Capstone: Compiler Optimization

Dec.12, 2022, Yueshan Li (yli287)

Introduction

In this project, I implemented 3 optimizations, including constant propagation, inlining, and common subexpression elimination, in order to speed up the program execution at runtime.

Design/Implementation

1. Constant Propagation

   Constant propagation involves replacing parts of the program that can be statically determined with their statically determined results during compile time to save computation during runtime. In my implementation of propagate constants, I use a propagate helper function to determine which part of the program could be replaced by their static result.

   To start, I replaced primitive operations including `add1`, `sub1`, `plus`, `minus`, `eq`, and `lt` with their statically determined results with pattern matching. The case of replacing let-bound names is more nuanced. Since after determining that a let-bound name could be replaced by number or boolean, in the body of the let expression, any instances of the let-bound names must also be replaced by the same number or boolean. To store the values associated with the static value of the let-bound name, I used a `Symtab` to map the name to its static value and when encountered in the body of the let expression, check the `Symtab` to replace the name with its static value.

   For conditionals, when the expression is an if statement, I first check if the condition would evaluate to a static value, if so and we can determine the static value to be false, then the expression should just return the folded version of the else expression. Otherwise it should return the folded version of the then expression.

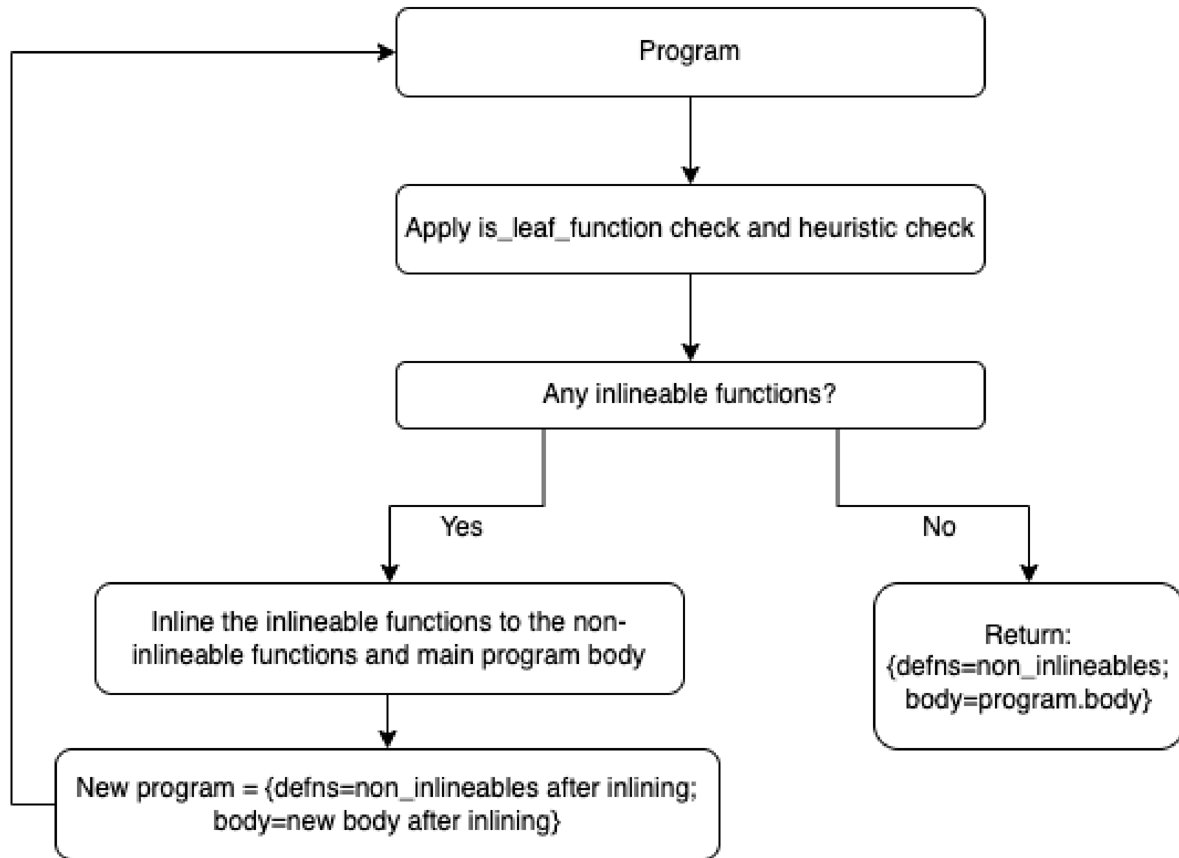
2. Globally unique variables

   Uniquify variables is an AST pass that guarantees that the input program would have globally unique names, as this would be very helpful for other optimizations

such as Inlining and Common Subexpression Elimination. To achieve uniqueness, I use a combination of `gensym` and `Symtab` to store each variable name to its new unique name during Let binding in the body expression of each function definition as well as and function argument names, and the main program body.

