiers in code you did not write may be ignored.

If you want to disable `restrict` checking, simply remove `restrict` from your partial order configuration file. Doing so will improve the resource usage of CQUAL if you also run without `-fflow-sensitive`.

## 3.3 Partial Order Configuration File

The qualifier partial order configuration file is specified with a command-line option of the form

`-config` ⟨`po-file`⟩

All qualifiers except the three standard C qualifiers `const`, `volatile`, and `restrict` must begin with a dollar sign.

The partial order configuration file contains a series of partial order declarations. For now these partial orders should be lattices, the discrete partial order, or any three-point partial order. For other partial orders the implementation may or may not generate correct results.

Each partial order is assumed to be orthogonal to any other partial orders specified in the file. For example, if $q_1$ and $q_2$ are two qualifiers from different partial orders, then the constraints $q_1 \leq q_2$ and $q_2 \leq q_1$ are always satisfiable. More formally, the qualifier partial order is the product of each of the partial orders specified in the configuration file [FFA99].

The full grammar for partial order configuration files is given in Section 5.4. Here we show how to specify partial orders by example. As one example, consider the two point lattice:

```
partial order {
     $a < $b
}
```

This partial order declaration declares two qualifiers, `$a` and `$b`, where `$a` $<$ `$b`. But now what should happen when we declare, say `$a int x`? Recall that `x` is given the type &$x$ `ptr(`$x$ `int)`. Where should the `$a` qualifier go?

If not specified, CQUAL assumes that a qualifier annotates $r$-types, and that it should be less than or equal to the corresponding qualifier variable. In the case of the declaration of `x`, CQUAL adds the constraint `$a` $\leq x$.

As another example, consider the qualifiers used for tainting analysis:

```
partial order {
      $untainted [level = value, color = "pam-color-untainted", sign = neg]
      $tainted [level = value, color = "pam-color-tainted", sign = pos]

      $untainted < $tainted
}
```

As in the previous example here we define a two-point lattice with `$untainted` $<$ `$tainted`. Further, we explicitly declare that `$untainted` and `$tainted` should annotate $r$-types with the option `level = value` (the default). We also specify that `$tainted` is a positive qualifier (`sign = pos`), meaning that it should be made less than the corresponding qualifier variable when used in a type, the default. `$untainted` is declared as a negative qualifier (`sign = neg`), meaning that it should be made greater than the corresponding qualifier variable when used in a type. For example, if we declare

```
$tainted int t;
$untainted int u;
```

then CQUAL generates the constraints `$tainted` $\leq t$ and $u \leq$ `$untainted`.

Finally, the `color` options specify the colors that should be used in PAM mode to mark-up identifiers that have tainted or untainted types.

As another example, consider

```
partial order [casts-preserve] {
      $user [level=value, color = "pam-color-6", sign = eq,
            ptrflow=down, fieldflow=down, fieldptrflow=all]
      $kernel[level=value, color = "pam-color-4", sign=eq,
            fieldptrflow=all]
}
```

Here `$user` and `$kernel` are in a discrete partial order: neither `$user` $<$ `$kernel` nor `$kernel` $<$ `$user`. The partial order is declared as casts-preserving, which means these qualifiers are preserved across casts between types with different shapes (see Section 3.2.4). The qualifiers are declared to be non-variant (`sign = eq`), meaning that when they occur in the source code they should be may equal to the corresponding qualifier variables. The `ptrflow=down` option on `$user` means that `$user` flows from outer to inner pointer levels, and `fieldflow=down` means that `$user` also flows from structures to fields of structures, and finally `fieldptrflow=all` means that `$user` flows both to and from pointers to structures and the pointers to their fields. (See Section 2.3.)

As another example, consider ANSI C's `const`:

```
partial order {
      const [level = ref, sign = pos]
      $nonconst [level = ref, sign = neg]

      $nonconst < const
}
```

Here the `level = ref` options mean that `const` and `nonconst` annotate $l$-types instead of $r$-types. For example, given the declaration `const int x`, CQUAL will generate the constraint `const` $\leq \&x$ (not `const` $\leq x$ like it would if `const` qualified $r$-types).

If `const` is not declared in the partial order file, `const` annotations will be ignored during type qualifier inference. This is the recommended usage, since the `const` inference described in [FFA99] is not fully implemented in this system.

