Dec 16, 2021 TVM 2021 - virtual conf **@qualcomm**

Common Subexpression Elimination for TIR

Franck Slama

Senior Engineer
Qualcomm Technologies, Inc.

Agenda

- 1 - Reminder about TIR (Tensor Intermediate Representation) structure
- 2 - The problem of redundant computations in TIR
- 3 - Removing redundant computations by CSE
- 4 - Relying on the SSA form
- 5 - Algorithm and data-structures
- 6 - Degrees of freedom
- 7 - Semantics preservation with CSE
- 8 - Extension : from syntactical to semantical comparisons
- 9 - Results and conclusion

1 - Reminder about TIR structure

- TIR is one of the two unified IR that TVM has
  - Relay is the highest-level IR. It’s a functional language
  - TIR is lower-level. A TIR program will represent a layer (often called an op)

- Inductive/recursive definitions of expressions and statements in TIR
  - For instance : An IF statement contains a statement for the then branch and another statement for the else branch
    - $\underbrace{\text{If(expr) Then stmt1 Else stmt2;}}_{\text{a Stmt}}$

1 - Reminder about TIR structure

- TIR has both some functional traits and some imperatives traits
  - It has let-in bindings, just like a functional language:
    - $\underbrace{\text{Let var = Expr in Expr}}_{\text{an Expr}}$ <u>Ex</u> : Let x = 1 in x+1
  - But it also has statements like an imperative language:
    - In particular, it has a construction for a Sequence of statements:
      - $\underbrace{\text{S1; S2; …; Sn}}_{\text{a Stmt}}$ <u>Ex</u> : Mem[i1] = 1; Mem[i2] = 2; …
  - So of course let-in bindings can also be Statements:
    - $\underbrace{\text{Let var = Expr in Stmt}}_{\text{a Stmt}}$ <u>Ex</u> : Let x = 5 in Mem[i1] = x

2 - The problem of redundant computations in TIR (1/2)

- TIR code is automatically generated
  - Therefore, it often contains redundant computations (for many passes it is easier to produce TIR code naively with these redundancies rather than eliminating them while performing some other work)
  - The CSE pass in the LLVM backend will take care of some redundancies, but not all of them
  - These re-computations are using computer power for nothing
  - Which in turns makes the output code less energy efficient…

- Example on real TIR code (thanks to contributor @wrongtest on Github):

```
A[i*256 + j*16 + 0] = B[i*256 + j*16 + 4096]
A[i*256 + j*16 + 1] = B[i*256 + j*16 + 4097]
...
```

  - Duplicate index computations produced by loop unroll
  - Needs to rely on target backends abilities which may or may not optimize them out!

2 - The problem of redundant computations in TIR (2/2)

- At Qualcomm Technologies, Inc., we want to make the compiled code run as fast as possible, while using as little energy as possible
  - Using many architecture-dependent optimizations, that use Qualcomm® Hexagon™ DSP capabilities (a family of DSP found in many devices, from smartphones to cars).
  - But also using architecture independent optimizations, like removing redundant computations!

- So we wanted to have a TIR pass for removing these redundant computations : a Common Subexpression Elimination (CSE) optimization pass

3 - Removing redundant computations by CSE

- What we want this pass to do:
  - To avoid re-computations of expressions and sub-expressions
  - By introducing the redundant computations into new variables before the first occurrence
  - And replacing all the occurrences of the sub-expression by the new variables

- For instance

The previous TIR chunk :

```
A[i*256 + j*16 + 0] = B[i*256 + j*16 + 4096]
A[i*256 + j*16 + 1] = B[i*256 + j*16 + 4097]
...
```

Should be transformed into : CSE pass

```
Let cse_var_1 = i*256 + j*16 in
   A[cse_var_1 + 0] = B[cse_var_1 + 4096]
   A[cse_var_1 + 1] = B[cse_var_1 + 4097]
   ...
```

- Note : Not only working on full expr, but sub-expressions too (**i*256 + j*16** here)!

4 – Relying on the SSA form

- The general idea is, within a function, to look for identical subexpressions, and to replace them by new variables, that will be introduced before the first occurrence.

- Question : What if variables change in the meantime?
  - … **i*256 + j*16** …;
  - i = 0;
  - … **i*256 + j*16** …;

- **It won't happen!** TIR code must always be in SSA (Static Single Assignment) form
  - The SSA form implemented in TVM is in fact a little bit more flexible than pure SSA, but the idea is the same : **a variable won't see its content change**
  - That is great for the CSE pass, as is removes the need to pay attention to variables changing content (otherwise, we could not common out **i*256 + j*16** if \`i\` could be updated between the different occurrences)

5 - Algorithm and data-structures (1/15)

The general idea (1/2)

- Take the root of the AST as input. Look at the eligible computations that it computes that can be introduced into a variable. Common them by introducing them into new variable and perform the replacements. (… to be continued…)

- Definition 1: A computation is eligible if and only if:
  - It is not an atom (there's no point in commoning them into new variables)
    - It is not constant
    - It is not a variable
  - It is not a forbidden computation (commoning them would change the program's semantics)
    - It is not a function call
    - It is not a memory load
  - It does not contain a forbidden computation (see above)

5 - Algorithm and data-structures (2/15)

The general idea (2/2)

- Take the root of the AST as input. Look at the eligible computations that it computes that can be introduced into a variable. Common them by introducing them into new variable and perform the replacements. (… to be continued…)
- Definition 2: An eligible computation can be introduced if and only if:
  - All the variables it uses are in scope at this point
  - It appears "often enough" (currently if nb_times_seen > 2 but this test could become more finer grained by taking into account its length)
- Then, continue to apply the same treatment recursively on the child nodes, because some computations that could not be introduce at the current level might become doable in the children node, when the variables that they use become all in scope.

