About

The lua_optimizer is a program that performs optimizations on Lua code. We use a technique called abstract interpretation to accomplish two optimizations: constant propagation and unreachable code elimination. The motivation for this work is that these types of optimizations cannot be done by a single-pass compiler. It is based on the Conditional Constant Propagation algorithm of Wegman and Zadeck [1].

Overview

The first step of our analysis is finding which variables are constant at each point in the program, and finding code that is unreachable. We say a variable is constant at some point of the program when it can have only one possible value at that point. At every statement, we try to determine the value of all variables immediately before and after each statement's execution, and we use that value to assess which branches the program will take. To that end, we use a technique called abstract interpretation.

Let's look at a few examples. We assume all variables start with a default value of `nil`.

```
x = 1
y = x
print(x, y)
```

Before the first assignment, `x` and `y` have the value `nil`. Right after the first assignment we determine `x` is `1`, because we assign the value `1` to `x`, and that `y` remains `nil`. Right after the second assignment, we determine that `y` has value `1`, because we are assigning the constant variable `x` to `y`, and that `x` remains `1`. Therefore, right before the `print` statement, we have both variables with value `1`. If we execute this program, it will print `1` twice.

Let's add conditions to our program. To simplify, from now on when we say before and after some statement we mean immediately before and immediately after executing that statement.

```
x = 1
if x > 0 then
  x = 2
  y = 3
else
  y = 0
end
print(x, y)
```

Knowing that `x` has a known value of `1` after the first line, we can determine which path our program will take. We know the `else` branch is unreachable, and that the program will always take the `then` branch. After the `if-then-else` statement, we can determine that `x` and `y` have the values of `2` and `3`, respectively. If we execute this program, it will print the values `2` and `3`.

Finally, let's look at loops.

```
x = 0
y = 0
repeat
  x = 1
  y = y + 1
  print(x, y)
until condition
```

Here, at the `print` statement inside the `repeat until` body, `x` is constant and `y` is not constant. First, let's assume that we know nothing about `condition`, so there's no way to find out how many times the statements `x = 1` and `y = y + 1` will be executed.

After the first two lines, we determine that `x` and `y` both have value `0`. When we enter the loop's body, things get more complex. The issue is that there are two ways of getting inside the loop. One coming from outside (that is, from the preceding assignment node), and another from the bottom of the loop (the back edge). And in the real execution, the back edge might be traversed multiple times. However we can still do some high level reasoning about it. Let's start with `x`.

On the first iteration the program assigns `1` to `x`. On subsequent iterations, the program assigns the same value again. Therefore the value of `x` after that assignment is always the same, and we know the value of `x` before the `print` statement. Note that we could add another assignment of the same kind, such as `x = 3` somewhere in the loop before the `print`. It still wouldn't change the fact that `x`—at the `print` statement—is known, because at that point there's only one possible value for `x`.

When analysing `y`, on the other hand, we don't have this guarantee. On the first iteration the program assigns `1` to `y`, on the second iteration it assigns `2` to `y`, and so forth. At the `print` statement, there are multiple possible values `y` can have Therefore `y` is not a constant.

There are a few subtleties in this example, however. We need to revise a few concepts to better understand what's going on.

Lattice

A lattice is a partially ordered set in which every two elements have a unique supremum (also called a least upper bound or join) and a unique infimum (also called a greatest lower bound or meet) [2]. We use a simple flat lattice in our analysis. There are three different levels of elements: the highest element is Top, the lowest is Bottom, and the elements in the middle are all values (e.g. all numbers, all strings, booleans, nil). Note that there is an infinite number of elements in the middle level. However, the lattice still has a bounded depth as every element is at most two "steps" away from Bottom.

The abstract interpretation algorithm associates a lattice element to every variable at each point in the program. Every element has a different meaning:

- Top means the variable is constant, yet to be determined.
- At the middle, the elements each represent a different value. That is, a variable with an element of this kind is a constant with a known value.
- Bottom means the variable is not constant (or that it cannot be guaranteed to be constant).

