A 341 View of Cyclone Memory Management

Dan Grossman

10 March 2004

Cyclone homepage:

`http://www.research.att.com/projects/cyclone`

What is Cyclone

- A language and compiler for a safe, C-like language
  - safe: like ML, Scheme, Java, etc. no data corruption or "undefined behavior"
  - C-like: explicit pointers, array representation, resource management, etc.
- Accepting exactly the "defined" C programs is mathematically impossible
  - Err on side of caution and require more specific types
- A research prototype
  - Pretty cutting edge (1 of 4 designers is talking)
  - But 341 has taught you enough to handle it!
- Today, just a flavor of one issue: memory management (and a lot of C code)

Memory Management

Fact: Applications run in finite memory (the less the better), but most languages don't restrict how much you create. When the language implementation runs out, the program dies or further allocation fails.

The standard HLL workaround:

- Local variables have a scoped (stack-based) lifetime
  - You cannot get references (pointers) to them, only use them and pass them by value
- Heap objects (everything else) conceptually live forever
- A garbage collector reclaims unreachable heap objects
- You hope not to run out of memory

C Memory Management

Life in C (and for the most part C++) is more dangerous:

- Local variables have a scoped lifetime
  - But you can get pointers to them
  - Following a *dangling* stack pointer is "undefined" (in practice accesses a totally different value of unknown type)
- Heap-allocated objects live until and only until you call free with a pointer to the object
  - Accessing a freed object is "undefined"
  - Freeing an object twice is "undefined"
  - Free too little and you can run out of memory

See examples in `bad.c`*...*

Non-Cyclone Solutions

1. Be really careful
   - Nobody is careful enough
   - Difficult to debug
   - Inappropriate for systems with untrusted code
2. Use garbage-collection and ban address-of-locals
   - can be hard to predict performance
   - unusable in some environments
   - address-of-locals can be elegant and useful when used correctly

   See examples in `okay.c`*...*

Cyclone and Region Names

So Cyclone uses more fine-grained types to prevent dangling-pointer dereferences:

- C: `t*` "pointer to memory (that for some portion of past/present/future) holds a `t`"
- Cyclone: `t*‘r` "pointer to memory in region named `‘r` that holds a `t` until region named `‘r` is deallocated"

A local-scope with label `L` creates a region named `‘L` that is deallocated when control leaves the scope.

Note on terminology: When a "region is deallocated" all its objects are deallocated (and new objects cannot go into it).

Note on inference: Somewhat like ML infers types, Cyclone can infer region names for local variables.

So far, We Have Overdone It

By including region names in types, we cannot return a pointer to a local or dereference a pointer outside of the local's scope. (Good)

But for the same reason, we cannot pass a pointer to another function, which is way too restrictive. (Bad)

The solution is "region polymorphism" just like ML has "type polymorphism". Example:

```
int*‘r inc_and_return<‘r>(int*‘r p) {
    *p=(*p)+1;
    return p;
}
```

"For all region-names ‘r, `inc_and_return` can take a pointer into ‘r and return a pointer into ‘r, (as long as the region is live)."

Now That's Better

To make this more convenient:

- Quantification and instantiation are both optional.
- And you can omit regon names that appear only once in the type.

See `play.cyc`...

So in general you can "pass live pointers down" but you cannot use assignment or return to create dangling pointers.

But that's just local variables. We cannot put everything on the stack, but `free` is unsafe.

(For one thing, C requires knowing the size (including array lengths) of all local variables.)

Growable Regions

A growable region has the same lifetime rules as a local-block (i.e., stack) region, but a *handle* lets you allocate more objects of any size into it.

Silly Example:

```
void f(int i, int j) {
  region r; // create region, r is the handle
  for(; i >= 0; --i) // make int[10] array for no reason
    rcalloc(r,j,sizeof(int));
}
```

Allocation function (`rcalloc`) takes a handle telling it where to put the object.

On return all i arrays get deallocated at once (faster than `i` calls to free).

What's the Point

Key idiom is *passing a region-handle to a callee* because often:

- Only the caller knows how long a computed result is needed (so it passes a handle for a region that lives long enough).
- Only the callee knows how big the result is (so it cannot safely assign into stack-allocated object.

See `play.cyc`...

I cannot emphasize enough that in C, the "convenience" of stack allocation over `malloc/free` leads to "caller allocates what should be enough room" and then it isn't because of malicious users.

This is the easiest exploit for virus writers and you should be fired for writing code that enables it!

Meta-point: Safe languages can teach you idioms for living in an unsafe world. Don't take shortcuts.

Conclusions

- C-style manual memory management is sometimes a fact of life.
- Certain idioms are safe when used carefully.
- Cyclone encodes some important idioms in a safe language and prevents unsafe uses.
- Polymorphism and inference make the result usable.
- Caller-chooses-lifetime, callee-chooses-size should be much easier than it is in C: "growable regions" do this nicely.