5 - Algorithm and data-structures (3/15)

Data-structures used (1/2) : Context

- In order to know if a variable is within scope or not, we will use a **context**
- This context will associate variables in scope to an optional expression
  - `Context = std::vector<std::pair<Var, MaybeValue>>;`
- Why optional? Because some variables don't have a fixed value
  - Parameters of functions don't have a fixed value
  - Counters of « For » Loops will see their value change
- When we encounter nodes that declare new variables, we will extend our context
  - « Let » nodes (both the expression and the statement) explicitly declare new variables
  - « For » nodes declare a counter that we need to add to the context as we want to be able to do commoning on expressions that use the counter
- The context will serve two purposes:
  - Knowing which variable are in scope at the moment
  - Knowing if a given expression already exists in an already defined variable

5 - Algorithm and data-structures (4/15)

Data-structures used (2/2) : Table of computations

- *Reminder* : Take the root of the AST as input. Look at the eligible computations that it computes that can be introduced into a variable. (…)
  → Need a data-structure for computing and storing the "computations done by a node"

- This data-structure, that we call a **table of computations**, is a hashtable which maps PrimExpr to integers
  - `TableOfComputations = std::unordered_map<PrimExpr, size_t, ObjectPtrHash, ObjectPtrEqual>;`

- The integer is the number of time the given computation has been seen

- When we encounter a PrimExpr, it is efficient to update (or add) the entry in the table

5 - Algorithm and data-structures (5/15)

The algorithm (1/11)

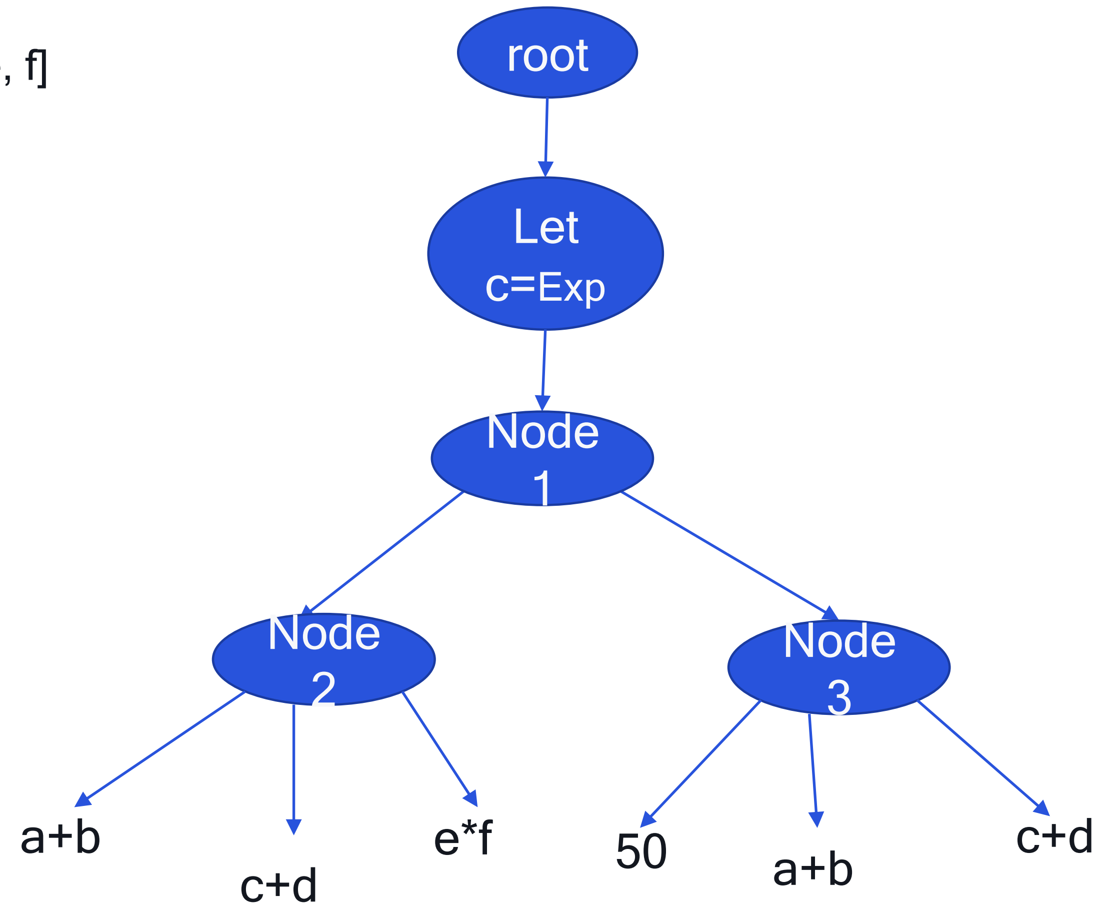
- *Reminder* : Take the root of the AST as input. Look at the eligible computations that it computes that can be introduced into a variable. (…)
- What to do with an eligible computation that can **not** be introduced into a variable? (either because it uses variables not yet within scope, or because it is not seen enough)
- Answer : consider its (eligible) direct subexpressions
- For instance:

  If (w+x)*(y+z) is eligible but can't be introduced (for instance because 'z' is not within scope yet), consider (w+x) and (y+z)

5 - Algorithm and data-structures (6/15)

The algorithm (2/11)

- Input:
  - With an initial context : [a, b, d, e, f]

