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 instruction, we try to determine the value of all variables immediately before and after the 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. Following Lua's semantics, we assume all variables start with a default value of `nil`.

```
local x, y
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 that `x` is `1`, and that `y` remains `nil`. Right after the second assignment, we determine that `y` has value `1`, 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 instruction 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 `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 y == 2*x
```

Here, at the `print` statement inside the `repeat-until` body, `x` is constant and `y` is not constant. Note that the `repeat-until` body will be executed multiple times.

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). In the real execution, the back edge will 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.

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 at that point.

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* (poset) in which every two elements have a *meet* and a *join* [2]. We'll explain what each of those terms mean.

Informally, a *partially ordered set* is a set together with a binary relation ( ) indicating that, for certain pairs of elements of the set, one of the elements precedes the other [3]. The word "partial", opposed to total, means that not all pairs of elements need to be comparable.

To understand the concepts of *meet* (denoted by the symbol , and is also known as an infimum or greatest lower bound) and *join* (denoted by the symbol , and is also known as a supremum or least upper bound) we need to define *lower bound* and *upper bound* of two elements in a poset. Let `A` be a set with a partial order ( ), and `x`, `y` and `z` elements in `A`: * The element `z` is a *lower bound* of the pair `x` and `y` if `z` `x` and `z` `y`.

- Similarly, `z` is an *upper bound* of the pair `x` and `y` if `x` `z` and `y` `z`.

With these two definitions, we can define *meet* and *join* of a poset [4][5]. Let `A` be a set with a partial order ( ), and `x`, `y` and `z` elements in `A`: * The element `z` is the *meet* (or greatest lower bound) of the pair `x` and `y` if it satisfies two conditions: `z` is a *lower bound* of `x` and `y`, and `z` is greater than or equal to any other *lower bound* of `x` and `y`.

- The element `z` is the *join* (or least upper bound) of the pair `x` and `y` if it satisfies two conditions: `z` is a *upper bound* of `x` and `y`, and `z` is less than or equal to any other *upper bound* of `x` and `y`.

Meet and *join* are associative, commutative and idempotent: * x (y z) = (x y) z * x y = y x * x x = x

For our analysis, we'll use a simple bounded lattice, which has a greatest element and a least element. There are three different levels of elements in our lattice: 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 most relevant operation between elements for our analysis is the *meet* operation, with the following rules. Here, Ci and Cj are some middle element: * x Top = x * x Bottom = Bottom * Ci Cj = Ci, if i = j * Ci Cj = Bottom, if i j

The abstract interpretation algorithm associates a lattice element to every variable at each point in the program. This mapping represents some knowledge about the value of each variable, where each element have the following meaning:

- Top means the variable is undefined. The variable has no value at that point.
- 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).

We could also use a more complex lattice. For instance, we could use a 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), or "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, the lattice would still remain bounded, and might provide more information for our optimizations.

Fixed Point

Our goal in this analysis is to assign to every variable at each point in the program a valid lattice element. To explain what "valid" means, let's take a step back and talk about the real execution (also called concrete interpretation) of the following program.

```
x = 1
while x < 5 do
  x = x + 1
end
```

Before the loop, the variable `x` can only assume the value `1`. At the beginning of the loop, `x` starts with value `1` and goes up to `4` in increments of `1`. After the loop, `x` has the value `5`. As such, in the concrete interpretation, the set of values `x` can have before the loop is `{1}`, the set at the beginning of the loop is `{1, 2, 3, 4}`, and the set after the loop is `{5}`

These sets are the concrete values `x` can have during the concrete interpretation, at their respective points of the program. We can map each set of concrete values to a lattice element through what is called an abstraction function. This abstraction function transforms a value from the concrete set to the abstract set, which is out lattice. For our lattice, the set `{1}` abstracts to the lattice element `1`, the set `{1, 2, 3, 4}` abstracts to Bottom, and the set `{5}` abstracts to the lattice element `5`.

A lattice element assigned to `x` at some point of the program is valid if that element is less or equal than the abstraction of the concrete values of `x`. In other words, at each point of the program, our analysis can only assign a lower or equal lattice element to `x` than the abstraction of its concrete values. If the concrete values of `x` abstracts to Bottom, our analysis must assign Bottom to `x`. If the concrete values of `x` abstracts to a constant, our analysis must either assign that constant or Bottom to `x`.

Note that assigning Bottom to every variable is always valid, because Bottom is less than or equal to any lattice elements. However, our analysis wouldn't be very useful if it did that. We want to find the greatest valid lattice element for every variable at every point in the program, because it gives us more information. That information is then used for constant propagation and unreachable code elimination. More on that later.

To that end, our analysis starts by initializing every variable to Top everywhere, and marking every statement as unreachable. This configuration is invalid for most programs. Through abstract interpretation, we may find out those values were incorrect and need to fall back into a more conservative truth, lowering lattice elements and marking statements as reachable 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. If the algorithm is stopped prematurely, the found configuration might be invalid.

