For this assignment, you will get experience using CQual, an example of a whole-program, static analysis tool.

1 Written Exercises

1. Given

\[ Q ::= \text{const} \mid \text{nonconst} \]
\[ \Gamma \equiv \text{strdup: } \kappa_0 \ (\kappa_1 \ \text{ref} \ (\kappa_2 \ \text{str}) \rightarrow \kappa_3 \ \text{ref} \ (\kappa_4 \ \text{str})), \]
\[ \text{cat: } \kappa'_0 \ (\kappa'_1 \ \text{ref} \ (\kappa'_2 \ \text{str}) \rightarrow \kappa'_3 \ \text{ref} \ (\kappa'_4 \ \text{str}) \rightarrow \kappa'_5 \ \text{ref} \ (\kappa'_7 \ \text{str})) \]

Where \( \kappa_0, \kappa'_0, \kappa_1 \) etc. are fresh qualifier variables. Using the simple, non-polymorphic system in Section 2 in the paper, prove whether or not the following program \( P \) (in a language extended to include string constants) is well-typed (i.e., we have \( \Gamma \vdash^t P : \tau \) for some \( \tau \)). If it is well-typed, provide a fully-annotated version of the program (i.e., show the results of qualifier inference).

let x = ref "hello" in
let y = strdup x in
cat x y

2. The qualifier language (Fig. 1 in the paper) does not include structured types. Add support for pairs, extending \( \nu \) (see text just preceding section 2.1 in the paper) and \( e \) to be

\[ \nu ::= \ldots \mid \tau \times \tau \]
\[ e ::= \ldots \mid (e_1, e_2) \mid \text{fst} \ e \mid \text{snd} \ e \]

(a) Extend the checking and subtyping rules in Fig. 3 to include rules for checking the new syntactic forms.

(b) Extend the inference rules in Fig. 4 to include rules for inferring qualifiers on the new syntactic forms.

Hint: look at the Cardelli paper to see how pairs are handled in normal systems, for subtyping and checking, and also the rules for qualified \( \rightarrow \) types in the Foster et al. paper for an idea of how to proceed for pairs in that context.
2 Checking Invariants using CQUAL

You can do one of two things: (1) check for possible deadlock in multi-threaded programs using CQUAL’s flow-sensitive type qualifiers, or (2) check for flows of data arising from an untrusted source flowing to a trusted sink. These options are described in turn.

After executing one of these, you should write a couple of paragraphs to address the following questions: Did you find any bugs? Did you find any false positives? How many of each? Were they easy to classify? Why or why not? What was your overall experience in using the tool?

CQUAL is installed on the junkfood cluster (compiled for Linux) in /fs/unsupported/cqual (version 0.991).

Option 1: Checking for deadlock in concurrent code

CQUAL supports flow-sensitive type qualifiers. That is, while the a value’s type, such as int or float, is invariant no matter how that value flows through the program, a value’s type qualifier is allowed to change. The example given in the manual in Section 1.5 shows this for types lock_t, which can have qualifier $locked or $unlocked depending on the value’s state.

For this project, you should find a program that uses POSIX threads, and check it for deadlocks. Section 2.5 of the manual has some examples for doing this in the Linux kernel using spin locks. You should adapt this technique by defining a similar lattice for non-reentrant mutexes used in POSIX threads. In particular, the function call pthread_mutex_lock should take a pthread_mutex_t argument with qualifier $unlocked, where the argument’s qualifier will have qualifier $locked following the call. The call to pthread_mutex_unlock should do the opposite. You will have to create a prelude file to specify this effect of these functions.

Below is a list of programs that use POSIX threads (based on my cursory look). Pick one, run the analysis, and write a couple of paragraphs about the experience, as directed above. Note that in this case you should use CQUAL version 0.98, rather than 0.991.

- http://caudium.net/
- http://sourceforge.net/projects/kadet
- http://nullhttpd.sourceforge.net/httpd/
- http://www.lysator.liu.se/~pen/phttpd/
- http://www.redhat.com/docs/manuals/tux/
- http://www.xitami.com/
- http://www.enderunix.org/aget/
- http://www.sleepycat.com/
Option 2: Checking for usage of tainted data

Tainted data is data received from a source outside the program that may not be trusted. For example, this could be data received in the fields of web form, from within an untrusted file, or typed in from the console. Tainted data is the source of many security bugs. For example, format string attacks are an attack in which tainted data is passed to `printf` in lieu of a format string, such that it smashes part of the stack (this attack is described in the CQUAL paper we read). Another attack is including user data directly within a SQL query string passed to database. This user string could include query directives that reveal sensitive information.

Data from an untrusted source can be given the qualifier `$tainted`, and functions that require data to be sanitized can be given the qualifier `$untainted`. If tainted data flows to a function that requires it to be untainted, it’s an error.

First, find a server program that takes user input, and check for format string bugs. If you like, you can try to repeat results of experiments run by Foster et al. Then, extend the notion of tainted data and annotate more functions. For example, make sure that strings passed to `exec` are always untainted, or strings passed to `open`. See how well CQUAL works as you add more and more annotations.