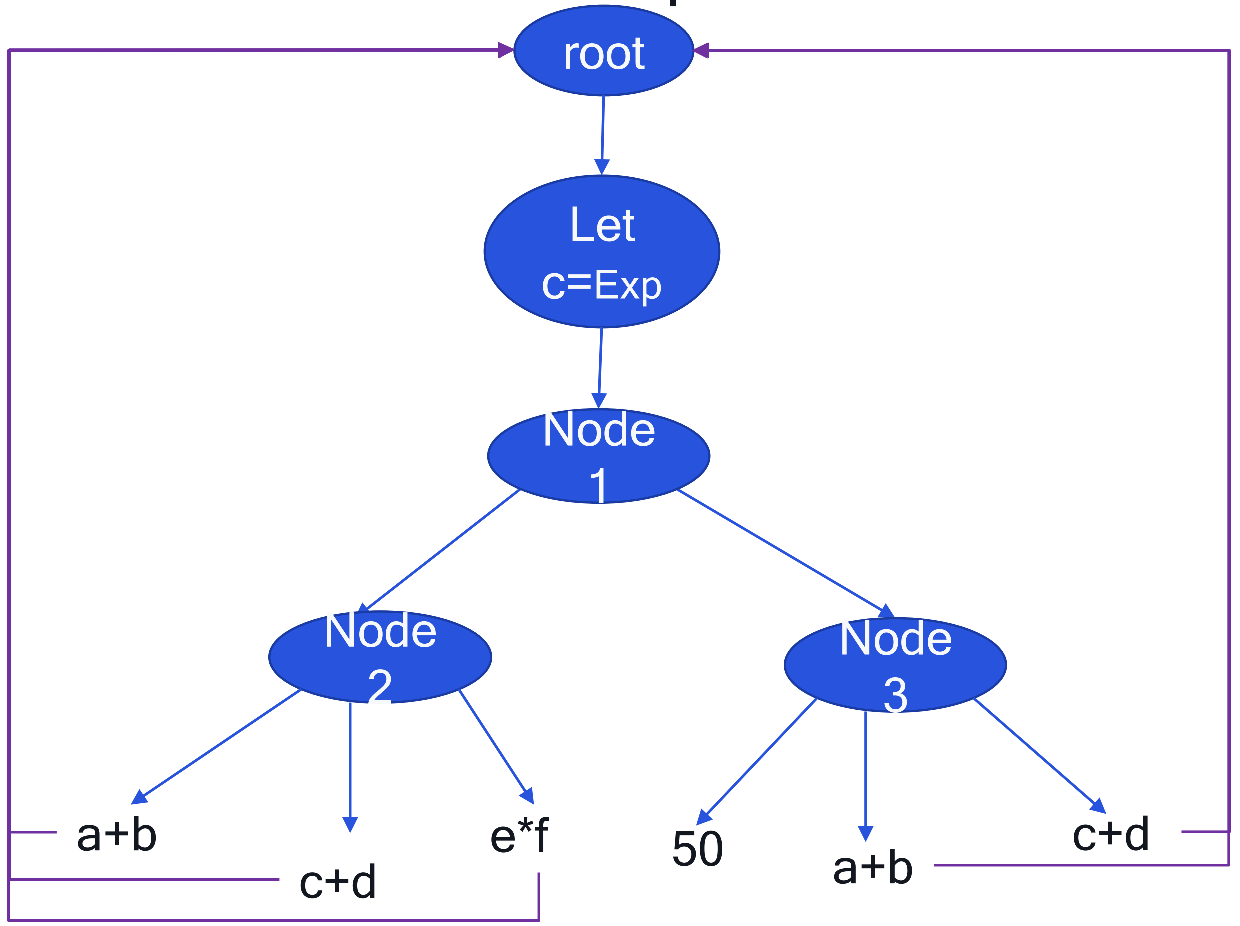


5 - Algorithm and data-structures (7/15)

The algorithm (3/11)

- 1 : Look at the eligible computations done by the current root, i.e. done by all the children of the root. This builds the table of computations.
  - Context : [a, b, d, e, f]
  - **Table of computation** :
    [a+b : 2, c+d : 2, e*f : 1]

- Note:
  - 50 is not eligible (a constant)



5 - Algorithm and data-structures (8/15)

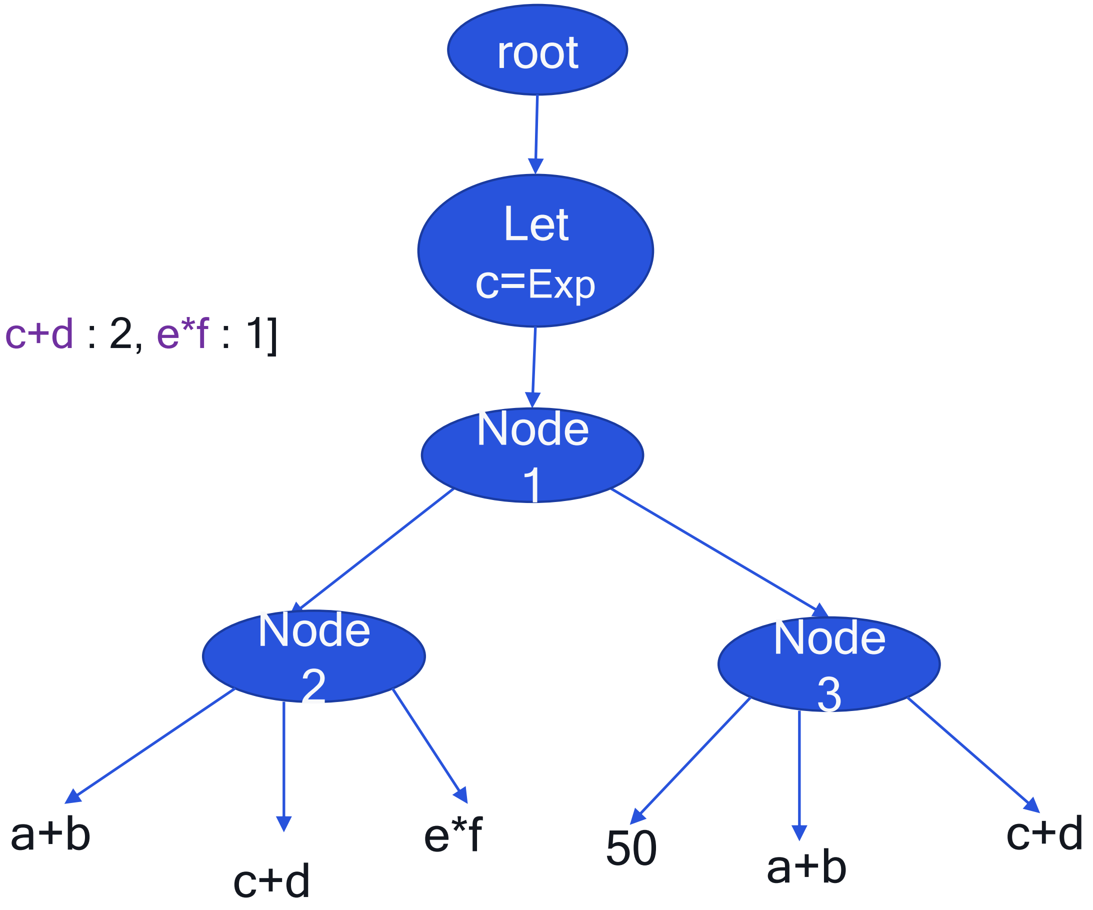
The algorithm (4/11)

- 2 - Considering each computation from the biggest to the smallest in size (not in number of time seen!), look if it can be introduced

- Currently looking at **a+b**
  - Context : [a, b, d, e, f]
  - Table of computation : [a+b : 2, c+d : 2, e*f : 1]

