Forest Flame Writeup

David Krajewski and Micah Reich

November 2024

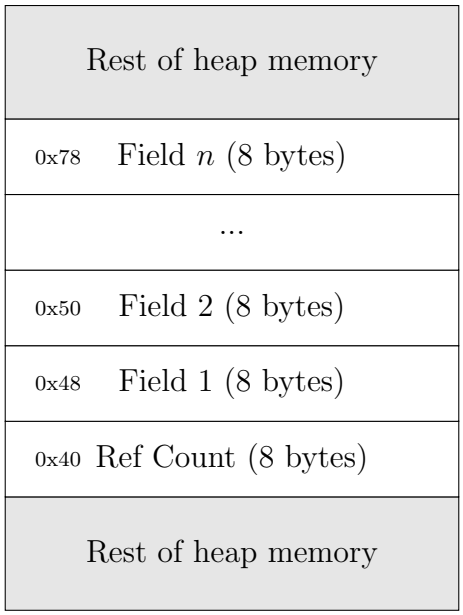
1 Garbage Collector Implementation

1.1 Functionality

For this assignment, we chose to implement garbage collection via reference counting. In the previous programming assignment, we implemented dynamic allocation with the `record` construct which serves as a general-purpose way to allocate user-defined record types on the heap which can store primitives (i.e. `int`, `bool` types) as well as pointers to other records. In addition, heap allocation does not have to be static allocation at compile-time in our language, giving lots of flexibility to the programmer to create complex datatypes, including recursively-defined records like linked lists and trees.

1.2 Tracking Reference Counts

In order to implement GC via reference counting, we augment each heap-allocated record with an extra 8-byte field in the beginning which stores its reference count. See the heap diagram below for how a record is laid out in memory:



1.2.1 Reference Count Increment

To implement GC, we use the notion of a write barrier to let the compilation know whenever a compiled expression is going to be assigned directly to another variable, and therefore whenever the reference count should be updated by 1. Ultimately, we want to "recursively" carry forward the notion of assignment, so that the compiler knows that if we return a pointer type and this pointer is being assigned to another variable, the reference count should increase. Take, for example this scenario involving a linked list:

```
(record node ((val int) (next node))

(let ((list_head (node 1 NULL)) (x 5))
    (= (lookup list_head val) x)
)
```

To manage the reference count of the newly created/allocated linked list node, we first note that this pointer is being assigned to the variable `list_head` inside the let binding, so we should increment its reference count by 1. When the let expression is over with, we should decrement the reference count of the list head pointer since `list_head` goes out of scope. Ultimately, this carrying forward of the assignment is done at runtime, and we store whether or not the current expression being evaluated is going to be assigned to another variable in the register `$rbx`. For two examples of how this check is created, used, and maintained, please look at the compilation of `ExprType::RecordInitializer` and `ExprType::Let` in the `compile_to_instrs` function in `compilation.rs`. For the base case, we start the program off with a carry forward value of 0, setting `$rbx` to 0 since we ultimately want all memory to be freed. We increment the reference count and set the carry forward flag to true whenever we are assigning an expression which evaluates to a pointer type to a local variable (or writing the evaluated pointer to a field in another record) and whenever we pass a record as a function argument.

1.2.2 Reference Count Decrement

For decrementing the reference count when variables go out of scope, we auto-generate functions in assembly. For each record type defined, we generate a function `record_name_rc_decr` which expects a memory address of a record to be passed in via `$rdi` and does the following:

- Step 1: decrement the reference count by 1
- Step 2: if the reference count after decrement is still greater than 0, return with no further action
- Step 3: otherwise, we must decrement the reference count of any other records which this one points to; using the known signature of the record, iterate through any fields which are record types and call their respective `field_record_name_rc_decr` functions
- Step 4: call `free` on the passed in pointer since the reference count has reached 0, and return

This construction allows us to recursively decrement and free pointers; as clarification, we

generate one such function per record type definition, so in a program which defines a linked list and BST node, you would have one reference count decrement function for a generic linked list node, and one for a generic BST node. See the function `compile_record_rc_decr_function_to_instrs` in `compilation.rs` for more information on this.

2 Tests

For each of the following tests you can run them by running:

```
make tests/TEST_NAME.run && ./tests/TEST_NAME.run <input> <max_heap_size>
```

Note: If you want to specify the max heap size, you must provide 2 arguments, the first being the input; otherwise, 1 provided argument will be interpreted as the input.

2.1 Add n to linked list

The test is named `add_ll` and takes 1 argument specifying the integer n to add to all values in the linked list.

2.2 Insertion into BST

The test is named `insert_bst` and takes 1 argument specifying the integer to insert.

2.3 10x Heap-size allocation

The test is named `heap_over_alloc`, and you should specify the max heap size to be 50 8-byte words (400 bytes total).

2.4 Out of memory

The test is named `oom`, and you should specify the max heap size to be 50 8-byte words (400 bytes total).