CSC2125

Type Qualifiers Homework with Solutions

1 CCured

Consider the following C program fragment.

```
char *foo () {
  char A[10];
  char *p;
  p = A;
  p = p + 20;
  return p;
}
```

1. What type qualifier (SAFE, SEQ, or DYN) will CCured infer for `p`? Explain.

   SEQ. The memory areas to which `p` can point, namely `A`, are of homogenous type (`char`), so `p` isn't DYN. Pointer arithmetic is performed on `p`, so `p` isn't SAFE. Therefore, `p` is SEQ.

2. Will CCured insert any runtime checks into `foo`? Explain.

   According to [2], no, since `p` is never dereferenced or cast to SAFE. (The CCured implementation does insert run-time checks, but this cannot be inferred from [2].)

3. Write a small function `bar` which calls `foo`, does not perform any checks on the value returned by `foo`, and uses the return value in a way that violates memory safety.

   ```
   void bar () {
     char *q = foo ();
     *q = 'a';
   }
   ```

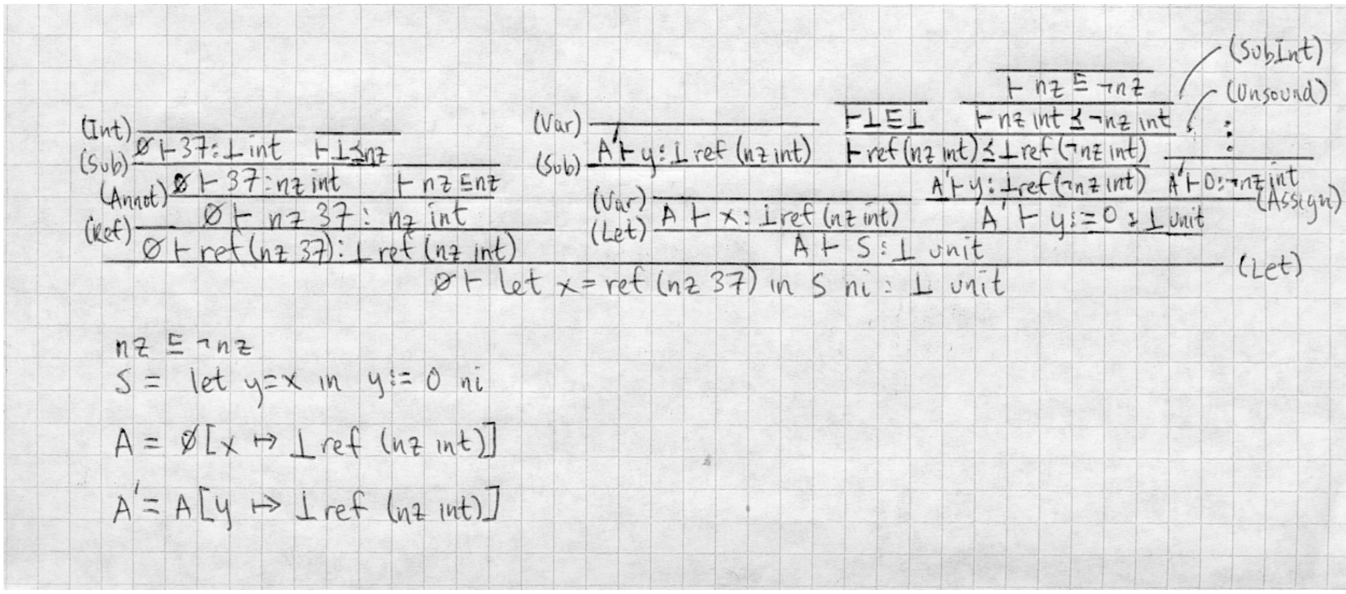
   Since `q` points to a local variable in `foo`, the statement `*q = 'a'` is unsafe since `foo` is out of scope. CCured does not insert an appropriate run-time check into `bar` to prevent this operation.

2 Flow-Insensitive Type Qualifiers

Consider the program P, written in the functional language used in [1]:

```
let x = ref(nonzero 37) in
let y = x in
  y := 0;
ni ni
```

Using the type-checking rules in [1], but with (Unsound) instead of (SubRef), show that this program type-checks. Specifically, prove $\emptyset \vdash P : \bot\ \mathit{unit}$, assuming $\mathit{nonzero}\ \mathit{int} \preceq \neg\mathit{nonzero}\ \mathit{int}$. Show that type-checking fails (as it should) using the correct rule, (SubRef), instead of (Unsound).



Using the (SubRef) rule would yield the premise $\vdash$ *nonzero int* $=$ $\neg$*nonzero int*, which is false (not provable).

3 Flow-Sensitive Type Qualifiers

Consider the function declarations

```
void acquire (unlocked lock_t *l);
void release (locked lock_t *l);
```

where `acquire` changes the qualifier of its `unlocked` argument to `locked` and `release` changes the qualifier of its `locked` argument to `unlocked`. The `locked` and `unlocked` qualifiers are incomparable (incompatible). For each of the following program fragments, state whether it will pass cqual's flow-sensitive type-checking and, if not, whether the `restrict` construct could be used to make it pass (without changing the semantics of the program). Briefly justify your answers.

1. ```
   if (...)
     l = l1;
   else
     l = l2;
   acquire (&l);
   release (&l);
   ```

   Where `l1` and `l2` are initially `unlocked`.

   Pass, since `l1` and `l2` are both initially unlocked, `l` is inferred to be unlocked at the join point following the if-then-else block.

2. ```
   if (x > 0)
     acquire (&l);
   if (x > 0)
     release (&l);
   ```

Where `l` is initially `unlocked`.

Fail, since `l` is inferred to be potentially locked and unlocked at the join point following the first if-block. Restrict doesn't enable type-checking here since the cause of the failure is cqual's path-insensitivity, not its imprecise alias analysis.

3. ```
   for (i = 0; i < N; i++)
   {
     acquire (&L[i]);
     release (&L[i]);
   }
   ```

   Where `L` is an array of `N` distinct locks, each initially `unlocked`.

   Fail, since cqual does not infer that both occurrences of `L[i]` refer to the same lock. Replacing the body of the for loop with

   ```
   restrict l = L[i] in
   {
     acquire (&l);
     release (&l);
   }
   ```

   enables the program to pass.

4. ```
   struct locknode {
     lock_t *lock;
     struct locknode *next;
   };

   while (L != NULL)
   {
     acquire (L->lock);
     release (L->lock);
     L = L->next;
   }
   ```

   Where `L` initially points to the head of a list of `locknode`s, each with an initially `unlocked` `lock`.

   Pass; the cqual implementation infers that both occurences of `L->lock` refer to the same lock, which seems to be inconsistent with the paper and cqual's behaviour on the previous program.

References

[1] Jeffrey S. Foster, Manuel Fahndrich, and Alexander Aiken. "A Theory of Type Qualifiers." In ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'99), Atlanta, Georgia, May 1999. http://citeseer.ist.psu.edu/foster99theory.html.

[2] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In Twenty-Ninth ACM Symposium on Principles of Programming Languages, Portland, OR, Jan. 2002.