The most relevant operation between elements for our analysis is the meet operation, represented by the symbol $\sqcap$. It is associative and commutative, following three rules:

// Vou colocar depois.

The Optimistic Assumption

Our analysis is done under the optimistic assumption. The abstract interpretation optimistically assumes every variable is constant at every point in the program, and falls back into a more pessimistic truth if the initial assumption was incorrect. Let's go back to our last example.

```
x = 0
y = 0
repeat
  x = 1
  y = y + 1
  print(x, y)
until condition
```

After the second line, we know that `x` and `y` have the lattice element `0`. However, at first, we cannot know that `x` and `y` continue to be constant inside the loop, because the loop assigns to them. In comes the optimistic assumption: we assume `x` and `y` remain unchanged.

Let's start with `x`. Inside the loop, before the first statement, we assume `x` continues to have the lattice element `0`. This may not be true because we might reach that point from the bottom of the loop (the back edge), and `x` might carry a different value from there. After that first statement, `x` has the lattice element `1`. When we reach the bottom, as `condition` is not known, we need to consider the possibility of repeating the loop's body. Back at the loop's top we reach a conflict.

The lattice element of `x` can be `0` (if it came from outside the loop) or `1` (if it came from the back edge). The meet operator between the elements `0` and `1` is Bottom. As such, we come to the conclusion our first assumption was incorrect: `x` is not constant at the loop's top. If we were to add `print` there, the value of `x` would depend on which iteration the loop is at. However, after the first assignment, `x` goes back to having a known value. So, at that point, our assumption was correct: `x` is constant at the print statement.

Now, let's analyse `y`. Inside the loop, before the second statement, we assume `y` continues to have the lattice element `0`. After that statement, `y` has the lattice element `1`, because we assign to `y` the value of `y` (which, under our assumption, is `0`) plus `1`. When we reach the bottom, as `condition` is not known, we need to consider the possibility of repeating the loop's body. Back at the top we reach a conflict.

The lattice element of `y` can be `0` (if it came from outside the loop) or `1` (if it came from the back edge). The meet operator between the lattice elements `0` and `1` is Bottom. As such, we come to the conclusion our first assumption was incorrect: `y` is not constant at the loop's top. Differently from `x`, however, `y` remains not constant after the second statement, because we are assigning to it a value that is, itself, not constant before that statement. So, at that point, our first assumption was also incorrect: `y` is not constant before the print statement.

Let's look into a more interesting example, from Click et al [3]:

```
x = 1
repeat
  x = 2 - x
  print(x)
until condition
```

After the first line, `x` has the lattice element `1`. When we enter the loop's body, we assume `x` remains constant. Inside the loop, before the assignment statement, `x` has the lattice element `1`. After the assignment statement, under our assumption, `x` has the lattice element `1`, because `2 - 1` is `1`. Again, as we don't know the value of `condition`, we have to consider the loop will execute multiple times.

Back at the top, `x` has two possible elements associated with it: the one coming from outside, and another from the back edge. We perform the meet operator between the two. That is, `1` meet `1`. Differently from the previous example, the meet results is `1`, as the meet between two constants, if they are the same, is their common value. Our assumption was correct: `x` is indeed constant at the loop's top. In fact, it is constant at every point of the loop.

Let's look at a more formal description of our algorithm.

Fixed Point and Complex Lattices

In abstract terms, our analysis is looking for the greatest fixed point. We start with the best possible scenario, assuming every variable is an unknown constant (Top) everywhere. Through abstract interpretation, we may find out this assumption was incorrect and need to fall back into a more pessimistic truth, lowering lattice elements until we reach a stable configuration. That stable configuration is a fixed point. We don't start in a fixed point, we reach it by lowering lattice elements.

Note that lowering every variable to Bottom is a stable configuration. However, it is not a very useful fixed point. We want to find the greatest fixed point, the configuration in which we find the highest number of constant variables, because it gives us more information. That information is then used for constant propagation and unreachable code elimination. More on that later.

