# Homework 8
# Memory

98-317: Hype for Types

Due: 30 Oct 2018 at 6:30 PM

## 1 Introduction

This week in lecture, we discussed ancient and modern techniques for memory management. Though we are all well-conditioned to state that manual memory management is bad and error-prone, the truth is that garbage collection is neither sufficient nor realistic for a wide variety of real-world software programs, ranging from embedded development to high-frequency trading. Nevertheless, we don't have to resort to the barbaric world of `malloc` and `free`, which aren't even fast![1] The world of type systems and static analysis gives us better tools to work with memory.

**Turning in the Homework** You should submit a PDF with your solutions to Autolab under Memory.

[1]Recall your 15-213 implementation of `malloc`. Its slowness is due to fragmentation and unpredictable free patterns, both of which are avoidable.

## 2 Tail Recursion

Many languages, such as Standard ML, implement an optimization known as *tail call optimization* that transforms functions whose return value is another function call from having a deeply nested call stack to a shallow one, by reusing the caller's stack frame. This prevents stack overflow on a large class of programs, and is widely considered an important optimization that is catching on in the non-functional world as well.

However, there are restrictions in the C language that prevent tail call optimization in the general case, even when the caller clearly returns another function call as its return value. Namely, the address-of operator, `&`, can be used to pass addresses of caller variables to callees. That means that a tail callee may rely on data lying within the caller's stack frame, and therefore the caller stack frame cannot be safely reused.

We can write some interesting programs with this observation. In fact, *very interesting* programs. Chicken, a cross-compiler for Scheme to C, actually uses the call stack to avoid heap allocations. We should be able to do the same!

Let's implement linked lists using the C call stack. In particular, let's implement lists of integers. We will use the following interface:

```c
typedef struct list_node {
    int data;
    struct list_node *next;
} list_t;
```

Take for granted a fixed array of data and its length $> 0$:

```c
static int DATA[] = {9, 8, 3, 1, 7, 9, 8, 2, 1, 0};
static int DATA_LEN = 10;
```

And a checker function:

```c
void check_list(list_t *list) {
    for (int i = 0; list != NULL && i < DATA_LEN; i++, list = list->next) {
        assert(DATA[i] == list->data);
    }
    assert(list == NULL && i == DATA_LEN);
}
```

**Required Task 1.** Write a C routine that will successfully invoke `check_list` on a valid linked list containing the elements in `DATA` in order.

For example, here is a correct invocation of the function:

```
list_t *list = NULL;
for (int i = DATA_LEN - 1; i >= 0; i--) {
    list_t *new = malloc(sizeof(list_t));
    new->data = DATA[i];
    new->next = list;
    list = new;
}
check_list(list);
```

However, your implementation *must not* call `malloc`, `calloc`, `alloca`, `mmap`, `brk`, `sbrk`, or any similar dynamic allocation functions. It should work without modification if the example list is extended. It should not store the list as static data, in an array, a large struct, etc. It should use the C call stack alone to implement the linked list. It does not have to be tail-recursive, but the natural implementation will be.

*Hint:* You can declare a struct on the stack:

```
{
    list_t list;
    list.data = 0;
    list.next = NULL;
}
```

*Hint 2:* Our solution is 15 lines.

## 3 Lifetime Subtyping

In lecture, we briefly discussed the notion of *lifetimes*. Lifetimes parametrize references, making them safe to use and allowing us to prove that dangling pointers are not dereferenced at runtime. Since lifetimes are associated with reference types, we would expect to be able to discuss polymorphism and subtyping for them.

A language like Rust allows us to describe lifetime polymorphism and subtyping. Here is a small program in Rust demonstrating lifetime polymorphism:

```
fn foo(xs: &mut [i32]) {
    xs[0] = xs[xs.len() - 1];
}
```

You don't have to know Rust to understand this function; it merely sets the first element of a list (technically called a *slice*) of integers to be equal to the last element of said list. The type of `xs`, `&mut [i32]`, denotes that it takes a mutable reference to a list of integers. That is, *somebody else* owns the actual list and manages its allocation and deallocation, while we only have the ability to mutate its fields. That is why we can only swap around elements, not extend or shrink the length of the list.