3. Inlining

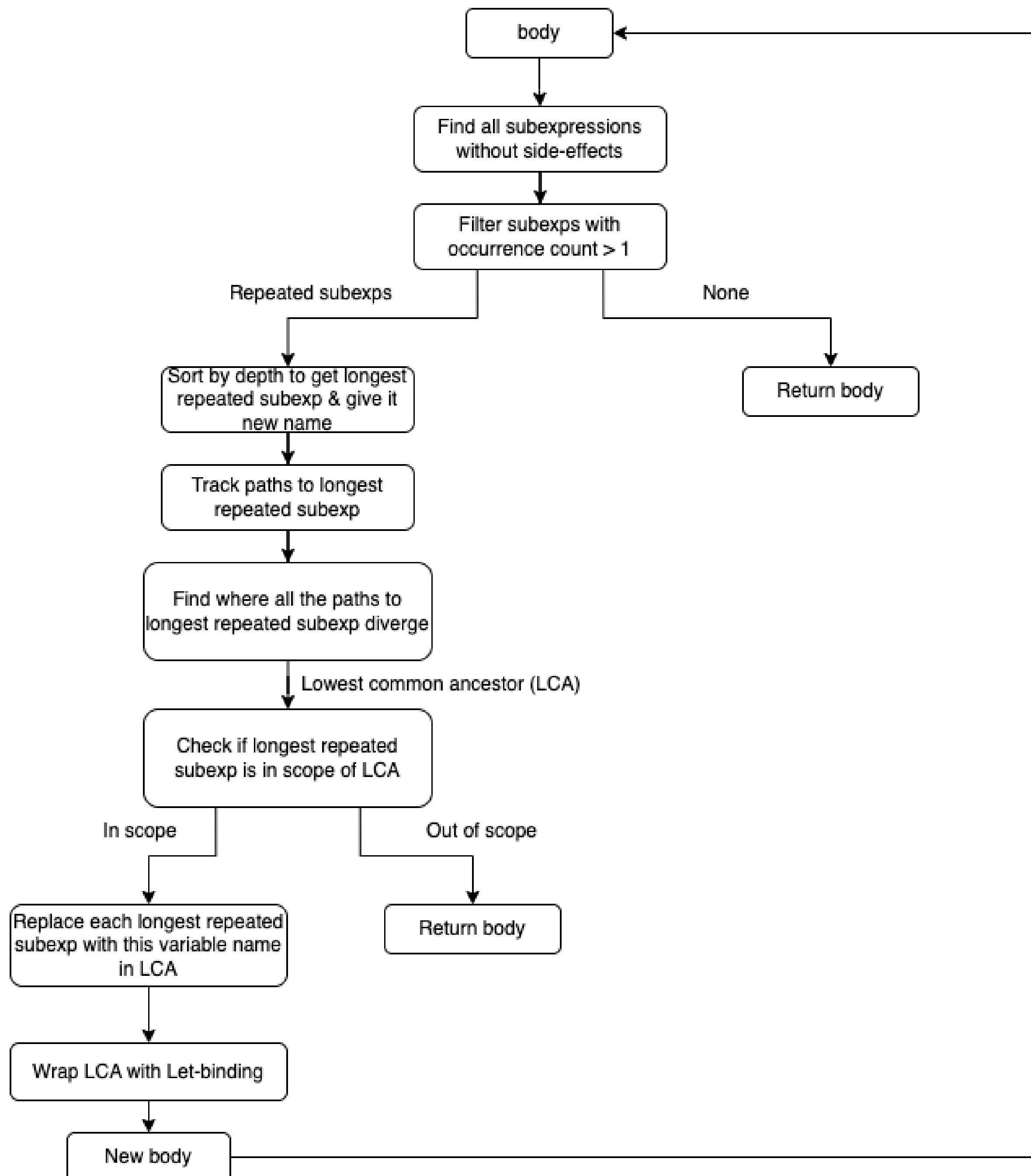
The general method I took for inlining in the project follows the flow-chart below:



In determining whether a function is a leaf function and checking if it satisfies the heuristics, I used `check_static_calls` and `check_depth` on each function. Depth is the number of nodes in the current function AST tree. If a function has 0 static calls in its function body, then it is a leaf function, if the number of statics calls to this function in all functions and the main program body * depth of this function ≤ 100, then I consider the function as passing the heuristic check. This would prevent memory intensive programs after inlining, which could slow down the program execution compared to just doing the function call.

4. Common Subexpression Elimination

The general method I took for common subexpression elimination follows the flow-chart below, the flow-chart represents the process to eliminate common subexpressions in each body expression:



This process would guarantee that we eliminate the longest common subexpression at first, even if it contains more smaller common subexpressions. The check in-scope is also important, especially after function inlining, there can be very long

common subexpressions that are inlined into the body expression from a previous leaf function that could be out of scope if they are extracted out of the previous let binding.

