CSC 526, Spring 2020, Assignment 7

Purpose: Heap allocation

Due: 11:59pm, Wednesday, April 22, 2020

Get the assignment code

These instructions assume that your course git repository is set up. Change into your course repository directory and enter the following commands.

```
git fetch assignments
git checkout assignments/master -- assignment7
git add assignment7
git commit -a
```

This will copy the `assignment7` directory into your working directory, start tracking the files in the `assignment7` directory, and commit those files to your local git repository.

Assignment Description

The purpose of this assignment is to augment the previous assignment with a heap allocated tuple type.

Syntax Additions

```
<expr> :=
    ...
    | (tup <expr list>)
    | (tup-len <expr>)
    | (tup-get <expr> <expr>)
    | (is-tup <expr>)
```

Tuple Heap Layout

Tuple expressions should evaluate their sub-expressions in order, and store the resulting values on the heap. The layout for a tuple on the heap is:

```
(first 8 bytes)   (8 bytes)   (8 bytes)   ...   (8 bytes)
+---------------+-----------+-----------+-----+-----------+
| # of elements | element 0 | element 1 | ... | element n |
+---------------+-----------+-----------+-----+-----------+
```

The first word is used to store the number of elements in the tuple and the subsequent words are used to store the values.

A tuple value is stored in variables and registers as the address of the first word in the tuple's memory, but with an additional `00` added to the value to act as a tag. With this change, the set of tag bits is extended to the following:

- Integers: `1` in the least significant bit
- Booleans: `10` in the least two significant bits
- Tuples: `00` in the least two significant bits

Visualized the type layout is:

```
0bBBBBBBBBBBBB[BBB1] -- integer
0b000000000000[0110] -- true
0b000000000000[0010] -- false
0bBBBBBBBBBBBB[BB00] -- tuple
```

Where `B` is a wildcard bit.

Accessing Tuple Contents

A tuple access expression, like

```
(tup-get (tup 2 4 6 8) 1)
```

The behavior should be:

1. Evaluate the tuple expression (the expression in the parentheses), and then the expression that follows the tuple.
2. Check that the tuple is actually a tuple by looking for the appropriate tag bits. If not, then call the error function with the appropriate error code.
3. Check that the index value is an integer. If not, then call the error function with the appropriate error code.
4. Check that the index integer is a valid index for the tuple value, that is, between zero and the number of elements minus one. If not, then call the error function with the appropriate error code.
5. Evaluate to the tuple element at the specified index.

General Heap Layout

Choose a register to use as a heap pointer. The `runtime.c` file executes a `calloc` call to allocate space for the heap. The resulting address is passed to the `code_entry_point` as an argument. This value should be stored in your chosen register. Since the memory address is arbitrary, you should initialize your chosen register to be aligned to the first location that has the tuple tag in the two least significant bits. Here is one way to do so, using register `%rbx`:

```
addq $8, %rbx
andq $-4, %rbx ; hex FFFFFFFC
```

It is your responsibility to ensure that the value of your chosen register is always the address of the next block of free space (in increasing address order) in the provided block of memory.

Interaction with Existing Features

When we add a new feature to a language, we need to consider its interactions with all the existing features. In our case, the existing features are:

- **Function calls and definitions**: Tuple values behave like other values when passed to and returned from functions. The tuple value is a tagged address that takes up a single word.
- **Let bindings**: Tuple values take up a single word and act like other values in let bindings
- **Binary operators**: The arithmetic expressions should continue to only allow integers. There is one binary operator that does not check types, namely `==`. We need to decide what the `==` operator does with tuple values as operands. This program fragment should evaluate to `true`:

  ```
  (let ((t (tup 1 2))) (== t t))
  ```

  However, we need to decide if

  ```
  (== (tup 1 2) (tup 1 2))
  ```

  should evaluate to `true` or `false`. That is, do we check if the tuple addresses are the same to determine equality, or if the tuple contents are the same. For this assignment, we will do the easier of the two and simply compare the addresses of tuples.
- **Unary operators**: The behavior of the unary operators is straightforward with the exception of the `print`. We could print the addresses, but that is not satisfying. Instead, we should recursively print the tuple contents. This is implemented for you in the `runtime.c` file.

Error Checking

The error checking of the previous assignments should be extended to perform a runtime type check for the `==` operator. This should check that both operands are the same type.

Turning in the Assignment

To turn in the assignment execute the following git commands from within your repository:

```
git add <file>
git commit -a
git push origin master
```

where a `git add <file>` command is needed for every file that is required for building the assignment executable. Failure to add any required files will result in a failing grade for the