Optimizing Yul Code

Christian Reitwießner etc.
chris@ethereum.org

1 Optimizing Operations

This section describes various operations that can be employed to Yul code, their benefits and drawbacks. The guiding principle for all these operations is that they perform simple replacements that are as local as possible in order to easy correct implementation. The second goal is that code should always be readable in order to be able to follow the actions of the optimizer. This is also one reason why the Yul optimizer does not make use of full static single assignment (SSA) form and instead tries to keep multi-level expressions and control structures.

TODO: For each step, we should investigate which normal forms are preserved

1.1 Disambiguator

The disambiguator renames identifiers such that each of them has a unique name across the whole code. This is a prerequisite for all other optimiser stages. One of the benefits is that identifier lookup does not need to take scopes into account and we can basically ignore the result of the analysis phase.

All subsequent stages have the property that all names stay unique. This means if a new identifier needs to be introduced, a new unique name is generated.

The resulting code is said to be *disambiguated*.

Prerequisites: None
Benefits: Makes other optimizer stages easier to implement
Drawbacks: Decreases code readability

1.2 Function Hoister

The function hoister moves all function definitions to the end of the topmost block. This is a semantically equivalent transformation on disambiguated code. This is because moving a definition to a higher-level block cannot decrease its visibility (forward-references to functions are valid) and variable references cannon cross function boundaries.

Prerequisites: Disambiguator
Benefits: Structure of the AST will be simpler (no nested functions)
Drawbacks: Modularity is partly lost

1.3 Function Grouper

The function grouper has to be applied after the disambiguator and the function hoister. Its effect is that all topmost elements that are not function definitions are moved into a single block which is the first statement of the root block.

After this step, a program has the following normal form:

```
{ I F... }
```

Where `I` is a block that does not contain any function definitions (not even recursively) and `F` is a list of function definitions such that no function contains a function definition.

We call this normal form *function grouped normal form*. All subsequent optimizer transforms assume the code to be in this normal form.

Prerequisites: Disambiguator, Function Hoister
Benefits: Structure of the AST will be slightly simpler
Drawbacks: None

1.4 Expression Breakup

This step is similar to what is generally called "local value numbering", when combined with the Expression Simplifier and the Common Subexpression Eliminator. The idea is that new variables are introduced such that all arguments to function call expression (including opcodes) are variables. Outside of the statement context (i.e. for example in loop conditions) naively applying this transform might change the order of function calls and thus its application is limited to statements. The resulting code thus will still have complex expressions in some cases.

Another benefit of this operation is that opcodes can be more easily reordered, because the arguments to functions are variables, whose references can span multiple statements.

More specifically, the transform is defined as follows:

The transform traverses the AST and performs changes to variable declarations, assignments and expression statements. These changes will inject new variable declarations right before the current statement and also modify the current statement.

The breakup routine applied to an expression will inject new variable declarations rigth before the current statement and return the name of a new variable. In detail:

Breakup aplied to a variable just returns the variable name.

Breakup applied to a literal `x` creates a new variable `v`, injects `let v := x` before the current statement and returns `v`.

Breakup applied to a function call `e(a1, ..., an)` applies breakup to `an`, ..., `a1` (in that order), resulting in variable names `vn`, ..., `v1`. It creates a new variable `v`, injects `let v := e(vn, ..., v1)` before the current statement and returns `v`.

The breakup routine only affects the following statements:

Variable declaration and assignment: If it is the value is a function call, apply breakup to its argumenst in reverse order.

Expression statement: If the expression statement is a function call, apply breakup to its arguments in reverse order.

Note that since the recursion is applied in reverse order, the order of function calls and opcode excution is unchanged.

Prerequisites: None
Benefits: Simplifies reordering of function calls
Drawbacks: Decreases code readability

1.5 Pseudo-SSA Transform

As explained in the introduction, we do not want to convert the code into a full static single assignment form. In certain cases, it might be beneficial, though, to at least disallow re-assignments to variables and complex expression trees.

We cannot disallow all re-assignments to variables, because of the scoping rules. If we introduce a new variable inside a nested block, it will not be visible in the parent block. For the assignment to `a` in the following example, we cannot introduce a new variable, because it will not be visible anymore for the multiplication operation. In general, due to the fact that Yul retains control structures and does not replace them by plain jumps, it cannot use Phi functions.

```
function f(a, b) -> c {
    if eq(a, 0) {
        a := 3
    }
    c := mul(a, b)
}
```

The transform is defined as follows:

We specify the transform in pseudo-code.

