UCSD CSE131 F19 – Garter

November 20, 2019

Checkpoint Due Date: 11pm Wednesday, November 27 **Final Due Date:** 11pm **Thursday** December 5

The specific features listed for the checkpoint are **Open to Collaboration** (detailed below), and the rest is **Closed to Collaboration**.

You will implement memory management atop a type-checked language with heap-allocated data and functions.

Classroom: FILL Github: FILL

Syntax

The concrete syntax and type language for Garter is below. We use $\cdots$ to indicate *zero or more* of the previous element. There are $\boxed{\text{boxes}}$ around the new pieces of concrete syntax.

$$
\begin{array}{lcl}
e & := & n \mid \texttt{true} \mid \texttt{false} \mid x \\
 & \mid & \texttt{(let ((}x\ e\texttt{)}\ \texttt{(}x\ e\texttt{)}\ \cdots\texttt{)}\ e\ e\cdots\texttt{)} \\
 & \mid & \texttt{(if}\ e\ e\ e\texttt{)} \\
 & \mid & \texttt{(}op_2\ e\ e\texttt{)} \mid \texttt{(}op_1\ e\texttt{)} \\
 & \mid & \texttt{(while}\ e\ e\ e\cdots\texttt{)} \mid \texttt{(set}\ x\ e\texttt{)} \\
 & \mid & \texttt{(}f\ e\cdots\texttt{)} \mid \boxed{\texttt{(null}\ \tau\texttt{)}} \\
 & \mid & \boxed{\texttt{(get}\ e\ n\texttt{)}} \mid \boxed{\texttt{(update}\ e\ n\ e\texttt{)}} \\
d & := & \texttt{(def}\ f\ \texttt{(}x\ \texttt{:}\ \tau\cdots\texttt{)}\ \texttt{:}\ \tau\ e\ e\cdots\texttt{)} \\
 & \mid & \boxed{\texttt{(data}\ C\ \texttt{(}\tau\cdots\texttt{))}} \\
p & := & d\cdots e \\
op_1 & := & \texttt{add1} \mid \texttt{sub1} \mid \texttt{isNum} \mid \texttt{isBool} \mid \texttt{print} \\
op_2 & := & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{<} \mid \texttt{>} \mid \texttt{==} \mid \boxed{\texttt{=}} \\
n & := & \text{63-bit signed number literals} \\
x, f, C & := & \text{variable, function, and constructor names}
\end{array}
$$

$$
\begin{array}{lcl}
\tau & := & \texttt{Num} \mid \texttt{Bool} \mid C \\
\delta & := & \texttt{fun} \mid \texttt{data} \\
\Delta & := & \{\delta\ f : \tau\cdots \to \tau, \cdots\} \\
\Delta[f] & \textit{means} & \textit{look up the type of } f \textit{ in } \Delta \\
\Gamma & := & \{x : \tau, \cdots\} \\
\Gamma[x] & \textit{means} & \textit{look up the type of } x \textit{ in } \Gamma \\
(x, \tau) :: \Gamma & \textit{means} & \textit{add } x \textit{ to } \Gamma \textit{ with type } \tau \\
\Delta; \Gamma \vdash e : \tau & \textit{means} & \textit{with definitions } \Delta \textit{ and env } \Gamma, e \textit{ has type } \tau \\
\Delta \vdash_d d : \checkmark & \textit{means} & \textit{with definitions } \Delta \textit{ the definition } d \textit{ type-checks} \\
\vdash_p p : \checkmark & \textit{means} & \textit{the program } p \textit{ type checks}
\end{array}
$$

Semantics

The semantics here are all provided for you, we describe them so you'll be able to write accurate tests.

Data Definitions, Construction, and Manipulation

The main new feature in Garter is data definitions `(data` C `(`$\tau\cdots$`))`, where s is the name of the data definition and the types $\tau\cdots$ are the types of the elements stored in instances of the data definition. Elements are accessed and updated positionally with *fixed* (not computed, as with arrays) numeric indices using `(get` e n`)` and `(update` e n e`)`.[1] The syntax for function applications is used to construct new data instances.

