UCSD CSE131 F19 – Garter Snake

November 25, 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: `https://classroom.github.com/a/o44AGgQq` Github: `https://github.com/ucsd-cse131-f19/pa7-student/`

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 $\texttt{(data}\ C\ \texttt{(}\tau\cdots\texttt{))}$, 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 Garter, 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 prints 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 Garter, 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))
  (pp21 (PairOfPair p2 p1))
```

[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.

```
)
(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

Garter 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. 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.

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

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

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

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 (ABI)

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
- `0x000000000000000[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 ][ element count n ][ name reference ][ 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`, and at that address would be stored:

```
0x00000000ABCDE120 : [ 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 B 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 - 1))`, `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 - 1))` 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.

Command Line Options

The binaries generated from the Garter compiler have *three* command line arguments:

```
        ---------------------
        [local var N        ] <- stack_top
        [...                ]
        [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        ]
        [ stored rbx value  ]
        [ret ptr to main    ] <- STACK_BOTTOM
```

```
$ ./output/program.run <heap size in words> <input> <dump>
```

The defaults are a 10000 word heap, input equal to 1, and no dumping of the heap on exit. If a heap size is specified, that argument is given to `setup_heap` instead of 10000. If an input is specified, it's expected to be a number (as in past assignments). The final parameter can only be the string `dump`, and its behavior is described below.

Memory/GC Interface and Features

The sections above detail how the Garter language works and lays out memory. This section details the specific tasks in the assignment including the necessary interfaces and places to add code.

Printing Memory (Checkpoint)

In many cases, it's helpful to print out the state of the heap to aid debugging. While you can and should use tools like `gdb` and `lldb` to do this, it's also helpful to build some of our own infrastructure.

To support your debugging efforts, you will implement a function that prints the heap. You can use it however you like in debugging, and it will also be used if the special third argument `"dump"` is given to a compiled binary, once the program is complete (e.g. after `our_code_starts_here` returns), the `print_heap` function is called with the current value of `HEAP_START` and the final value of `r15`.

For example, for the input program

```
(data Point (Num Num))
(data PLink (Point PLink))

(let ((p1 (Point 4 5))
      (p2 (PLink p1 (null PLink))))
  p2)
```

A version we wrote shows data like this:

```
$ ./output/twostructs.run 100 0 dump
(PLink (Point 4 5) null)
HEAP_START------------------------------------
----------------------------------------------
0x1002021c0 | GC   - 0
0x1002021c8 | Size - 2
0x1002021d0 | Name - Point
0x1002021d8 | [0]  - 4
0x1002021e0 | [1]  - 5
----------------------------------------------
0x1002021e8 | GC   - 0
0x1002021f0 | Size - 2
0x1002021f8 | Name - PLink
0x100202200 | [0]  - 0x1002021c0
0x100202208 | [1]  - null
HEAP_END--------------------------------------
```

Here the heap started at `0x1002021c0` and the two values are printed with some rich structure – we used what we knew about the value layout to print numbers and null as their actual value (rather than the representation), to label the size and name, and so on.

For the checkpoint, you should submit a PDF report with:

- The C code for your printing implementation **included in the PDF in monospaced font**.
- An example program and its output with `dump` that includes on the heap at least two different sizes of data instance with at least 3 total instances on the heap, using each kind of value (address, null, true/false, and numbers) with how you chose to print them.

This checkpoint is due early to encourage you to write some test programs and work with the heap representation early. That said, we do **not** encourage you to believe that you're on a good pace to complete the assignment if you only complete printing by the checkpoint deadline. We really recommend moving on beyond this as soon as you can.

Since this part of the assignment is open to collaboration, feel free to share cool ideas and tricks you used to implement your printing with one another, especially as you practice and recall details of working with pointers and heap representations.

Note that the starter code defines both `print_heap` and `print_stack` – we found it useful to print and format these differently. For the checkpoint, we're only grading based on `print_heap`'s behavior on the dump at the end of the program.

Garbage Collection (Main Assignment)

Runtime Interaction

The generated code from Garter checks, on each allocation of a new `data` value, if enough space is left on the heap to fit that value. It does this by comparing the value in `r15` to the value stored in `HEAP_END`, which is a global variable set by `main`. If enough space isn't available, the generated code calls the `try_gc` function with several arguments:

- `alloc_ptr` – the current value in `r15`
- `words_needed` – the number of words needed by the allocation

- `first_frame` – the address of the stored `rsp` in the topmost (most-recently-called) stack frame of a Garter function (also the most recent value of `rsp` used by Garter)
- `stack_top` – the address of the top word in the stack used by Garter; also equal to `rsp - (stackloc si)` at the time of allocation.

The goal of `try_gc` is to trigger garbage collection, and check if (after the gc step) enough memory is available for the attempted allocation. You shouldn't need to change `try_gc` at all unless you're trying something quite exotic, and you can't change the way the runtime calls it (because you can't change the generated code).

The GC Function

The `try_gc` function passes along information to the `gc` function, which you will implement. The `gc` function is passed:

- `STACK_BOTTOM`, which is the address of the bottom-most word on the Garter stack, which corresponds to the return pointer to `main` used at the end of `our_code_starts_here`.
- `first_frame` and `stack_top` as provided to `try_gc`
- The current value stored in `HEAP_START`. `HEAP_START` is initially set by `setup_heap` on program startup; you can change `setup_heap` if you need to (subject to the constraints below). Your program can (but does not need to) manipulate this variable.
- The current value stored in `HEAP_END`. `HEAP_END` is initially set by `setup_heap` on program startup; you can change `setup_heap` if you need to (subject to the constraints below). Your program can (but does not need to) manipulate this variable.

The `gc` function is expected to return a new value for the heap pointer after doing any work needed to free up space on the heap. You can choose any strategy for collecting garbage, subject to the following constraints:

- The generated code will always allocate by putting values at `r15` and incrementing `r15`, so your strategy must leave a contiguous region of memory for the generated code to use.
- Your strategy should allow for programs to use as much live data as the number of words given as the command-line argument to the binary. This means that, for example, if you choose semispace swapping as your allocation strategy, you may want to allocate *twice* the space requested so that you can actually fit the requested live space.
- Your strategy must run out of memory if the program tries to allocate *more* than the amount of words given at the command line.
- Your strategy should not allocate heap space any larger than *three times* the size of the number of words requested. So you can do twice for semispace swapping, or even more if you like, but you cannot dramatically over-allocate.

Two straightforward choices are to implement mark/compact and semispace swapping – the lectures and the readings have several descriptions of the required use of metadata and algorithms.

Implementation Conventions and Recommendations

The variables in `ALL_CAPS` are globals in C. You can alter the setup for `HEAP_START` and `HEAP_END` as needed, and manipulate them as necessary during garbage collection. `FINAL_HEAP_PTR` is used by the generated code and `main` to communicate to the heap printer where to stop printing, and you shouldn't need to change it. `STACK_BOTTOM` is set by the generated code to the very bottom of the stack.

We provide a struct definition `Data`:

```
typedef struct {
  int64_t gc_metadata;
  int64_t size;
  char* name;
  int64_t elements[];
} Data;
```

We found it useful to cast pointers into the heap into `Data*` in a number of cases – you can see the existing print and equality functions for examples of this use.

You can use

```
$ make output/program.run
```

as with past assignments. Do feel free to use tools that may be available on your system, like `valgrind`, to check for memory issues as you debug, for example on the department lab machines you can use

```
$ valgrind output/program.run
```

to check that the program ends with no memory errors.

You can also test your garbage collector by adding tests to the `test.ml` file, where there are new test options that allow you to specify heap sizes as part of your test (see the given examples).

Handin and Grading

As with past assignments, you will hand in both code and a short report. The majority of your credit will come from automated grading of your code on this assignment, though there are also some open-ended questions we're interested in:

- 80% – auto-grading tests (via `pa8`, some run after final handin)
- 20% – report submitted to `pa8-written`
  1. 8% – Give an example (as Garter code) of a long-running program that allocates a lot of memory that quickly becomes dead across several garbage collections. Describe your collector's behavior on that program using a few interesting moments during its execution.
  2. 8% – Give an example (as Garter code) of a program that allocates some memory early on that is live throughout the entire program's execution, even though some other values are allocated that are later collected as garbage. Give a brief description of how much work your collector does handling those long-lived live values across the run of the program.
  3. 4% – Describe something you learned about C programming by completing this assignment.