- Answer:

  Yes (a+b) can be introduced
  (it only contains variables within scope, and it is seen often enough)

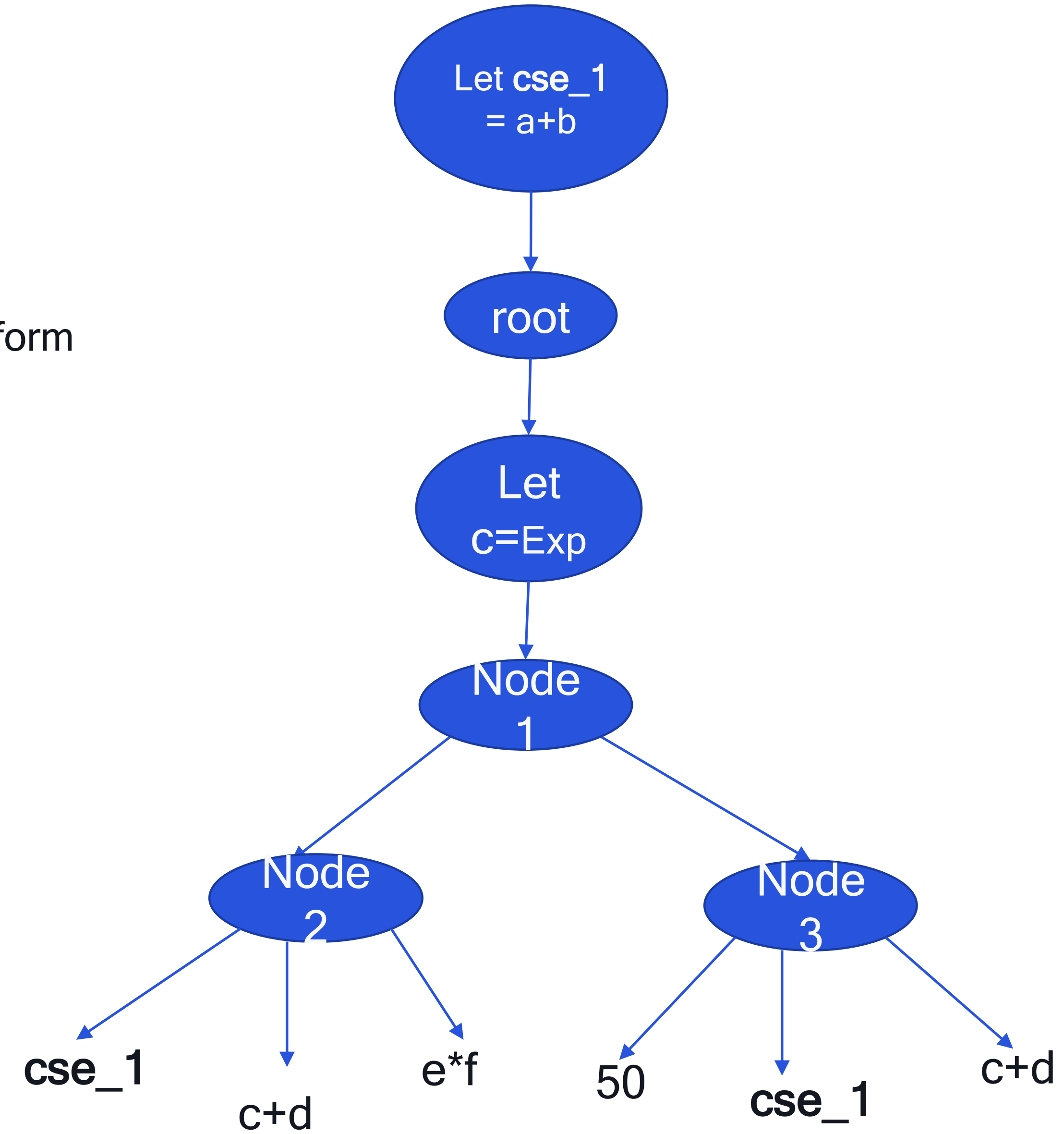


5 - Algorithm and data-structures (9/15)

The algorithm (5/11)

- Actions:

  Create new variable for **(a+b)** and perform replacements



5 - Algorithm and data-structures (10/15)

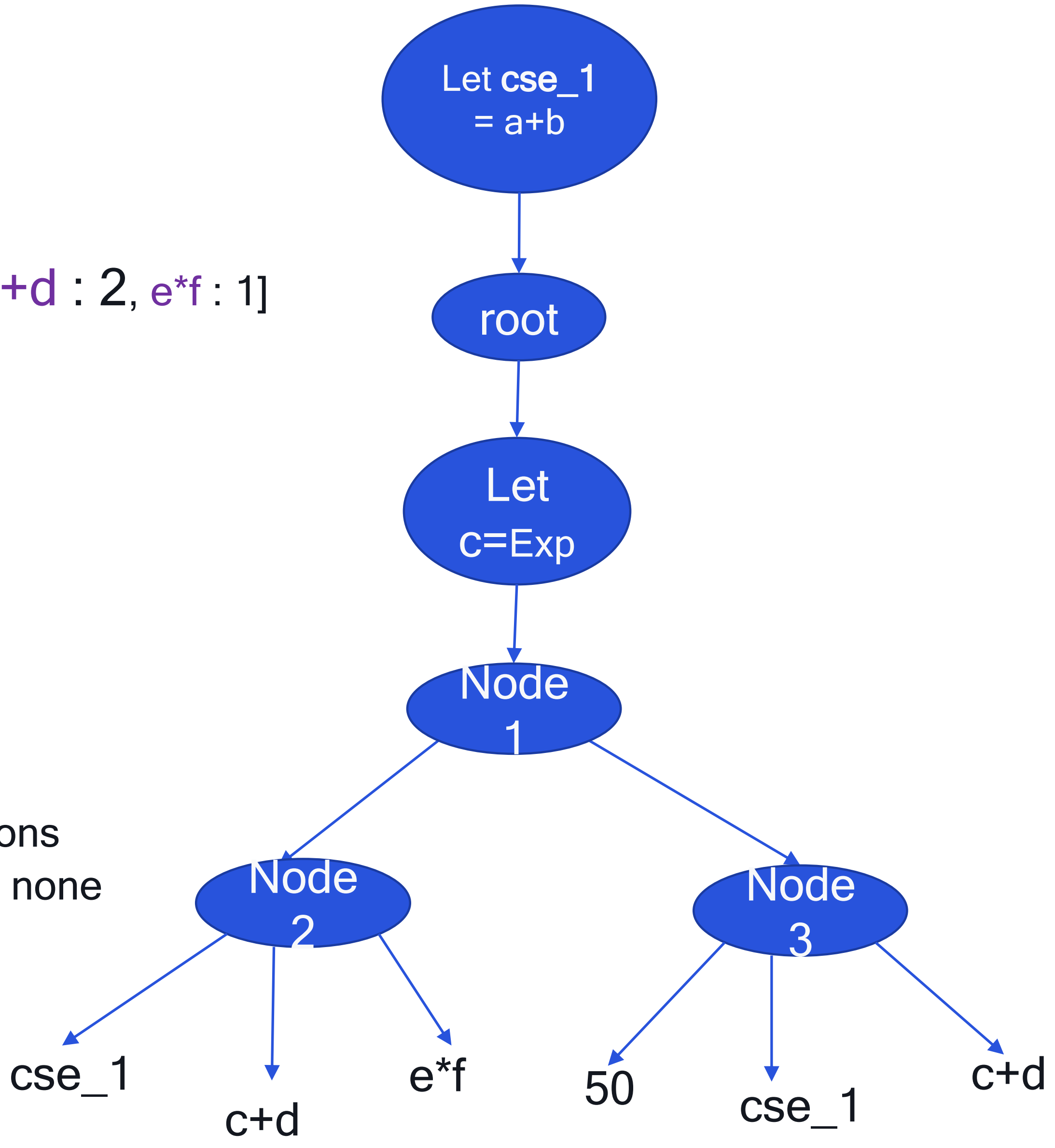
The algorithm (6/11)