As an example, this program evaluates to 67:

```
(data Pair (Num Num))
(let (
    (p1 (Pair 4 5))
    (p2 (Pair 4 5))
    (p3 p1)
    )
  (update p1 0 11)
  (update p2 1 56)
  (+ (get p3 0) (get p2 1)))
```

Printing Data Instances

In Egg-Eater, locations (referring to instances of data) are a new kind of value that can be printed, just like numbers and booleans.

When an instance is printed, it should print in the format

$(C\ \ v_1\ \ v_2\ \ \cdots)$

Where C is the name of the constructor used to create it, and values v_1 and v_2 are the printed form of the values stored in its fields, separated by spaces.

For example:

```
(data Pair (Num Num))
(data PairOfPairs (Pair Pair))
(let ((p (PairOfPairs (Pair (+ 1 2) 6) (Pair (add1 6) 8))))
  p)

# prints:
(PairOfPairs (Pair 3 6) (Pair 7 8))
```

Equality

There two types of equality in Egg-Eater, reflecting the new nuances of heap-allocated data. The first, `==`, behaves as before on existing values, and on locations referring to instances of data, returns `true` if the *locations* are identical. The second, `=`, behaves as before on existing values, and on locations returns `true` if the two instances came from the same constructor and the *contents* of those locations are all equal according to `=`.

For example:

```
(data Pair (Num Num))
(data PairOfPair (Pair Pair))
(data Point (Num Num))
(let (
  (p1 (Pair 3 4))
  (p2 (Pair 3 4))
  (p3 (Point 3 4))
  (p4 (Point 3 5))
  (pp12 (PairOfPair p1 p2))
```

[1] As an analogy, data definitions are somewhat like structs in C, but use positional lookup instead of names; as another analogy, data definitions are like tuples in OCaml and we can match on them by statically known positions using functions like `fst` and `snd`, but not compute the position of lookup.

```
(pp21 (PairOfPair p2 p1))
)
(print (= p1 p2)) ; true, same constructor and contents
(print (== p1 p2)) ; false, different locations
(print (= p1 p3)) ; false, different constructors
(print (== p1 p3)) ; false, different locations
(print (= p3 p4)) ; false, different contents
(print (= pp12 pp21)) ; true, same (nested) contents
0
)
```

Type Checking

Egg-Eater has essentially the same type rules as Diamondback for expressions. The definitions environment Δ is constructed with the types of the constructors for data definitions as well as function definitions; these are distinguished by either `data` or `fun` before the name.[2] As an example, the definition `(data Point (Num Num))` would appear in Δ as `data Point` : (`Num Num` $\rightarrow$ `Point`). There is a new rule for each new syntactic form, except for `data` definitions which don't need separate type checking.

$$\textsc{TR-Null}\ \frac{\Delta[\texttt{data C}] = (\tau_1 \cdots \rightarrow \tau_r)}{\Delta;\Gamma \vdash \texttt{(null } C\texttt{)} : C}$$

$$\textsc{TR-Get}\ \frac{\Delta;\Gamma \vdash e : \mathrm{C} \qquad \Delta[\texttt{data C}] = (\tau_1 \cdots \tau_n \tau_{n+1} \cdots \rightarrow \tau_r)}{\Delta;\Gamma \vdash \texttt{(get } e\ n\texttt{)} : \tau_n}$$

$$\textsc{TR-Update}\ \frac{\Delta;\Gamma \vdash e : \mathrm{C} \qquad \Delta[\texttt{data C}] = (\tau_1 \cdots \tau_n \tau_{n+1} \cdots \rightarrow \tau_r) \qquad \Delta;\Gamma \vdash e_v : \tau_n}{\Delta;\Gamma \vdash \texttt{(update } e\ n\ e_v\texttt{)} : \tau_n}$$

There are a few important features here.

