Egg Eater Writeup

David Krajewski and Micah Reich

October 2024

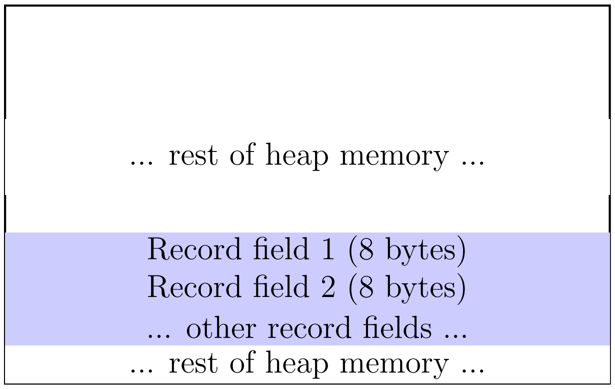
1 Grammar

The concrete syntax of our language is as follows:

```
<prog> := <defn>* <expr>
<defn> := (fun <fun_name> ((<name> <type>)*) <type> <expr>)
<record_defn> := (record <record_name> ((<field_name> <type>)+)) (new!)

<expr> :=
  | <number>
  | true
  | false
  | input
  | <identifier>
  | (let (<binding>+) <expr>)
  | (<op1> <expr>)
  | (<op2> <expr> <expr>)
  | (set! <name> <expr>)
  | (if <expr> <expr> <expr>)
  | (block <expr>+)
  | (repeat-until <expr> <expr>)
  | (<name> <expr>*)
  | (lookup <expr> <field_name>)           (new!)

<op1> := add1 | sub1 | print
<op2> := + | - | * | < | > | >= | <= | =

<type> := int | bool | record_name

<binding> := (<identifier> <expr>)
```

Beyond Diamondback, we add the `<record_defn>` construct to define a record type. Instantiating a record type has the same as calling a function, i.e. you use the name of the record type followed by a list of one or more expressions which each evaluate to a field. We also add the `lookup` expression which takes as argument an expression which evaluates to a record type as well as a string indicating which field name to lookup.

2 Heap-Diagram


... rest of heap memory ...
Record field 1 (8 bytes)
Record field 2 (8 bytes)
... other record fields ...
... rest of heap memory ...

When allocating values to the heap, our assembly calls malloc, which will allocate each field next to one another in 8 byte increments. Then internally a record is stored as a pointer to the first field in the heap.

3 Test Cases

3.1 simple_examples

```
(record Vec2 ((x int) (y int)))
(record Point ((pos Vec2) (value int)))

(fun set_pos ((p Point) (pos Vec2)) Point
    (Point pos (lookup p value))
)

(fun get_pos ((p Point)) Vec2
    (lookup p pos)
)

(fun set_value ((p Point) (value int)) Point
    (Point (lookup p pos) value)
)

(fun get_value ((p Point)) Vec2
    (lookup p value)
)

(fun max_value ((p1 Point) (p2 Point)) int
    (if (> (lookup p1 value) (lookup p2 value))
        (lookup p1 value)
        (lookup p2 value)
    )
)

(let
    ((p1 (Point (Vec2 0 0) 255)) (p2 (Point (Vec2 1 1) 127)))
    (block
        (print (max_value p1 p2))
        (let
            ((p1 (set_value p1 0)))
            (print (max_value p1 p2))
        )
    )
)
```

This example showcases creating points that have some position and value. The position field is actually a pointer to another record, Vec2, and the value represents an int. We defined different getter/setter functions as well as a ”max_value” function to show the different ways of creating and accessing heap allocated data. Inside our main block at the bottom, we define 2 points, print the max value of both (which should give us the first point's value). We then set the value of the first point to 0 and re print the max value of both, this time giving us the value of the second point. The output of this program is the following:

```
255
127
127
```

This is the intended behavior, the first 2 lines being the prints and then the last line being the overall value returned from the program (which was the last thing printed).

3.2 error_alloc

```
(record Vec2 ((x int) (y int)))

(fun add_vec ((a Vec2) (b Vec2)) Vec2
    (Vec2 (+ (lookup a x) (lookup b x)) (+ (lookup  a y) (lookup b y)))
)

(
    let
    ((a (Vec2 1 1 1 1)) (b (Vec2 2 2)))
    (add_vec a b)
)
```

This program defined a Vec2 record and attempts to initialize it with too many arguments (e.g. 4 numbers instead of 2). We expect this to throw an error during parsing, which it correctly does with the following message:

```
Too many arguments when initializing: Vec2
```

3.3 error_read

```
(record Vec2 ((x int) (y int)))

(
    let
    ((a (Vec2 1 1)))
    (lookup a z )
)
```

In this program, we create a new Vec2 and attempt to access a field, z, which does not exist. We expect this to fail during compilation, which it does with the error:

```
Invalid field lookup: field z not found in record Vec2
```

3.4 error3

```
(record Vec2 ((x int) (y int)))

(let
    ((vec (Vec2)))
    (5)
)
```

Similar to the alloc error above, this program tries instantiating a Vec2 with no arguments. In this case, we get a different error message:

```
Too few arguments when initializing: Vec2
```

This is what we expect since 0 arguments were passed as opposed to the 2 required.