We could also use a more complex lattice, where types are also added (e.g. the meet of two different numbers is not Bottom, but an element that represents the number type), and "truthy" and "falsy" elements (e.g. the meet between numbers, strings and true is "truthy", the meet between nil and false is "falsy"). Though these changes would increase the depth of our lattice, it would still remain bounded, and might provide more information for our optimizations.

Abstract Interpretation

The abstract interpretation operates on the program's control flow graph. The nodes represent statements, and directed edges represent the control flow. On top of that, each node contains a pair of cells: an "in cell" and an "out cell". A cell is a structure that contains the program's state at that point of execution. It is essentially a map of all variables in scope, associating each variable to a lattice element.

The abstract interpretation works by scheduling edges and executing the node those edges point to. We use a simple iterative worklist technique, containing CFG edges that need be processed. We start with an empty worklist, marking all edges as "not executable", and setting all variables in all cells to Top. This is our optimistic assumption.

The first step of the abstract interpretation is marking the start edge as executable, and adding it to the previously empty worklist. Then we enter a loop: while the worklist is not empty, we remove an edge from the list and execute the node it is pointing to. During the execution of a node, we schedule new edges to be processed, marking them as executable in the process.

Executing a node involves a few steps:

1. Calculate the new "in cell" from all executable incident edges. The algorithm gets all preceding nodes whose edges are traversable, and performs the meet operation between their "out cells". In other words, for all executable in edges, we get the "out cell" of the node they're from, and for each variable of those cells we perform the meet operation between their lattice elements.

2. If the node hasn't been executed before, mark the node as executed and skip to step 3.

   Otherwise, compare the new "in cell" with the previous one. This is the stop condition of our algorithm. If they're equal, we stop this node's execution, because whatever the node is about to execute will yield the same results as the previous iteration. In other words, for the same pair in-state/execution, the out-state will be the same. Therefore there isn't any state changes to propagate, they've already been propagated before.

3. Evaluate the node using the new "in cell" as its state, update the "out cell" with the changes from this computation, and schedule out edges. Scheduled edges are marked as executable. This step depends on the node type:

   - Assignment nodes update a variable's lattice element and schedule its single out edge.
   - Local assignment nodes add a variable to the scope, assign it a value (possibly nil if there're no expressions), and schedule its single out edge.
   - If, while, and repeat until nodes evaluate their condition, and conditionally schedule its out edges. If the condition is a constant, only one branch needs to be scheduled. If it is Bottom, both must be scheduled, as its condition's value is unknown.
   - Generic and numeric for nodes add variables to the inner block's scope, setting their lattice element to Bottom. We then schedules both edges.
   - Function call statement nodes do not change the cell. Their single out edge is scheduled normally.
   - Return statement nodes do not change the cell. They do not schedule edges.

When there are no more edges in the work list, the abstract interpretation is done and a fixed point has been found.

Edges vs Nodes

In our method we schedule edges, rather than nodes. A different approach could, instead, associate the executable flag to nodes. However, this is not optimal, as we can find more constants scheduling edges. The reason is that two nodes may be executable and there may be an edge between them, but that edge may not be traversable. This would result in an unnecessary meet operation that could potentially set bottom to a constant variable. Let's look at a modified example from Wegman and Zadeck [1]:

```
i = 1
while true do
  i = i + 1
  if i == 10 then
    break
  end
end
print(i)
```

Here, the edge exiting the `while` node is not traversable, as its condition is always true. The only way to reach the `print` statement is through the break inside the loop.

If we schedule edges, at the `print` statement the only executable in edge is the one coming from inside the `if` statement. The meet between all its traversable in edges (in this case there's only one) will yield that `i` is a constant with value `10`, because that's the only possible value for `x` inside the `if`.

However, if we associate the executable flag to the nodes, the meet operation between all previous executable nodes would include the `while` body. We would do the meet between Bottom (which is the lattice element associated with `x` at the loop's end) and Constant `10` (which is the lattice element associated with `x` at the end of the `if` node). So we would conclude that `i` is not constant.