- The `null` expression comes with a type that it should be treated as. The type checker simply checks that this annotation is some `data` type and treats the `null` value as that type. This allows us to construct instances of recursively-defined datatypes like `(Link (Num Link))`.
- In TR-Get and TR-Update, we check that the first expression has a type of some data definition C. The types before the $\rightarrow$ are the types of the fields or elements listed in the data definition.
- We assume the existing rule for TR-App in applications, which simply checks that values with the right types are present in order according to the data definition (just like for function calls).

Application Binary Interface

Value and Heap Layout

The value layout is extended to keep track of information needed in garbage collection:

- `0xXXXXXXXXXXXXXXX[xxx1]` - Number
- `0x000000000000000[0110]` - True
- `0x000000000000000[0010]` - False

[2]As an implementation note, we found it useful to simply pass around the entire list of definitions in several functions.

- `0x0000000000000000[0000]` - Null
- `0xXXXXXXXXXXXXXXXX[x000]` - Data Reference, an address of a data instance on the heap laid out as follows (each set of `[]` is one 8-byte word)

  ```
  [ GC word ][ name reference ][ element count n ][ value 1 ][ value 2 ] ...  [ value n ]
  ```

The use of the GC word is completely up to your memory management implementation and is always initialized to 0 (see below). The name reference is the address of a C string that holds the struct's name (essentially a `char*`) used in printing and equality. The element count tracks the number of elements stored in the data value.

As an example, consider this program:

```
(data Pair (Num Num))
(let (
    (p1 (Pair 4 5))
    (p2 (Pair 6 7))
    (p3 p1)
    )
    ...)
```

The stack word for `p1` would hold a value like `0x00000000ABCDE120`, where at address `0x00000000ABCDE120` would be stored:

```
0x00000000ABCDE230 : [ 0x0000000000000000 ] ; gc word
                     [ 0x00000000NAMEADDR ] ; address of "Pair"
                     [ 0x0000000000000002 ] ; count of elements
                     [ 0x0000000000000009 ] ; representation of 4
                     [ 0x000000000000000B ] ; representation of 5
```

Where at `0xNAMEADDR` we would find the characters `Pair\0`, and 9 and 11 are the representations of 4 and 5. At the stack word for `p3` we would also find `0x00000000ABCDE120`. At the stack word for `p2` we should expect to find a different address, say `0x00000000ABCDE230`, with a similar layout but different values:

```
0x00000000ABCDE230 : [ 0x0000000000000000 ] ; gc word
                     [ 0x00000000NAMEADDR ] ; address of "Pair"
                     [ 0x0000000000000002 ] ; count of elements
                     [ 0x000000000000000D ] ; representation of 6
                     [ 0x000000000000000F ] ; representation of 7
```

Calling Convention

We use a calling convention similar to the one discussed in class, so at any given moment there are a number of function calls on the stack, each with arguments and local variables.

Some important highlights:

- On the right, we show the addresses stored in the arguments given to `try_gc` which are passed on to the `gc` function you will write. This includes `stack_top`, which is equal to `rsp - (stackloc si)`, `first_frame`, which is equal to `rsp`, and `STACK_BOTTOM`, which is a global that refers to the original value of `rsp` right after calling `our_code_starts_here`. We will say more about each of these in the next section.
- We made sure the compiler implements the invariant that `rsp - (stackloc si)` will always refer to the word above the topmost valid value, and that there won't be any invalid values in the local variables or the arguments on the stack.

```
        [   UNUSED SPACE    ] <- stack_top
        ---------------------
        [local var N        ]
        [...                ]
        [local var 1        ]   these locals and args are
        [arg 1              ]   for the topmost active
        [...                ]   function call
        [arg N              ]
        [prev rsp value     ]
rsp ->  [return address     ] <- first_frame
        ---------------------
                 ...
        ---------------------
        [local var N        ]   these locals and args are
        [...                ]   for a current active
        [local var 1        ]   function call
        [arg 1              ]
        [...                ]
        [arg N              ]
        [prev rsp value     ]
        [return address     ]
        ---------------------
        [local var N        ]  these locals are for
        [...                ]  the main expression
        [local var 1        ]
        [ret ptr to main    ] <- STACK_BOTTOM
```