Evaluating Expressions

We also need a way to evaluate expressions to abstract values (that is, expressions to lattice elements). To that end, we first recursively evaluate each subexpression, and then evaluate the expression itself. Let's use, as an example, expressions that contain only integer constants, variables, and addition.

Constants evaluate to their respective lattice element. Variables evaluate to the lattice element assigned to them at that point in the program, which may be Bottom. Addition of two constant lattice elements result

in a new lattice element with the result of the addition. Addition of Bottom and any other lattice element results in Bottom.

Note that a variable cannot evaluate to Top, because (in the case of Lua) we cannot use a variable before defining it, and defined variables are initialized with `nil`. When we add the entire language to our expression evaluator, we may discover bugs in the program, such as trying to add `nil` to a string. In those cases, we may stop prematurely and raise an error.

Abstract Interpretation

We can now give an informal description of our algorithm and revisit out last example:

```
x = 0
y = 0
repeat
  x = 1
  y = y + 1
  print(x, y)
until y == 2*x
```

Every statement contains a before state and an after state. The program state is given by a map of every variable to its lattice element, which we initialize with Top.

After the second statement, we know that `x` and `y` have the lattice element `0`. When we get to the loop, we don't know from which edge the program reached that point. The abstract interpretation performs the *meet* between all variables of all incident states, and continues with the updated state.

Let's start with `x`. On the first iteration, we perform the *meet* between the element of `x` coming from the preceding assignment (`0`) and the element of `x` coming from the back edge (Top, as the analysis hasn't reached that point yet). *Meet* between Top and `0` is `0`, so we continue our analysis with `x` being `0`. We may find out that is not true later in the analysis.

After that first statement, `x` has the lattice element `1` because we assign the value `1` to `x`. When we reach the bottom, as the condition evaluates to false, we need to follow the edge that goes back to the top of the loop and repeat the loop's body. Back at the top, we perform the *meet* operator again. Now 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 conclude `x` is not constant at the loop's top. However, after the first assignment, `x` goes back to having a single possible value. So, after the assignment, `x` is constant.

Now, let's analyse `y`. On the first iteration, we perform the *meet* between the element of `y` coming from the preceding assignment (`0`) and the element of `y` coming from the back edge (Top, as the analysis hasn't reached that point yet). *Meet* between Top and `0` is `0`, so we continue our analysis with `y` being `0`. We may find out that is not true later in the analysis.

After second statement, `y` has the lattice element `1`, because the expression `y + 1` evaluates to `1`, as discussed in the previous section. When we reach the bottom, as the condition evaluates to false, we need to follow the edge that goes back to the top of the loop and repeat the loop's body. Back at the top, we perform the *meet* operator again. Now 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 elements `0` and `1` is Bottom. As such, we conclude `y` is not constant. At the second assignment statement, the expression `y + 1` evaluates to Bottom, because (at that point) `y` equals Bottom. So we assign Bottom to `y`, concluding that after that statement `y` is also Bottom, and as such is not constant at that point.

Let's look into a more interesting example, from Click et al [6]. Let's assume `condition` evaluates to Bottom, and as such we don't know how many times (or even if) that loop will repeat.

```
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 perform the *meet* between the two possible elements of `x`: `1` from the previous assignment, and Top from the back edge. *Meet* between `1` and Top is `1`, so we continue with `x` equals `1`, which we may later find to be incorrect. Inside the loop's body, after the assignment statement, `x` has the lattice element `1`, because the expression `x - 1` is `1`. As we don't know the value of `condition`, we have to conservatively assume 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* result is `1`, as the *meet* between two constants, if they are the same, is their common value. As such, we conclude `x` is constant with the lattice element `1` throughout the loop's execution.

The Algorithm

The abstract interpretation operates on the program's control flow graph. The nodes represent instructions, and directed edges represent the control flow.

To each point in the program (that is, before an instruction and after an instruction) we associate a structure that contains the program's state at that point of the execution. We'll call this structure a cell, naming the cell before the instruction the "in cell" and the one after it the "out cell". A cell is a map of all variables in scope to a lattice element. In other words, we map each point in the program to a cell, which maps every variable to a lattice element. As such, to each pair point and variable there's 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 dead, and setting all variables in all cells to Top.

The first step of the abstract interpretation is marking the start edge as alive, 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 alive in the process.

Executing a node involves a few steps:

1. Calculate the new "in cell" from all alive 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 alive 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 alive. This step depends on the node type:
   - Assignment nodes evaluate their expression, update the assigned variable's lattice element with the result of the evaluation and schedule their single out edge.

- Local assignment nodes evaluate their expression, add a variable to the scope, assign it the evaluated expression or `nil` if there are no expressions, and schedule their 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 the condition is Bottom, both edges must be conservatively scheduled, as the 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 dead or alive flag to nodes. However, this is not optimal, as we can find more constants scheduling edges. The reason is that two nodes may be alive 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 alive 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 dead or alive flag to the nodes, the *meet* operation between all previous alive 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 `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] https://en.wikipedia.org/wiki/Partially_ordered_set

[4] https://en.wikipedia.org/wiki/Join_and_meet

[5] https://en.wikipedia.org/wiki/Infimum_and_supremum

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