As another example, consider qualifiers for checking locking:

```
partial order [flow-sensitive] {
      $locked [level = value, color = "pam-color-locked", sign = eq]
      $unlocked [level = value, color = "pam-color-unlocked", sign = eq]
}
```

Here the `flow-sensitive` modifier means that `$locked` and `$unlocked` should be propagated flow-sensitively.

Finally, consider

```
partial order [nonprop] {
      volatile [sign = eq, level = ref, color = "pam-color-4"]
}
```

This entry declares that `volatile` is a non-propagating qualifier, i.e., it does not flow through the qualifier constraint graph. In other words, if `b` is `volatile` as we assign `a = b`, that does not mean that `a` is `volatile`.

## 3.4 Prelude Files

One way to add annotations to your program, especially annotations for library functions, is to use prelude files. One or more prelude files can be passed as arguments to CQUAL with the syntax

`-prelude` ⟨`file`⟩

If you specify one or more prelude files with this flag, then these files will be analyzed before any other files (and in order from left to right). Additionally, CQUAL assumes that any file called `prelude.cq` is a prelude file, whether or not it is preceded by `-prelude`. Thus a convenient way to maintain per-project prelude files is to include a local `prelude.cq` in the source directory.

The declarations in prelude files override declarations in non-prelude files. Therefore if there is some library function you want to give a polymorphic type (see below), you can give it a type in the prelude file and not worry about how it's actually declared in the source files.

CQUAL comes with some default prelude files:

- `config/prelude.cq` can be used to find format-string bugs in C programs.
- `config/proto-noderef.cq` and `config/linux-syscalls.cq` can be used to find user/kernel bugs in the Linux kernel.

## 3.5 Qualifier Polymorphism

One of the important techniques for improving the accuracy of CQUAL is to add *polymorphism* to qualified type annotations. Consider the following simple example code:

```
char id(char x) { return x; }
...
tainted char t;
untainted char u;
char a, b;

a = id(t); /* 1 */
b = id(u); /* 2 */
```

Because of call 1, we infer that `x` is a `$tainted char`, and therefore we also infer that `a` is `$tainted`. Then call 2 type checks (because `$untainted char` $\leq$ `$tainted char`), but we infer that `b` must also be `$tainted`.

While this is a sound inference, it is clearly overly conservative. Even though this simple example looks unrealistic, this problem occurs in practice, most notably with library functions such as `strcpy`. The problem arises because we are summarizing multiple stack frames for distinct calls to `id` with a single function type—`x` has to either be untainted everywhere or tainted everywhere. The solution to this problem is to introduce *polymorphism*, which is a form of context-sensitivity.

A function is said to be *polymorphic* if it has more than one type. Notice that `id` behaves the same way no matter what qualifier is on its argument `x`: it always returns exactly `x`. Thus we can give `id` the signature

`forall` $q$ `.` $id$ `fun (`$q$ `char) →` $q$ `char`

meaning that `id`, applied to a `char` qualified by any qualifier $q$, returns a `char` qualified by that same qualifier $q$. ($id$ is the qualifier on the function `id`—think of $id$ as qualifying the arrow.)

Operationally, when we call a polymorphic function, we *instantiate* its type—we make a copy of its type, replacing all the generic qualifier variables $\alpha$ with fresh qualifier variables. Intuitively, this corresponds exactly to inlining the function, except that instead of making a fresh copy of the function's code, we make a fresh copy of the function's type. In this case we say that `id` has a *polymorphic type*, which we constructed by *generalizing* the type variable $q$.

In CQUAL, if the `-fpoly` flag is specified on the command line, then CQUAL will perform polymorphic type qualifier inference; for the above example, `id` will be inferred to have the polymorphic type as specified. As you might expect, polymorphic qualifier inference requires more resources (time and space) than ordinary monomorphic qualifier inference.

CQUAL also allows the user to explicitly specify that certain functions are polymorphic in their qualifiers. This is useful when you don't have the code for a function but want to assign it the correct type (e.g., for library functions). Inside of a type, if you use qualifiers beginning with `$_`, they are interpreted as named qualifier variables. Names are sequences of integers separated by `_` (examples below). Function types containing explicitly named qualifier variables are generalized. For example, the declaration

```
$_1 int foo($_1 int);
```

gives `foo` the type

```
forall foo_ret . foo fun (foo_ret int) -> foo_ret int
```

Whenever `foo` is used, the generalized variables in its type will be instantiated with fresh qualifier variables:

| Program | Type of `foo` |
|---|---|
| `foo(a);` | *foo* `fun (`*foo_ret@0* `int) -> `*foo_ret@0* `int` |
| `...` | |
| `foo(b);` | *foo* `fun (`*foo_ret@1* `int) -> `*foo_ret@1* `int` |

In this way the qualifiers from distinct calls to `foo` are kept distinct. It is important to note that on any place on a type where an explicit qualifier variable is *not* mentioned, the qualifier in that position will not be generalized. For example, in

```
$_1 int foo($_1 int, float);
```

, all instances of `foo` share the same qualifier on the `float`, whereas they get separate instances of the qualifier on the `int` and return type.