Rust uses affine types, so perhaps you're wondering how we're able to easily copy an element into another field. Integers are small, so there is primitive support for copying integers. However, if we wanted to make this function polymorphic, things get trickier:

```
fn foo<'a, T>(xs: &mut [&'a T]) {
    xs[0] = xs[xs.len() - 1];
}
```

What we've changed here is that `xs` no longer takes a list of integers, but rather a list of references to some type `T`. References are very much like pointers at runtime, so copying them is still no issue at the machine level. But now we have to ask what the lifetimes of the references are. `xs` is of type `&mut [&'a T]`, read as a mutable reference to a slice (list) of "references to `T` with lifetime `'a`". Note how `foo` is also parameterized with `<'a, T>`, making it *generic* over the type `T` and the lifetime `'a`. Don't be confused by the notation here: `'a` denotes a lifetime in Rust even though it looks like a ML type variable (the type variable here is just `T`).

This function being generic over both `'a` and `T` means that whatever the type `T` is, and however long the references are supposed to last, the function will operate over those references. In particular, the typechecker is able to prove that we are not placing pointers to other data types, or pointers that end up dangling, into the list. In this case, we are merely duplicating an element already in the list, which is clearly okay.

But the requirement currently is quite strong: `xs` only accepts references of lifetime `'a`. That means we have to be extremely particular, and it's hard to see what we might even be able to put into `xs` other than duplicates of what is already inside! To solve this problem, we revisit *subtyping*.

**Required Task 2.** We will represent the subtyping of lifetimes using the following judgment:

$$\texttt{'a} <: \texttt{'b}$$

which states that `'a` is a subtype of `'b` satisfying subsumption (any `'a` can be used in place of `'b`). Suppose we had a long lifetime `'long` and a short lifetime `'short`. A concrete example of the long and short lifetimes may be:

```
{
    let x: T = T::new();    // Create a T
    let x_ref: &T = &x;     // Obtain a &'long T reference to x
    {
        let y: T = T::new();    // Create a T
        let y_ref: &T = &y;     // Obtain a &'short T reference to y
        // Consider 'long and 'short at this point
    }
}
```

Which of these two relations is true?

$$\texttt{'long} <: \texttt{'short}$$
$$\texttt{'short} <: \texttt{'long}$$

**Required Task 3.** In the context of subtyping, there is a notion of bottom type, the type that can be safely upcast to any other. Give a compelling description of a bottom type when it comes to lifetimes. There is a precise answer to this question.

*Hint:* What are the different storage classes of data in C? Feel free to look it up.

**Fun Task 4.** Is there a compelling notion of top type? Can you think of one?

## 4 Linear Programming

In class, we saw examples of programs in the linearly-typed lambda calculus. Really it was the affine-typed version, since discarding variables is not considered too harmful, and the result looks much like Standard ML without structural sharing.

As a refresher, we argued that if we wanted to implement

```
val clone : 'a list -> 'a list * 'a list
```

the following would *not* be a correct implementation:

```
fun clone xs = (xs, xs)
```

since it uses `xs` twice in the manner of sharing. Philosophically, it is unsatisfying because it does not allow us to predict the usage of a resource like memory. It claims to clone a list, but really just returns two references to the same list, and given a collection of lists we cannot predict how much memory is used by the system.

Instead, we needed a helper function to clone the elements:

```
val f : 'a -> 'a * 'a
```

and we traversed the entire list, duplicating at each level.

We will now write some more programs in the affine-typed lambda calculus.

**Required Task 5.** Implement the following:

```
val append : 'a list -> 'a list -> 'a list
```

and characterize its heap usage in terms of the size of the input. Assume that `[]`, `::`, and any instance of `'a` take up one heap cell of space. Remember that the input already makes available some number of heap cells.

*Hint:* This shouldn't be difficult. But can you get the constants exactly right?

**Crazy Task 6.** Implement the following:

```
val mergesort : 'a list -> ('a -> 'a -> order) -> 'a list
```

and provide the heap usage, if you are really crazy.

*Hint:* `http://raml.co`