We traverse the AST and keep track of the current name of each local variable in a mapping called `C`.

For simplicity, we only explain single-assignments.

TODO: Reset C for function definitions?

```
transform(N, C) =
  match N
    identifier x -> C[x], C
    declaration let x := e -> let x := transform(C, e), C[x] := x
    assignment x := e ->
      allocate new variable x'
      let x' := transform(C, e), C[x] := x'
    block { S1, ..., Sn } ->
      S1', C1 := transform(S1, C)
      ...
      Sn', Cn := transform(S{n-1}, C{n-1})
      let U be a block containing a sequence of assignments of the form
        xi := Cn[xi]
      for all xi such that C[xi] != Cn[xi] and xi is not
      declared inside the block
      S1', ..., Sn', U, C
    * -> apply transform recursively
```

After this transformation, a variable is guaranteed to still contain the value of its initial declaration at a point P if TODO is there a good condition for this?

Example transform:

```
{
    let a := 1
    {
        a := 2
    }
    let c := mul(a, 3)
}
```

is transformed to

```
{
    let a := 1
    {
        let a1 := 2
        { a := a1 }
    }
    let c := mul(a, 3)
}
```

Note that variables can still be re-assigned, and it is generally advised to keep track of re-assignments, but variables retain their values for a longer stretch of code. Because the old values of variables are still visible, this allows the Common Subexpression Eliminator to be more efficient.

Prerequisites: None
Benefits: Reduces mutability of variables
Drawbacks: Decreases code readability

1.6 Functional Inliner

The functional inliner depends on the disambiguator, the function hoister and function grouper. It performs function inlining such that the result of the inlining is an expression. This can only be done if the body of the function to be inlined has the form `{ r := E }` where `r` is the single return value of the function, `E` is an expression and all arguments in the function call are so-called movable expressions. A movable expression is either a literal, a variable or a function call (or EVM opcode) which does not have side-effects and also does not depend on any side-effects.

As an example, neither `mload` nor `mstore` would be allowed.

This inliner is useful for expressions where Expression Breakup cannot be applied.

1.7 Full Function Inliner

This inliner can inline any function call, but has the drawback that the resulting code will contain more statements. It is best applied after Expression Breakup.

Statements of the form `let x1, ..., xn := f(a1, ..., an)`, where `f` is a function defined as `function f(v1, ..., vn) -> r1, ..., rn B` are replaced as follows:

Let `B'` be a copy of `B` where each variable `v` is replaced by a new unique name `C[v]` and `C[ri] = xi`. Then the statement is replaced by the following statements:

```
let x1, ..., xn
let C[v1] := a1
...
let C[vn] := an
B
```

Note that `a1`, ..., `an` must be variables.

Prerequisites: Expression Breakup

1.8 Rematerialisation

The rematerialisation stage tries to replace variable references by the expression that was last assigned to the variable. This is of course only beneficial if this expression is comparatively cheap to evaluate. Furthermore, it is only semantically equivalent if the value of the expression did not change between the point of assignment and the point of use. The main benefit of this stage is that it can save stack slots if it leads to a variable being eliminated completely (see below), but it can also save a DUP opcode on the EVM if the expression is very cheap.

The algorithm only allows movable expressions (see above for a definition) in this case. Expressions that contain other variables are also disallowed if one of those variables have been assigned to in the meantime. This is also not applied to variables where assignment and use span across loops and conditionals.

TODO: Rewrite this assuming pseudo-SSA and especially Expression Breakup. After Expression Breakup, movability can be relaxed to something like "the expression commutes with all expressions in between".

1.9 Unused Definition Pruner

If a variable or function is not referenced, it is removed from the code. If there are two assignments to a variable where the first one is a movable expression and the variable is not used between the two assignments (and the second is not inside a loop or conditional, the first one is not inside), the first assignment is removed.

1.10 Function Unifier

1.11 Expression Simplifier

This step can only be applied for the EVM-flavoured dialect of Yul. It applies simple rules like `x + 0 == x` to simplify expressions.

1.12 Ineffective Statement Remover

This step removes statements that have no side-effects.

1.13 Common Subexpression Eliminator

1.14 Redundant Writes Eliminator

1.15 Write Delay

This stage swaps two commuting statements with the goal of delaying writes to memory or storage. The goal is to move writes to the same location together, so that one of them can be removed by the Redundant Writes Eliminator.

2 Order of Operations

Now we want to discuss a reasonable order in which the operations mentioned above should be applied. Since some operations are more or less opposite to each other, the order has a big influence on the resulting code.