CQUAL ignores the definition of any function given a polymorphic type, i.e., CQUAL assumes that all polymorphic function declarations are correct. The intention is to give polymorphic types to library functions, e.g., `strcpy`.

You can write down types containing more complicated constraints between the qualifiers using special notation. The declaration

```
$_1_2 int foo($_1 int);
```

assign `foo` the type

```
forall foo_arg0 foo_ret . foo fun (foo_arg0 int) -> foo_ret int
```

where *foo_arg0* $\leq$ *foo_ret*. In general, explicit qualifier names are interpreted as sets (not sequences) of integers, and if the set derived from one qualifier name $q_1$ is a subset of the set derived from another qualifier name $q_2$, then the constraint $q_1 \leq q_2$ is added. So, for example,

```
$_1_2 int foo($_2 int);
```

is an alternate declaration that assigns `foo` the same polymorphic type.

In general we recommend placing declarations of polymorphic functions in prelude files (see Section 3.4). Since declarations in prelude files override declarations in regular files, adding a function declaration to a prelude file has the same effect as rewriting functions declarations in all the source files.

Currently polymorphism in the flow-sensitive qualifiers is not supported. If you use functions given polymorphic signatures in a flow-sensitive analysis, CQUAL will simply ignore your polymorphic declarations during the flow-sensitive portion of the analysis.

## 3.6 Deep Subtyping with const

As described in Section 2.2, we use a conservative rule for pointer subtyping. This rule can lead to non-intuitive backwards flow, which often causes false positives. For example, consider the following code:

```
f(const char *x);
$tainted char *a;
char *b;
f(a);
f(b); /* b gets tainted */
```

Here the declaration of `a` adds the constraint $\texttt{\$tainted} \leq *a$ The first function call to `f` adds the constraints $a \leq x$ and $*a = *x$ The second function call generates the constraints $b \leq x$ and $*b = *x$. Notice that

$$\texttt{\$tainted} \leq *a = *b$$

and thus `*b` is tainted, which is counter-intuitive but necessary if `f` writes to `*x`.

Observe, however, that `f`'s argument `x` is of type `const char *`, so `f` cannot taint `*x` if it is not tainted in the first place. We can modify the subtyping rule for pointers to take advantage of this fact:

$$\frac{q_1 \leq q_2 \quad \texttt{const} \leq q_2 \quad \tau_1 \leq \tau_2}{q_1\ \texttt{ptr}(\tau_1) \leq q_2\ \texttt{ptr}(\tau_2)}$$

For example, for an assignment

```
const char *s;
char *t;
...
s = t;
```

CQUAL generates the constraints $t \leq s$ and $*t \leq *s$. If `s` were not `const`, CQUAL would generate the more conservative $*s = *t$.

If the flag `-fconst-subtyping` is enabled (the default), then CQUAL will use deep-subtyping for pointers explicitly qualified in the source program with `const`. I.e., the requirement $\texttt{const} \leq q_2$ above means that $q_2$ must have been explicitly annotated with `const`. Explicit annotations for `const` are kept in the qualifier graph even if `const` does not appear in the partial order file.

Currently, `const` annotations are ignored during the flow-sensitive portion of the analysis, so there is no deep subtyping for flow-sensitive type qualifiers.

## 3.7 Functions with Variable Numbers of Arguments

C allows functions to be declared to take a variable number of arguments by specifying a "rest" parameter `...` in a function declaration. As in C, by default CQUAL does not type check rest arguments (arguments passed to the rest parameter). For some analyses to be correct, however, we do need to type check rest arguments.

CQUAL extends the syntax of C to allow functions to have a *rest qualifier*, which is syntactically specified as a type qualifier on the `...` of a function. If a function `f` is declared with a rest qualifier and `f` is called with rest argument `p`, then the qualifiers of `p`'s types are constrained to be equal to `f`'s rest qualifier. More precisely, for each qualifier $q$ of `p`, CQUAL instantiates a fresh copy of `f`'s rest qualifier $r$ as *r_inst*i with the appropriate constraints and add the constraint $q = r_inst$i. If `f` has no rest qualifier, then no type constraints are generated for rest arguments.