- Currently looking at **c+d**
  - Context : [a, b, d, e, f]
  - Table of computation : [a+b : 2, c+d : 2, e*f : 1]

- Answer:

  No (c+d) can not be introduced as 'c' is not in the context (i.e, not within scope yet)

- Actions:
  - Adding its eligible direct-subexpressions to the table of computations. There are none as 'c' and 'd' are not eligible (variables)



5 - Algorithm and data-structures (11/15)

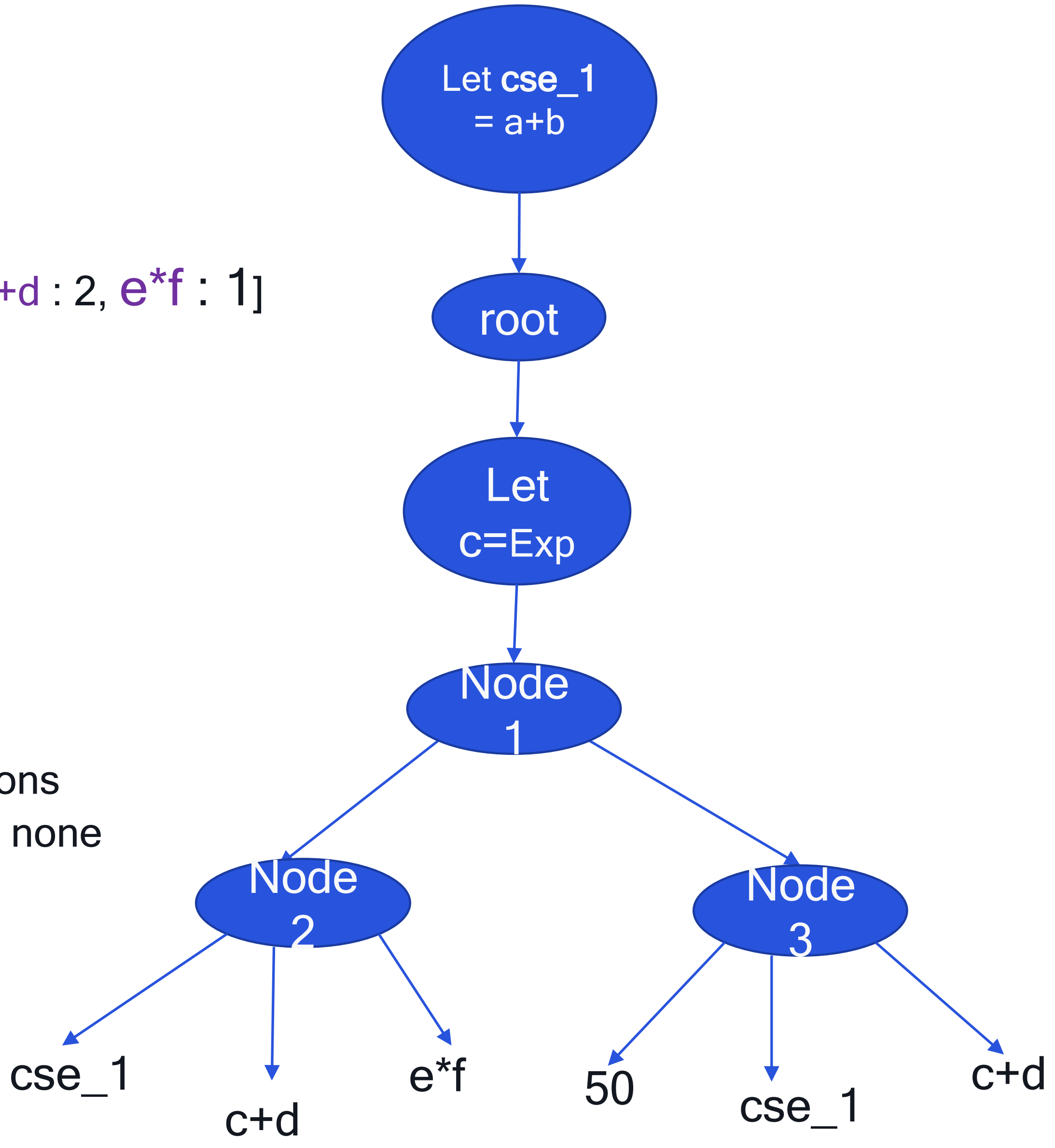
The algorithm (7/11)

- Currently looking at **e*f**
  - Context : [a, b, d, e, f]
  - Table of computation : [a+b : 2, c+d : 2, e*f : 1]