Constant Propagation and Unreachable Code Elimination

Now that the fixed point has been found we can perform constant propagation, constant folding and unreachable code elimination. At every node, we have the lattice element associated with each variable at that point. With that information, we can substitute all occurrences of constant variables with their respective values.

The algorithm is essentially a depth first search in the program's CFG. However, only edges marked as executable are traversed. At every node, all expressions are inspected, constant variables are substituted by their value and constants are folded as needed. Code that is unreachable is not processed.

Constant conditions are also simplified. Whenever we find an `if` statement with a constant condition, we eliminate the conditional branching.

```
if constant then
  -- then body
else
  -- else body
end
```

If `constant` evaluates to true, then the entire statement is transformed into a block of code without conditional branching. We wrap the body in a `do` statement to preserve block scoping.

```
do
  -- then body
end
```

On `while` statements, if the condition evaluates to false, we eliminate the loop entirely. Otherwise, if we can guarantee there are no executable `break` and `goto` statements inside the loop, we eliminate everything that comes after it.

On `repeat until` statements, if the condition evaluates to true, we transform it in a single `do` statement. Otherwise, if we can guarantee there are no executable `break` and `goto` statements inside the loop, we eliminate everything that comes after it.

Preparing the Abstract Syntax Tree

Our program starts by parsing Lua code using the LPeg library, generating an abstract syntax tree. But for our analysis, we require a control flow graph of the program, along with information about the scope and upvalues. So we do a preparation step before performing the abstract interpretation, doing the following:

- Build the CFG on top of the AST, without destroying the latter. That is, the statement nodes of the AST become the nodes of the CFG, and we add edges between those nodes without losing the original AST structure.
- Create the "in cells" and "out cells" for every node, initializing all their variables to Top. Note that the cells only contain the variables in scope at that point of the program, so we also have to build the scope on this preparation step.
- Find upvalues, setting their lattice element to bottom at every cell from the point they become an upvalue onwards. We need to do this in case we call some function that changes the value of those variables under our noses. We don't need to set them to Bottom at every cell, it is enough to do so from the point they are captured onwards.

  A more sophisticated analysis could keep their element as Top, and use a more complex lattice to get more refined information about values and types. Then, in the abstract interpretation, only set their values to Bottom when an unknown function is called or when an unknown value escapes.
- Rename each local variable to a unique name, and change all global variables to an `_ENV` table lookup (global `x` becomes `_ENV["x"]`, following Lua's global variable semantics). This step isn't exactly necessary, but for future work with function inlining this will be needed to preserve scope rules in multiple assignments and eliminate possible name collisions.
- Associate all closures with an integer, so we can represent a function lattice element as an index.

This is done with a single iteration over the AST, recursively executing it for closures. Closure parameters are added to the scope with the element Bottom.

Putting It All Together

Our implementation has five main steps.

1. Parse input Lua code using the LPeg library, generating an abstract syntax tree.
2. Prepare the AST, building the control flow graph on top of it, as described previously.
3. Abstract interpretation of the program as described previously. After this is done, all variables have an associated lattice element.
4. Using the found constants, perform constant propagation, constant folding and unreachable code elimination as described previously. This step is a depth first search on the CFG, and modifies the AST itself. Remember that, from the prepare step, AST and CFG nodes are shared.
5. From the modified AST, generate Lua code. Unlike the previous steps, this one iterates over the AST structure, rather than the CFG. Only nodes marked as visited from the previous DFS are considered.

Once this is done, we output the optimized Lua code.

References

[1] Mark N. Wegman and F. Kenneth Zadeck. 1991. Constant propagation with conditional branches. ACM Trans. Program. Lang. Syst. 13, 2 (April 1991), 181–210. DOI:https://doi.org/10.1145/103135.103136

[2] https://en.wikipedia.org/wiki/Lattice_(order)

[3] Click, Cliff & Cooper, Keith. (2000). Combining Analyses, Combining Optimizations. ACM Transactions on Programming Languages and Systems. 17. 10.1145/201059.201061.