For example, in the sample prelude file for tainting analysis `config/prelude.cq`, the function `sprintf` is declared as

```
int sprintf(char $_1_2 *str, const char $untainted *format, $_1 ...);
```

This declaration tells CQUAL to generate constraints $q \leq \$\_1\_2$ for all qualifiers $q$ on rest arguments to `sprintf`.

Be aware that the current implementation of varargs annotations is not completely sound. Specifically, rest qualifiers may be lost when varargs functions are stored and retrieved through function pointers. Also, note that due to limitations with CQUAL's parser, at most two qualifiers or qualifier variables can be specified on ..., one to the left and one to the right.

### 3.8 Old-Style Functions

If you declare a function `f` using the K&R style, then no type checking is done to arguments at a call to `f`. This matches the behavior of C, but it can lead to unexpected results. If wish to run CQUAL on a program written in the K&R style, you can use the GNU package `protoize` to ANSIfy the function definitions and declarations. CQUAL will warn about some, but not all, uses of old-style functions.

### 3.9 Operators

In some type qualifier-based analyses, the user-defined qualifiers interact with C operators. For example, CARILLON requires strings that are dereferenced to be qualified with `$NONYEAR`.

CQUAL provides an experimental interface for adding such rules. You can annotate operators with type qualifiers by declaring special functions (probably in a prelude file). For example, to require that every dereferenced object be a `$NONYEAR`, you can declare

```
$$a _op_deref($$a *$NONYEAR);
```

This declaration says that `_op_deref` is a polymorphic function that takes a pointer to type `$$a` and returns a value of type `$$a`, for any type `$$a`. Further, that pointer must be qualified with `$NONYEAR`.

Currently you can only add a signature to the dereference operator. The constraints are applied to every dereference, even implicit ones. For example, the assignment `y = x;` is interpreted as dereferencing both `&y` and `&x`, even though the dereference operator, `*`, is never mentioned.

### 3.10 iquals

CQUAL includes the program IQUALS, which is a simple interface to the qualifier constraint solver. IQUALS accepts as an option a partial order configuration file, in the same format as CQUAL. IQUALS reads in a file of qualifier constraints, solves the constraints, and then outputs the results.

IQUALS is intended mainly as a debugging tool for the qualifier constraint solver.

## 4 PAM Mode

### 4.1 The Interface

In the default configuration, PAM is invoked by typing `M-x cqual` in emacs. PAM launches CQUAL as a sub-process. CQUAL analyzes its input files and sends the results back to PAM. CQUAL then enters an event loop in which it responds to mouse-click events from PAM.

There are five active bindings when in PAM mode:

| | |
|---|---|
| `middle click` | Follow hyperlink |
| `shift middle click` | Jump to qualifier definition |
| `C-c C-l` | Follow hyperlink |
| `C-c C-f` | Run CQUAL on another file |
| `C-c C-r` | Exit PAM and kill all PAM buffers |

## 4.2 Changing the Analysis

PAM runs the analysis defined by the variable `pam-default-analysis`, which is a list of strings, the first of which is the path name of the executable and the rest of which are arguments. PAM will interactively ask for the target file and append it to the argument list. For example, here is the default analysis in the author's `personal.el`.

```
(setq pam-default-analysis '("/home/jfoster/cqual/bin/cqual"
                             "-fpam-mode"
                             "-hotspots"
                             "10"
                             "-fflow-sensitive"
                             "-config"
                             "/home/jfoster/cqual/config/lattice"))
```

You can add extra options to PAM mode by inserting them into the list. For example, if you want to use the default prelude file for tainting analysis (Section 3.4), change the above to

```
(setq pam-default-analysis '("/home/jfoster/cqual/bin/cqual"
                             "-fpam-mode"
                             "-hotspots"
                             "10"
                             "-fflow-sensitive"
                             "-prelude"
                             "/home/jfoster/cqual/config/prelude.cq"
                             "-config"
                             "/home/jfoster/cqual/config/lattice"))
```

Be sure to re-evaluate your `personal.el` file (`M-x eval-buffer`) or re-launch EMACS after making a change to the file.

## 4.3 Customizing Colors

You can customize the colors that PAM uses by editing your `.emacs` file. By default, PAM defines nine type faces: `pam-color-`$i$, where $i$ is between 1 and 8, and `pam-color-mouse`, the color to use to highlight a link when the cursor is dragged over it.