- Answer:

  No (e*f) can not be introduced because it is not seen enough

- Actions:
  - Adding its eligible direct-subexpressions to the table of computations. There are none as ‘e’ and ‘f’ are not eligible (variables)



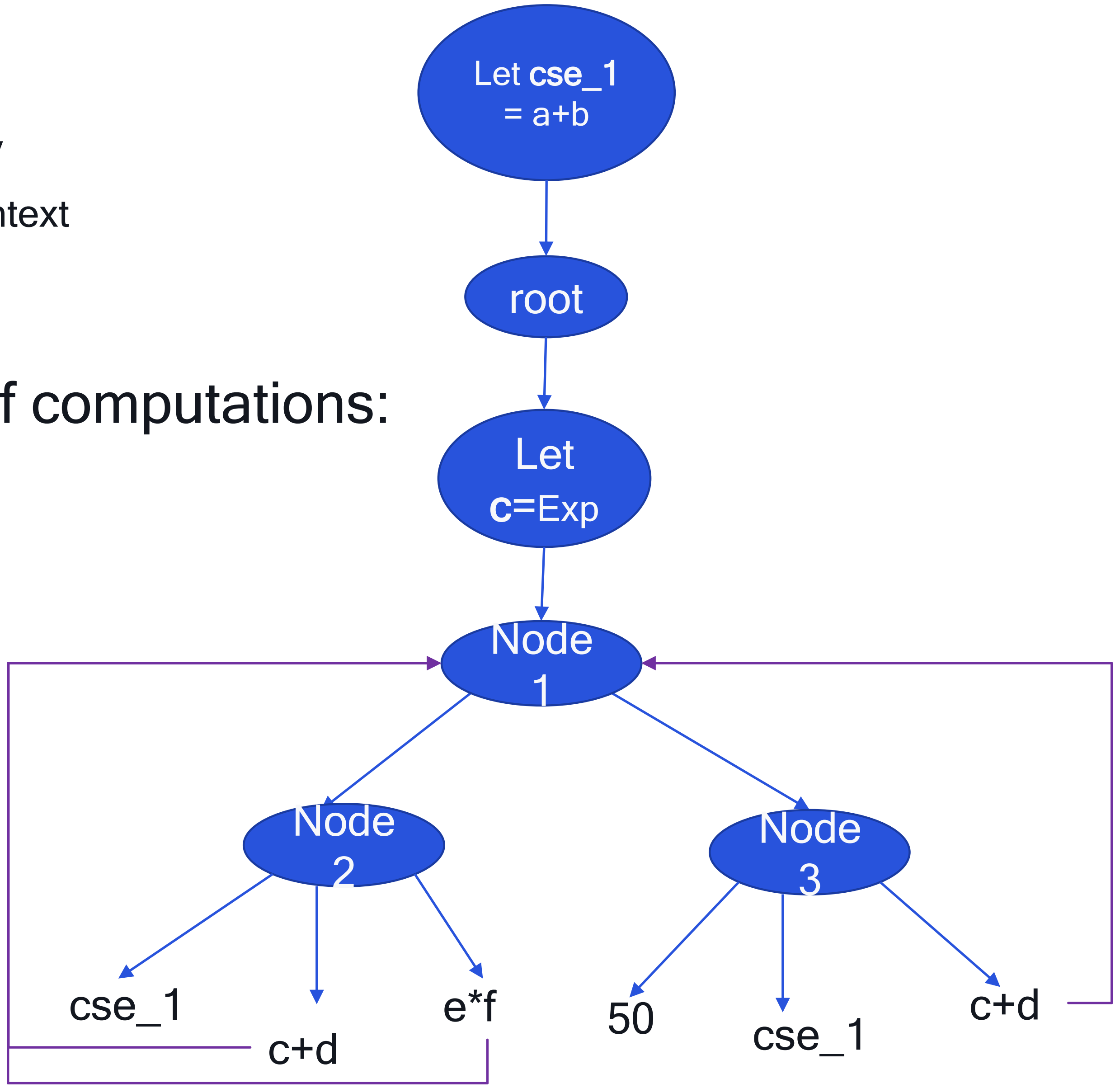
5 - Algorithm and data-structures (12/15)

The algorithm (8/11)

- 3 - Continue recursively
  - This is a Let node, extend the context
    - **Context** : [c, a, b, d, e, f]

- 1 - Compute the table of computations:

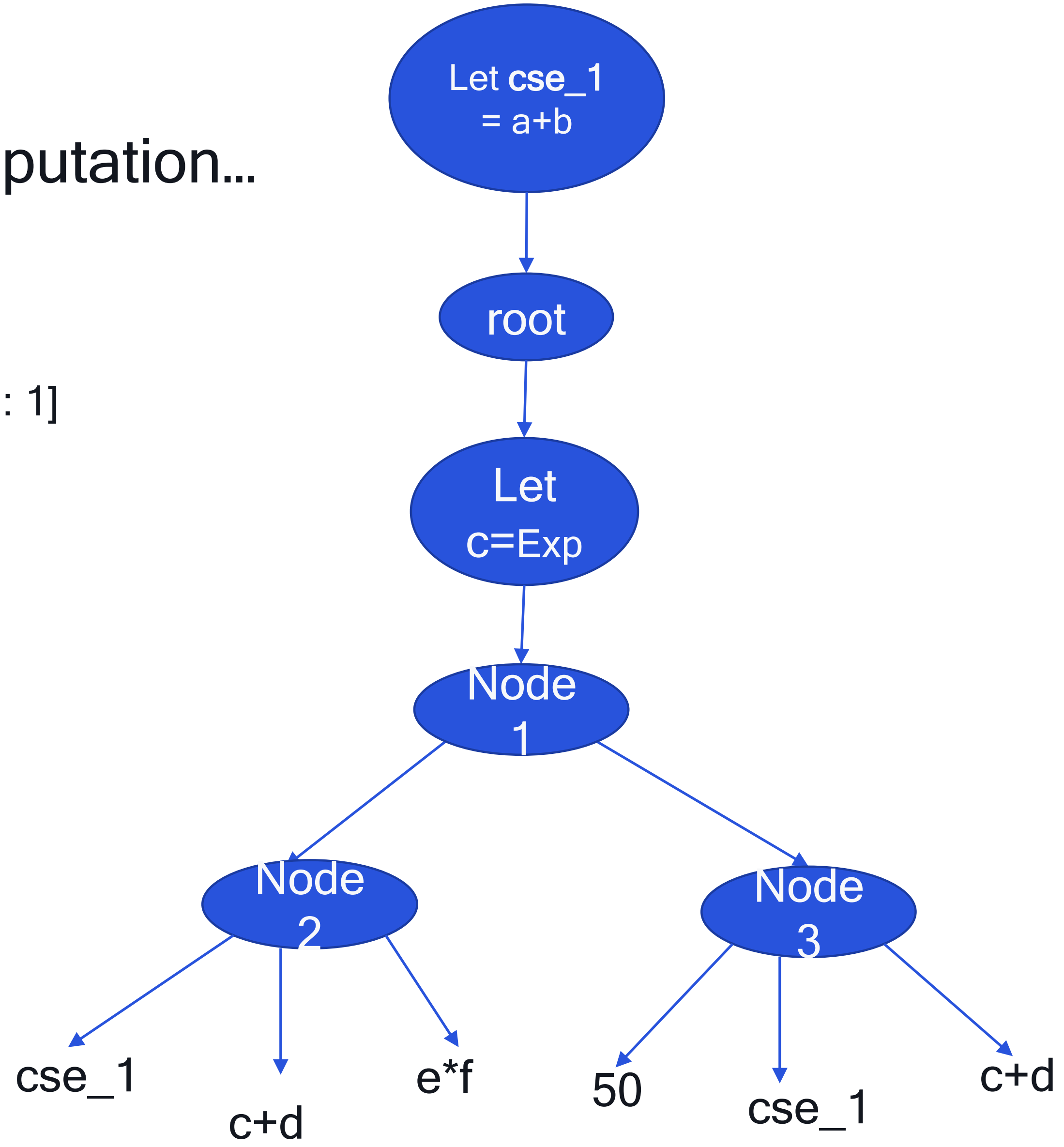
[c+d : 2, e*f : 1]



5 - Algorithm and data-structures (13/15)

The algorithm (9/11)

- 2 - Considering each computation…
- Currently looking at **c+d**
  - Context : [c, a, b, d, e, f]
  - Table of computation : [c+d : 2, e*f : 1]
- Answer:
  Yes (c+d) can be introduced
  (it only contains variables within scope, and it is seen enough)

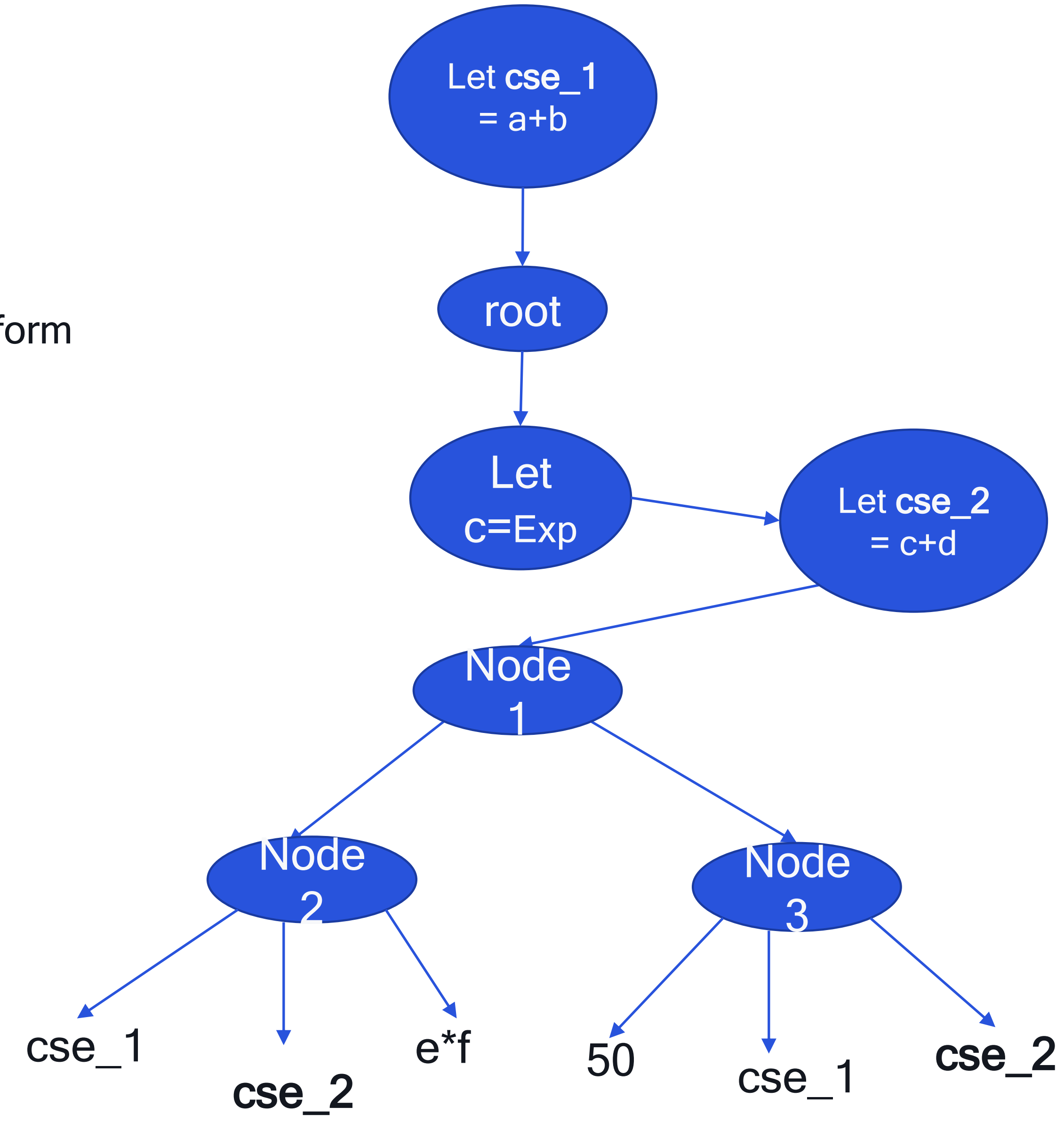


5 - Algorithm and data-structures (14/15)

The algorithm (10/11)