Results

Constant Propagation:

To test the performance of constant propagation, I pick the benchmark tests whose runtime would differ drastically between applying and not applying constant propagation.

To compare, I have the following cases as examples where constant propagation significantly reduce the runtime:

benchmarks/const_prop_if.lisp: (runtime -73.56%)

`const_prop_if.lisp,const prop,5579055.500015784`

`const_prop_if.lisp,no-op,21104358.100024`

benchmarks/const_prop_simple.lisp: (runtime -83.59%)

`const_prop_simple.lisp,const prop,6398201.299964512`

`const_prop_simple.lisp,no-op,38981784.999987215`

benchmarks/const_prop_test.lisp: (runtime -39.03%)

`constant-prop-test.lisp,const prop,5472295.599997778`

`constant-prop-test.lisp,no-op,8975568.39997833`

Inline:

To test the performance of inlining, I pick the benchmark tests whose runtime would differ drastically between applying and not applying uniquify variables + inlining.

To compare, I have the following cases as examples where inlining significantly reduce the runtime:

benchmarks/many_functions_to_inline.lisp: (runtime - 7.83%)

`many_functions_to_inline.lisp,inline,5524806.5999649055`

`many_functions_to_inline.lisp,no-op,5994112.600001244`

benchmarks/inlining-small-function-with-many-call-sites.lisp (runtime - 8.55%):

`inlining-small-function-with-many-call-sites.lisp,inline,6218130.199999905`

`inlining-small-function-with-many-call-sites.lisp,no-op,6799365.099959686`

benchmarks/inlining-allows-cprop.lisp (runtime -5.52%):

`inlining-allows-cprop.lisp,inline,94378585.30000083`

`inlining-allows-cprop.lisp,no-op,99887359.20003363`

benchmarks/inline-with-cprop.lisp (runtime -15.07%):

`inline-with-cprop.lisp,inline,5345569.4999456685`

`inline-with-cprop.lisp,no-op,6293929.300022682`

CSE:

To test the performance of common subexpression elimination, I pick the benchmark tests whose runtime would differ drastically between applying and not applying uniquify variables + CSE.

To compare, I have the following cases as examples where CSE significantly reduce the runtime:

benchmarks/cse_recursive_power.lisp (runtime: -66.46%):

`cse_recursive_power.lisp,cse,8184540.000002016`

`cse_recursive_power.lisp,no-op,24403595.499961738`

benchmarks/inlining-allows-cse.lisp (runtime -7.68%):

`inlining-allows-cse.lisp,cse,93688667.49998689`

`inlining-allows-cse.lisp,no-op,101484254.00000179`

benchmarks/repeatedsum-args.lisp (runtime -16.58%):

`repeatedsum-args.lisp,cse,6275773.000061234`

`repeatedsum-args.lisp,no-op,7522740.999934284`

benchmarks/repeated-subexpressions-multiple (runtime -11.53%):

`repeated-subexpressions-multiple.lisp,cse,6098607.299963987`

`repeated-subexpressions-multiple.lisp,no-op,6893369.500016889`

All Optimizations:

To test the performance of all optimizations working together, I pick the benchmark tests whose runtime would differ drastically between applying and not applying all optimizations in the sequence of constant propagation, uniquify variables, inline, and eliminate common subexpressions.

To compare, I have the following cases as examples where all optimizations significantly reduce the runtime:

benchmarks/const-prop-reassoc.lisp (runtime -30.72%):

```
const-prop-reassoc.lisp,all-op,5091941.599994244
const-prop-reassoc.lisp,no-op,7349752.799996167
```

benchmarks/const-prop-w-inlining.lisp (runtime -31.37%):

```
const-prop-w-inlining.lisp,all-op,5322271.699992598
```

```
const-prop-w-inlining.lisp,no-op,7754620.199989404
```

benchmarks/expanded-fibonacci.lisp (runtime -64.86%):

```
expanded-fibonacci.lisp,all-op,6385940.799987111
expanded-fibonacci.lisp,no-op,18170936.000024088
```

benchmarks/inline_req_constprop.lisp (runtime -97.56%):

```
inline_req_constprop.lisp,all-op,7181759.600007355
inline_req_constprop.lisp,no-op,294613675.200003
```

benchmarks/wilsons-theorem.lisp (runtime -37.92%):

```
wilsons-theorem.lisp,all-op,689518578.4999967
wilsons-theorem.lisp,no-op,1110709975.200007
```

Conclusion

I learned a lot from implementing compiler optimization and realized it is a pretty hard task to achieve optimal performance in all cases, since despite the increased performance in parts of the benchmark test cases, some optimized program actually runs a bit slower than the non-optimized program, potentially to due to their memory intensity after optimization, and finding the best heuristic also needs trail and error.