If you want to change a color defined by PAM, use `custom-set-faces`:

```
(require 'pam-faces)
(custom-set-faces
      '(pam-color-1 ((t (:foreground "Yellow" :underline t))) t)
      '(pam-color-6 ((t (:foreground "Black" :underline t))) t))
```

If you want to add a new type face, use `pam-add-face`:

```
(require 'pam-faces)
(pam-add-face pam-color-tainted ((t (:foreground "Red" :underline t))))
(pam-add-face pam-color-untainted ((t (:foreground "Green" :underline t))))
```

# 5 Reference

## 5.1 cqual

**Synposis**

```
cqual [ options ] source-files ...
```

**Description** Invoke the type qualifier inference on `source-files`. CQUAL accepts all of the standard GCC options, most of which have no effect on CQUAL's behavior. CQUAL silently ignores any options it doesn't understand.

| | |
|---|---|
| `-config` ⟨`file`⟩ | Specifies the partial order configuration file to use (Sections 3.3 and 5.4) |
| `-prelude` ⟨`file`⟩ | Specifies the prelude file to use (Section 3.4) |
| `-hotspots` ⟨`num`⟩ | If specified, generate a list of the top `num` qualifier variables involved in error paths. Don't take this information too seriously. |
| `-program-files` ⟨`file`⟩ | Add the files listed one per-line in *file* to the list of files to be analyzed. |

In addition, there are a number of flags that change cqual's behavior. If `-f`⟨`flag-name`⟩ appears as an option, the flag is enabled. If `-fno-`⟨`flag-name`⟩ appears as an option, the flag is disabled.

| | |
|---|---|
| `pam-mode` | Enter into PAM mode after analysis is complete. Usually only used if PAM itself is invoking cqual. Default value is off. |
| `print-quals-graph` | Generate `quals.dot`, containing the (non-transitively closed) constraints, which is interpretable by `dot`. Default value is off. |
| `strict-const` | Assume anything not marked const is non-const. Default value is off. |
| `print-results` | Print a summary of the results after the analysis is complete. Intended mostly for regression testing. Default value is off. |
| `use-const-subtyping` | (Section 3.6) Use `const` qualifiers to increase the precision of the analysis by using subtyping, rather than equality, under a `const` pointer. This flag has no effect on flow-sensitive analysis. Default value is on. |
| `flow-sensitive` | (Section 2.5) Perform flow-sensitive qualifier inference after flow-insensitive qualifier inference. If you enable this flag the analysis will consume more resources, even if no flow-sensitive qualifiers appear in the source code. Hence we recommend you disable it if you do not need the flow-sensitive analysis. Default value is off. |
| `ugly` | Display memory addresses next to qualifier variable names. This is mainly useful for big programs that tend to reuse local variable names—without using this flag it's hard to tell them apart. |
| `explain-errors` | For each error message, display a constraint path exhibiting the error. This is useful when running CQUAL directly from the command line without using PAM. Default value is off. |
| `poly` | Enabled polymorphic recursive inference. This requires more time and memory than purely monomorphic inference, but the result will be more precise. Default value is off. |

## 5.2 iquals

**Synposis**

```
iquals [ -config ⟨file⟩ ] [ -g ] constraint-file
```

**Description** Solve the qualifier constraints in `constraint-file`.

| | |
|---|---|
| `-config [file]` | Specifies the partial order configuration file to use. |
| `-g` | Generate `quals.dot`, containing the (non-transitively closed) constraints, which is interpretable by dot. |

The `constraint` file should consist of a list of constraints of the form

| | |
|---|---|
| `q1 <= q2` | inequality |
| `q1 = q2` | equality |
| `q1 == q2` | unification |
| `q1 <= q2 ==> q3 <= q4` | conditional inequality |

Here if `qi` begins with `$` it is assumed to be a partial order element specified in the partial order file. Otherwise `qi` is assumed to be a variable. Variables may contain numbers, upper- and lower-case letters, and underscores.

