Assignment 3: Optimization

Due: 10/16/25 at 11:59pm
Submission link: https://www.gradescope.com/courses/1107485/assignments/6891118

In this assignment, you will implement two different kinds of compiler optimizations using the program analysis techniques we discussed in class.

3a: Dataflow Analysis (50 points)

Motivation

As a motivating example, consider this Rice program:

```
fn main() {
  let x = "Hello" in
  let y = x ^ " world" in
  println(y)
}
```

The Rice compiler will generate unoptimized bytecode for `main` (with some unused locals removed):

```
fn main(x0: ()) -> () {
  local x0: ()
  local x1: ()
  local x2: string
  local x3: string
  local x4: fn(string) -> ()
  bb0:
    x2 = "Hello"
    x3 = x2 ^ " world"
    x4 = closure(println, [])
    x1 = x4(x3)
    return x1
}
```

Constant analysis on `main` should deduce that `x2` is always `"Hello"`, `x3` is always `"Hello world"`, and `x4` is always `closure(println, [])`. After constant analysis, the compiler can propagate those constants in the bytecode:

```
// After constant propagation
fn main(x0: ()) -> () {
  local x0: ()
  local x1: ()
  local x2: string
  local x3: string
  local x4: fn(string) -> ()
  bb0:
    x2 = "Hello"
    x3 = "Hello world"
    x4 = closure(println, [])
    x1 = println("Hello world")
    return x1
}
```

Next, the compiler can eliminate the unused variables `x2`, `x3`, `x4` by performing *dead code elimination* (DCE). DCE analyzes the liveness of variables and uses the following rule: given a statement $x = rv$, if x is not live after the statement, and rv has no possible side effects, then the statement is dead code and can be removed. Applied to our example, we end up with the following program:

```
// After constant propagation and dead code elimination
fn main(x0: ()) -> () {
  local x0: ()
  local x1: ()
  bb0:
    x1 = println("Hello world")
    return x1
}
```

Specification

The Rice starter code contains the skeleton of a dataflow analysis framework in `bc/dataflow/mod.rs`, heavily inspired by the `rustc_mir_dataflow` module in the Rust compiler. A dataflow analysis must implement the `Analysis` trait, specifying an abstract domain `Domain` which must be a join-semilattice, as well as specifying a bottom element and a transfer function.

Worklist algorithm

Your first challenge is to implement `analyze_to_fixpoint<A: Analysis>`, which uses the worklist algorithm to compute the fixpoint solution for an arbitrary dataflow analysis. The algorithm should follow the direction specified by `A::DIRECTION`. Its return type is:

```
IndexVec<Location, <A as Analysis>::Domain>
```

This type conceptually represents a mapping from locations (CFG instructions) to abstract domains. It is implemented using the `IndexVec` type from the `indexical` crate, which is essentially a `Vec` except indexed by `LocationIdx` instead of `usize`. Note that the i-th entry should contain σ_i^{in}, not σ_i^{out}, as that is generally the state you want during optimizations.

The reference implementation of `analyze_to_fixpoint` is 40 significant lines of code (SLOC, code without whitespace or comments).

Optimization passes

Your second challenge is to implement constant propagation and dead code elimination as optimization passes based on constant analysis and liveness analysis. An optimization pass is any function which takes `&mut Function` and returns a `bool` which is true if the pass modifies the program. For example, for DCE you will need to define a function such as:

```
pub fn dead_code(func: &mut Function) -> bool {
  // return true if code was eliminated
}
```

To register this pass with the optimizer, add `Box::new(dead_code)` to the `passes` vector in `optimize_func`. This function executes all dataflow passes in sequence to a fixpoint.

Constant propagation. Your constant analysis must implement support for the following features:

- Anything which is a `Const` or a `Closure` with an empty environment should be trackable as a constant. This does not include GC-allocated types like tuples, structs, arrays, interface objects, or closures with environments.
- Constants should be tracked through assignments and copies. This does not include projections or array indexing. For example, given the program `let x = (1, 1) in x.0`, your analysis does not need to determine that `x.0` is `1`.
- Constants should be folded for at least the following binary operators: addition/subtraction/-multiplication/division over floats and integers, and concatenation over strings.