3.5 points

```
(record Vec2 ((x int) (y int)))

(fun construct_vec ((x int) (y int)) Vec2
    (Vec2 x y)
)

(fun add_vec ((a Vec2) (b Vec2)) Vec2
    (Vec2 (+ (lookup a x) (lookup b x)) (+ (lookup a y) (lookup b y)))
)

(let
    ((a (construct_vec 1 1)) (b (construct_vec 2 2)) (c (add_vec a b)))
    (block
        (print a)
        (print b)
        (print c)
    )
)
```

This program defines a Vec2 record and a few functions with it. We first define ‘construct_vec‘ to build a Vec2 from two ints, x and y. We also define an ‘add_vec‘ function to add 2 vectors together. Our main block of code at the bottom constructs 2 vectors, a and b, then adds them together to create a third vector c. We then print each vector sequentially. We would expect this to print the values of each vector, then to print c again since it is the return value of the function. Our compiler yields an executable that gives the following output:

```
(
1
1
)
(
2
2
)
(
3
3
)
(
3
3
)
```

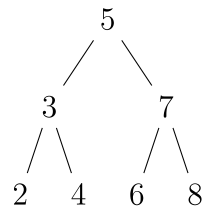
As you can see, parentheses group together the same vectors (i.e. values on the heap) and our output is the output we expect.

3.6 bst

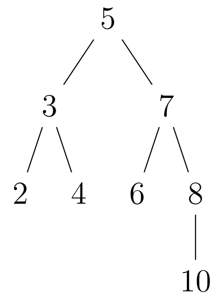
```
(record bst_node ((val int) (left bst_node) (right bst_node)))

(fun search_value ((val int) (root bst_node)) bool
    (if (= root NULL)
    (false)
    (let
        ((curr_val (lookup root val)))
        (if
            (= curr_val val)
            (true)
            (if (< val curr_val)
                (search_value val (lookup root left))
                (search_value val (lookup root right))
            )

        )
    )
    )
)

(fun insert_value ((val int) (root bst_node)) bst_node
    (if
        (= root NULL)
        (bst_node val NULL NULL)
        (let
            ((curr_val (lookup root val)))
            (if
                (< val curr_val)
                (bst_node (lookup root val) (insert_value val (lookup root left)) (looku
                (bst_node (lookup root val) (lookup root left) (insert_value val (lookup
            )
        )
    )

)

(let
    ((root (bst_node 5 (bst_node 3 (bst_node 2 NULL NULL) (bst_node 4 NULL NULL)) (bst_n
    (
        block
        (print (search_value 10 root))
        (let
            ((new_root (insert_value 10 root)))
            (block
                (print (search_value 10 new_root))
                (print (lookup (lookup (lookup new_root right) right) right))
                (print (search_value 2 new_root))
                (print (search_value 20 new_root))
            )
        )
    )
)
```

This BST program defines a record for a BST node (containing a value and child pointers), a search value function which returns a boolean if the provided value is found in the tree, and an insert value function which inserts the given value into the tree. Our language does not support field assignment for records, so the insertion function will copy the tree and return the root node of the newly created tree.

Our test case at the bottom of the program begins by constructing a tree that looks like the following:



It then searches for the value 10 in the tree, which should return false, and proceeds by inserting 10 into the tree and saving the root of this new tree into a value called "new_root". This yields a new tree that looks like:



We confirm that this worked by searching for 10 in this new tree, which should return true, and also traversing 3 right branches of our tree, which should give us 10. Finally as a sanity check to ensure our tree was copied over correctly we search for a known value, 2, and a value we know is not in the tree, 20. Thus, the result of our program should be false, true, the node containing 10, true, false, and one last false since our program returns false. This is what our program actually returns:

```
false
true
(
10
{pointer}
{pointer}
)
true
false
false
```

which is exactly as we expected.

4 Programming Language Examples

We choose to compare Egg-Eater against C and Python. Our language's implementation of records is more similar to that of C's, since in C, the programmer can declare a struct with field name and field type pairs and can index into the struct using the field names; further, in C, structs and their fields are strongly typed. While Python has a notion of structs with the use of the `@dataclass` decorator, these are mostly just named tuples. Alternatively, many programmers just use a Python class to act as a struct or record. As you can see, the lack of a native struct type in Python and lack of strong typing within the class or dataclass makes Egg-Eater more similar to C; what's more, in Python, one can change the type of a field within a dataclass, class, or tuple, unlike in Egg-Eater.

5 Resources

We referenced the following resources for this assignment:

- Python dataclasses documentation: https://docs.python.org/3/library/dataclasses.html
- How to link against libc: https://stackoverflow.com/questions/26277283/gcc-linking-libc-static-and-some-other-library-dynamically-revisited