## 5.3 gcqual

**Synposis**

```
gcqual [-debug] [-cc <CC>] [-cqual <cqual>] [cqual options] [-- [CC options]] <file1>
                                            <file2> ...
```

**Description** Preprocess the given input files with the C preprocessor, then call cqual on the results. `gcqual` doesn't preprocess input files ending in `.i`, and doesn't preprocess any input files if the `-fpreprocessed` flag is given. `gcqual` always preprocesses prelude files. If no lattice file is specified on the command line, `gcqual` looks in the following places, in order, until one is found:

- `$CQUAL_CONFIG_DIR/lattice`
- `./lattice`
- `/usr/local/share/cqual/lattice`

If no lattice or prelude files are specified on the command line, then `gcqual` also loads all prelude files (`*.cq`) in the same directory as the lattice file it finds.

| | |
|---|---|
| `-debug` | display commands executed by `gcqual` |
| `-cc <CC>` | use `<CC>` instead of `gcc` |
| `-cqual <cqual>` | use `<cqual>` instead of `cqual` |

## 5.4 Partial Order Configuration File

The partial order configuration file should contain a series of entries defining partial orders. In the current version of the code these partial orders should be lattices to generate valid inference results. Inference should also work correctly on any discrete partial order, and on any of the three-point partial orders. A future version of CQUAL will correct this limitation.

Below is the grammar for partial order configuration files. In this grammar, $x^*$ means zero or more occurrences of `x`, and `[` $x$ `]`$^?$ means either zero or one occurrences of `[` $x$ `]`.

| | | |
|---|---|---|
| *po-defn* | ::= | `partial order [` *po-opt*$^*$ `]`$^?$ `{` *po-entry*$^*$ `}` |
| *po-opt* | ::= | `nonprop` |
| | \| | `flow-sensitive` |
| | \| | `casts-preserve` |
| *po-entry* | ::= | *qual-name* `[` *qual-opt*$^*$ `]`$^?$ |
| | \| | *qual-name* $<$ *qual-name* |
| *qual-opt* | ::= | `color = "`*color-name*`"` |
| | \| | `level =` *level* |
| | \| | `sign =` *sign* |
| | \| | `ptrflow =` *flow-dir* |
| | \| | `fieldflow =` *flow-dir* |
| | \| | `fieldptrflow =` *flow-dir* |
| *level* | ::= | `ref` \| `value` |
| *sign* | ::= | `pos` \| `neg` \| `eq` |
| *flow-dir* | ::= | `down` \| `up` \| `all` |

In addition, comments beginning with `/*` and ending with `*/` may be added to the configuration file. Comments may not be nested, following the C convention.

# References


[AFKT03] Alex Aiken, Jeffrey S. Foster, John Kodumal, and Tachio Terauchi. Checking and Inferring Local Non-Aliasing. In *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 129–140, San Diego, California, June 2003.

[ANS99] ANSI. *Programming languages – C*, 1999. ISO/IEC 9899:1999.

[EFA99] Martin Elsman, Jeffrey S. Foster, and Alexander Aiken. Carillon—A System to Find Y2K Problems in C Programs, 1999. `http://bane.cs.berkeley.edu/carillon`.

[Fos02] Jeffrey Scott Foster. *Type Qualifiers: Lightweight Specifications to Improve Software Quality.* PhD thesis, University of California, Berkeley, December 2002.

[FA01] Jeffrey S. Foster and Alex Aiken. Checking Programmer-Specified Non-Aliasing. Technical Report UCB//CSD-01-1160, University of California, Berkeley, October 2001.

[FFA99] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A Theory of Type Qualifiers. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 192–203, Atlanta, Georgia, May 1999.

[FTA02] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-Sensitive Type Qualifiers. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002. To appear.

[GA01] David Gay and Alexander Aiken. Language Support for Regions. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 70–80, Snowbird, Utah, June 2001.

[PAM] Christopher Harrelson. Program Analysis Mode. `http://www.cs.berkeley.edu/~chrishtr/pam`.

[STFW01] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *Proceedings of the 10th Usenix Security Symposium*, Washington, D.C., August 2001.

# A Limitations and Bugs

- Only some partial orders will work correctly: any lattice, any discrete partial order, and any of the two- or three-point partial orders. Your mileage will vary with other partial orders.
- The constraint graph traversal in PAM mode really works best if the program is analyzed using only a single, two-point lattice. It works for other partial orders, but less reliably.
- `const` inference (described in [FFA99], which used a previous version of this system written in ML) is not fully implemented. Specifically, the relationship between `const` fields and `const` structures is not handled fully correctly.
- Only the dereference operator can be annotated with qualifiers without hacking the source code.
- If you kill a marked-up buffer in PAM mode, then you need to re-run CQUAL to recover the buffer.
- Structure initializers aren't always handled correctly.
- The flow-sensitive analysis does not support polymorphism, gated qualifiers, or `-fconst-subtyping`.

# B Copyright