The reference implementation of constant propagation is 170 SLOC.

Dead code elimination. Your DCE must implement support for the following features:

- Liveness should be tracked at the granularity of locals. Your liveness analysis does not need to be field-sensitive.
- The following operations should be assumed to have a potential side effect: function calls, array indexing (reads or writes), and division. You can assume memory allocation will never fail, and so does not have a side effect.

The reference implementation of DCE is 80 SLOC.

Implementation

Some tips for the implementation:

- When implementing `analyze_to_fixpoint`, note that `JoinSemiLattice::join` is defined such that it returns true if `self` changes. You can use this fact to implement a variant of the worklist algorithm we discussed in class.
- For constant analysis, you will need to define a new data type for its abstract domain, and implement `JoinSemiLattice` for that type. For liveness analysis, you should *not* need to define a new data type for its abstract domain.
- For optimization passes, the `VisitMut` trait is a handy way to walk over a function and edit its instructions in-place.
- For DCE, once you delete instructions from the body, you will need to call `Body::regenerate_domain` to remove unused locations from the body's location domain.

Grading

The autograder will use the `-O1` flag to enable optimizations and the `--dump-ir` flag to emit a JSON representation of the optimized bytecode. Your compiler will be run on a series of programs which should contain no constant assignments or binary operators once optimized. For a given program, if your compiler misses any optimization opportunities, then it fails the test. Additional regression tests have been added to ensure the soundness of your analyses.

3b: Escape Analysis (50 points)

Motivation

As a motivating example, consider this Rice program:

```
fn construct_greetings() -> [string] {
  let parts = ("Hello", " world") in
  let greetings = [parts.0 ^ parts.1, parts.1 ^ parts.0] in
  greetings
}

fn main() {
  let g = construct_greetings() in
  println(g[0]);
  println(g[1])
}
```

The tuple `parts` does not escape the function where it is defined, and the array `greetings` does. An efficient language runtime can therefore *stack-allocate* the tuple `parts` outside the purview of the garbage collector, and free its memory immediately upon returning from `construct_greetings`.

Specification

Pointer analysis

Your first challenge is to implement an intra-procedural flow-insensitive field-sensitive Andersen-style *pointer analysis*. This pointer analysis should be able to determine for any given place p the set of allocations which p may point to. The pointer analysis does not need to distinguish between different indexes of an array.

The reference implementation of pointer analysis is 200 SLOC.

Escape analysis

Your second challenge is to implement an escape analysis on top of the pointer analysis. Your escape analysis should identify all allocations in a function which definitely do not escape by (1) a return, (2) a function call, or (3) an assignment to an argument. An optimization pass built on the escape analysis should then change the `AllocLoc` of all non-escaping allocations to `Stack`.

The reference implementation of escape analysis is 70 SLOC.

Stack runtime

Your third challenge is to implement stack allocations in the Rice runtime. Your implementation should satisfy the following design criteria:

- Tuples and arrays can be stack-allocated (you can ignore closures).
- The representation of a pointer to a stack-allocated value is up to you to decide.
- Stack-allocated data should be stored on the `Frame` so it is deallocated when the frame is deallocated. ("Stack-allocated" is a slight misnomer, as you will have to heap-allocate these objects, just not in the Wasmtime GC.)

- You only need to implement stack allocation in the interpreter, not the Wasm compiler.

Implementation tips

- To track allocations, it may be helpful to define a newtype for locations like `struct Allocation(Location)` along with a newtype index over allocations.
- When considering function/method calls in pointer analysis, the safest default is to assume that all inputs could flow to each other, and that all inputs could flow to the output.
- Don't forget to check every sub-place of a returned place when looking for escaping allocations.
- When considering the representation of pointers to stack-allocated values, it may be helpful that you can check at runtime whether a Wasmtime `Val` represents a primitive value or a GC'd reference.

Grading

As with 3a, the autograder will use the `--dump-ir` flag to analyze your compiler's generated bytecode. Your pointer and escape analysis will be graded on a set of test cases based on whether your compiler converts every possible heap allocation to a stack allocation (not too many, not too few). Your stack runtime is difficult to automatically grade, so I will manually review your code for its adherence to the design criteria specified above.