UCSD CSE131 F19 – PA6

November 15, 2019

Due Date: 11pm Wednesday, November 20 **Open to Collaboration**

Starter code: Start where PA5 left off, or use the PA5 starter code, or use your PA4 implementation. (See below).

Requirements

You have two choices in this assignment: extend PA5 with more heap allocation features or implement some compiler optimizations. You can only hand in one of these; if for some reason you hand in to multiple versions of the Gradescope assignment, you **must** tell us which one you want us to use before the assignment deadline.

Extend Heap Allocation

You can submit PA6 as a completion/extension of PA5. For this version of the PA, you must satisfy all the same requirements as PA5, with the addition of the following required goals:

- Implement both **structural equality** and **reference equality** on the heap allocated data you designed. As a reminder, **structural equality** means comparing the contents of heap values, not just their references.
- Make sure that **structural equality** and **printing heap values** does not result in an infinite loop or stack overflow, but instead returns a meaningful value or reports a meaningful error when a cycle is detected. (The simplest thing is to detect a cycle and report an error).
- Write a new test `input/equal.boa` that demonstrates how structural and reference equality work in your language on non-cyclic values. Make sure to include enough examples, including cases that return true and false, to demonstrate the behavior thoroughly.
- Write a new test file `input/cycle-print.boa` that demonstrates how printing works in your language on cyclic values.
- Write a new test `input/cycle-equal.boa` that demonstrates how structural equality works in your language on cyclic values.

You will hand in your code to `pa6-heap-code` and a writeup to `pa6-heap-written`.

If you submit this option, **include your entire writeup for PA5, which can be updated as much as you like, with an added section containing the elements below**. Include:

1. (15%) A description of your approach to handling structural equality, including relevant snippets of C or generated assembly code.
2. (15%) A description of your approach to handling cycles, including relevant snippets of C or generated assembly code.
3. (20% each) In the test section with the description of each of the three new tests, the actual output of the generated binary when run (in terms of stdout/stderr), the output youd like it to have (if theres any difference) and any notes on interesting features of that output. Graded on completeness and if your implementation matches the desired behavior.
4. (5%) A description of features other than structural equality and handling cycles that you improved since your PA5 submission.

5. (5%) A list of the resources you used to complete the assignment, including message board posts, online resources (including resources outside the course readings like Stack Overflow or blog posts with design ideas), and students or course staff discussions you had in-person. Please do collaborate **and give credit to your collaborators**.

Grading

If you choose this option, we will grade **only your PA6 submission**, and its grade on the parts related to PA5 will count fully for your PA5 submission, with separate credit for PA6 based on the new features.

You may want to choose this option if you felt your submission for PA5 was something you could improve upon and you'd like to explore and finish that effort, while earning credit for it. You may also simply be more interested in the structural algorithms relating to equality, printing, and cycles.

Implementation Recommendations

Structural equality is related to the recursive printing required for PA5, but takes two arguments and compares them. Remember that if two values have different *tags*, they cannot be equal.

To detect cycles, you may find it useful to store some extra metatdata for heap allocated values. This extra metadata can be initialized to 0, and be used during the equality and printing algorithms to track whether a particular heap-allocated value has already been printed or checked for equality during the current traversal of values. Note that since a value could be printed or checked for equality more than once, this data should be set back to 0 after checking equality or printing.

Implement Compiler Optimizations

You will implement several compiler optimizations.

Required Features

Your compiler must support:

- `input` as in past assignments and variable assignment (the `set` expression)
- Three optimizations:
  - Constant propagation (CP) of constant variables
  - Constant folding (CF) of (at least) arithmetic
  - Dead code elimination (DCE) for (at least) if expressions and while loops
- A fixpointing optimization that optimizes using all of CP, CF, and DCE

Required Tests

You **must** write the following programs to test your compiler.

- `input/cp.boa` A program with at least three instances of code that is interesting for constant propagation, including at least one instance where a constant is *not* propagated because it's the value of a variable that is assigned to with `set` later on.
- `input/cf.boa` – A program with at least three instances of code that is interesting for constant folding and where your optimization is applied.
- `input/dce.boa` – A program with at least three instances of code that is interesting for dead code elimination and where your optimization is applied.
- `input/combined.boa` – A program that demonstrates the fixpointing of all three optimizations where in at least two cases applying one optimization enables another (so all three are eventually used).

Handin, Report, and Grading

You will submit your implementation to `pa6-opt`, which will run a subset of the PA4 grader tests to make sure your compiler hasn't broken any features with the optimizations. You'll also submit a PDF to `pa6-opt-written` containing the report questions below. Your grade will be calculated from:

- (5%) Continuing to pass the tests from PA4.
- (10%) A description of how you implemented constant propagation, including how you avoid propagating a constant that might later be changed by a variable assignment. Include snippets of OCaml code as necessary.
- (10%) A description of how you implemented constant folding, including snippets of OCaml code as necessary. Argue in 2-3 sentences why your optimization will not change the *behavior* of any programs,
- (10%) A description of how you implemented dead code elimination, including snippets of OCaml code as necessary.
- (15% each) For each of the required 4 tests, include the interestingly optimized part of the generated assembly by first running the compiler *without* your optimizations, then with them, and point out what changed. Write a sentence or two about the interesting aspects of the change.
- (5%) A list of the resources you used to complete the assignment, including message board posts, online resources (including resources outside the course readings like Stack Overflow or blog posts with design ideas), and students or course staff discussions you had in-person. Please do collaborate **and give credit to your collaborators**.

If you choose to submit this version of PA6, we will grade your original PA5 submission and PA6 independently. Your score on PA5 **and** PA6 will be the **higher** of the two scores.

Implementation Recommendations

As an overarching concern, you should focus on finding *some* cases where you can optimize and ensure that your optimizations don't break anything. Some particularly problematic cases are:

- Checking for overflow when folding arithmetic – make sure that if you perform constant folding on arithmetic it doesn't overflow the representable range, and report errors as appropriate.
- Avoiding propagating a constant past a variable assignment. For example, these two programs are not equivalent, even though a naive constant propagation may generate the second from the first:

```
(let ((x 5))
  (print x)
  (set x 10)
  (print x)))

(let ((x 5))
  (print 5)
  (set x 10)
  (print 5)))
```

  We *don't* recommend trying (at least not at first) to make the second `print` turn into `10`.[1] Instead, it's a good idea to be *safe* and not propagate the `x` variable at all.

  In order to detect this case, one approach is to write a function that detects if a variable appears on the left-hand side of a `set` within another expression:

```
let rec is_variable_set_in_e (e : expr) (x : string) : bool = ...
```

  Alternatively, you could collect *all* the assignable variables in an expression, and use that as a helper:

```
let rec get_assignable_vars (e : expr) : string list = ...
```

  It's up to you to design and find the places to use these functions.

[1] This gets quite complex with loops and conditionals, and gets into the topic of *control-flow analysis*, which is a bit beyond this assignment