Egg Eater Design Document

CSE 231: Programming Assignment 6

Emmanuel Anaya Gonzalez
PID: *A59016399*

The concrete syntax for this implementation of Egg Eater extends Diamondback with two new constructors for representing `tuple`s of a fixed size and a primitive for accessing them at a given `index`.

```
<prog> := <defn>* <expr>
<defn> := (fun (<name> <name>*) <expr>)
<expr> :=
  | <number>
  | true
  | false
  | input
  | <identifier>
  | (let (<binding>+) <expr>)
  | (<op1> <expr>)
  | (<op2> <expr> <expr>)
  | (set! <name> <expr>)
  | (if <expr> <expr> <expr>)
  | (block <expr>+)
  | (loop <expr>)
  | (break <expr>)
  | (tuple <expr>*)                    (new!)
  | (index <expr> <expr>)              (new!)
  | (<name> <expr>*)

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

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

Note that we allow tuples of size 0.

Tuples are allocated in the heap, and a n-tuple is stored using $n + 1$ contiguous 8-byte locations. The first word of the is used for storing the length of the tuple. So if a n-tuple is stored at address `a` in the heap, the region `[a, a + 8, ..., a + n * 8]` looks like `[size, p1, ..., pn]` where `pi` are the elements of a the tuple. `(tuple <expr>*)` evaluates to the address in the heap where the tuple is allocated, encoded with LSBs `01` to tag them as tuples. For `(index <expr> <expr>)`, the first expression must evaluate to a tuple,

and the second to a number. Tuples a are 0-indexed.

Tests

1. simple_examples

```
(let ((t (tuple 1 10 100)))
  (index t input))
```

This simple test accesses a tuple of three elements at the index given as `input`. For example, running `simple_examples.run 1` outputs `10`.

2. error-tag

```
(index false 0)
```

This test errors at runtime due to `false` not being a `tuple` that can be indexed.

3. error-bounds

4. error3

```
(index (tuple 1 2 3) false)
```

This test errors at runtime due to `false` not being a valid index to query the tuple.

5. points

```
(fun (to_point x y)
  (tuple x y))
(fun (add_points p1 p2)
  (let ((xnew (+ (index p1 0) (index p2 0)))
        (ynew (+ (index p1 1) (index p2 1))))
    (to_point xnew ynew)))
(let ((a input)
      (b (* input 2))
      (p3 (add_points (to_point a b)
                      (to_point (* a 10) (* b 10)))))
  (block
```

```
(print (index p3 0))
(print (index p3 1))
0))
```

This function uses the `to_input` and `add_points` functions to implement adding element-wise addition of numeric pairs. For example, running `points.run 3` outputs

```
33
66
0
```

If we were to compare the heap memory management of Egg Eater to other programming languages, for example `C` and `Python`, we can say that our design is more like the latter. Memory for heap-allocated objects is handled automatically, without the need of special directives such as `malloc` and `free`. Additionally, similar to `Python` our language provides an immediate way to create objects that reference each other or even themselves.

References and collaborators

1. CSE231 lecture code. `https://github.com/ucsd-compilers-s23/lecture1/blob/egg-eater/src/main.rs`
2. ChatGPT