- Actions :

  Create new variable for **(c+d)** and perform replacements



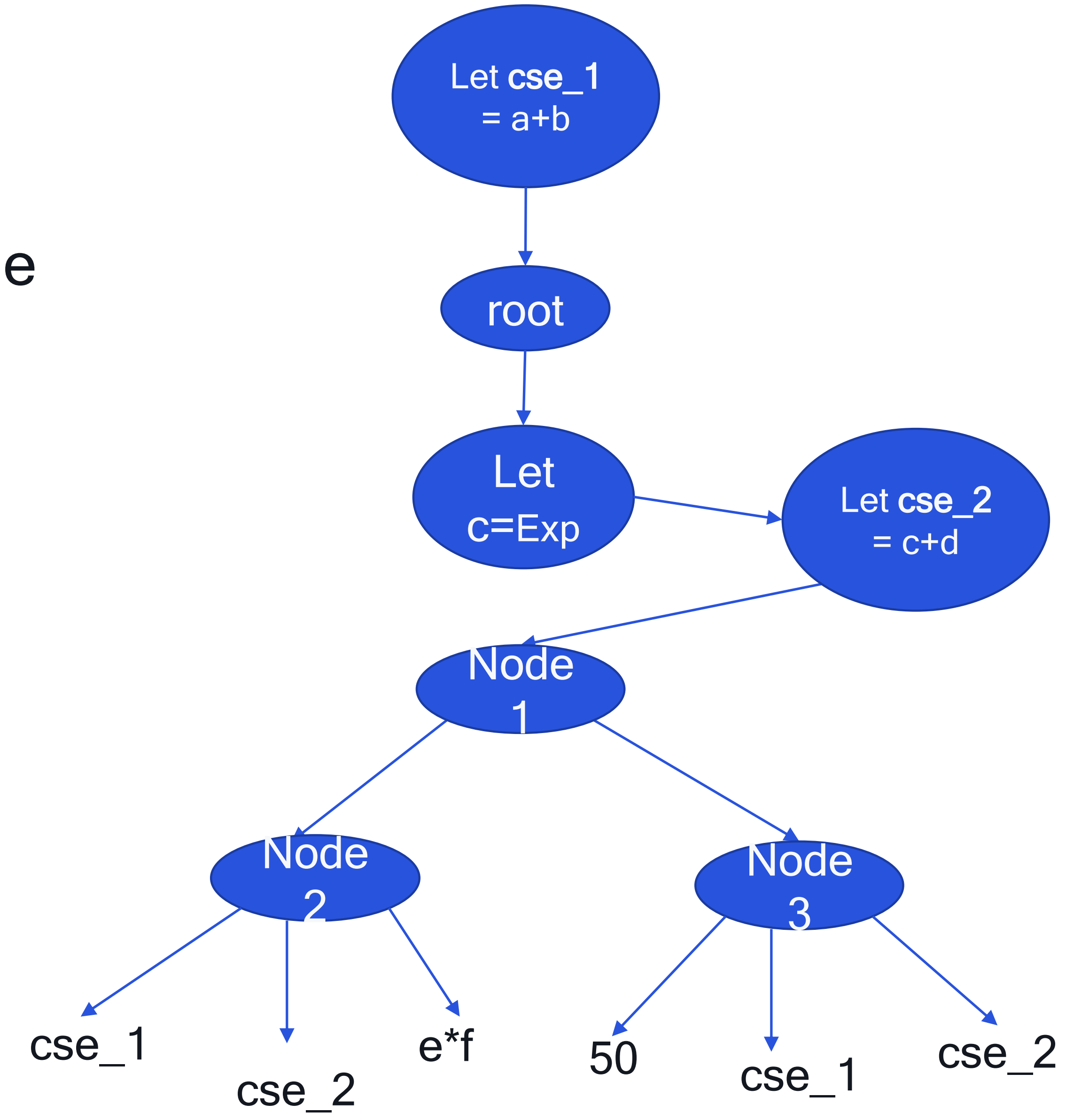
5 - Algorithm and data-structures (15/15)

The algorithm (11/11)

- It won’t do anything for the computation e*f

- …

- Output:



6 - Degrees of freedom

- Often, multiple different choices could have be made as to what to introduce in a new variable.
- Starting by introducing the bigger redundant terms first is better:

```
Let bigComp = … in
    Stmt
```

- The smaller common subterms will be introduced later-on in another Let (i.e. further outside). That will maximize the amount of terms that are put in common

```
Let smallComp = … in
    Let bigComp = … in
        Stmt
```

- That's the reason why we considered the eligible computations from bigger to smaller in the main loop of the algorithm

7 - Semantics preservation with CSE (1/3)

When performing CSE on an IR with imperative traits, one has to be careful with memory loads and function calls

- It’s possible to load two times from the same buffer at the same location, and to obtain different values. That could happen if something has been written between the two loads
  - X = Mem[i1]+42; … … …; Mem[i1] = newValue; … … …; Y = Mem[i1]+42

    → We can’t perform commoning on the redundant expression Mem[i1]+42 !

7 - Semantics preservation with CSE (2/3)

When performing CSE on an IR with imperative traits, one has to be careful with memory loads and function calls

- Two identical function calls f(x) are not guaranteed to evaluate to same value : f might perform side-effect, leading to a function that might compute different result for the same input

  →The CSE pass should not do commoning on the program:

  ```
  f(10);
  …
  …
  f(10)
  ```

- This condition could be relaxed a little bit in the future by relying on a tag “isPure”. Calls to a pure function could be commoned out

7 - Semantics preservation with CSE (3/3)

- As for any compiler pass, preserving the semantics of the program is crucial!
- **Semantics preservation for CSE:**

Consider a TIR program P that has a redundant computation comp:
P = Prog(… comp … comp … comp …)
which previously evaluated to the value val: $P \rightarrow val$

If comp is pure (i.e, does not perform any side effect), **then**

After the CSE pass, which leads to
P’ = Let cse_var = comp in Prog(… var … var … var)

it will still evaluate to the same value val: $P' \rightarrow$ **val**

8 – Extension : from syntactical to semantical comparisons

- Instead of eliminating only duplicated expressions that are **syntactically** the same, it's possible to easily extend the infrastructure of the CSE pass for dealing with any **semantical** comparison (i.e, any equivalence relation on terms)

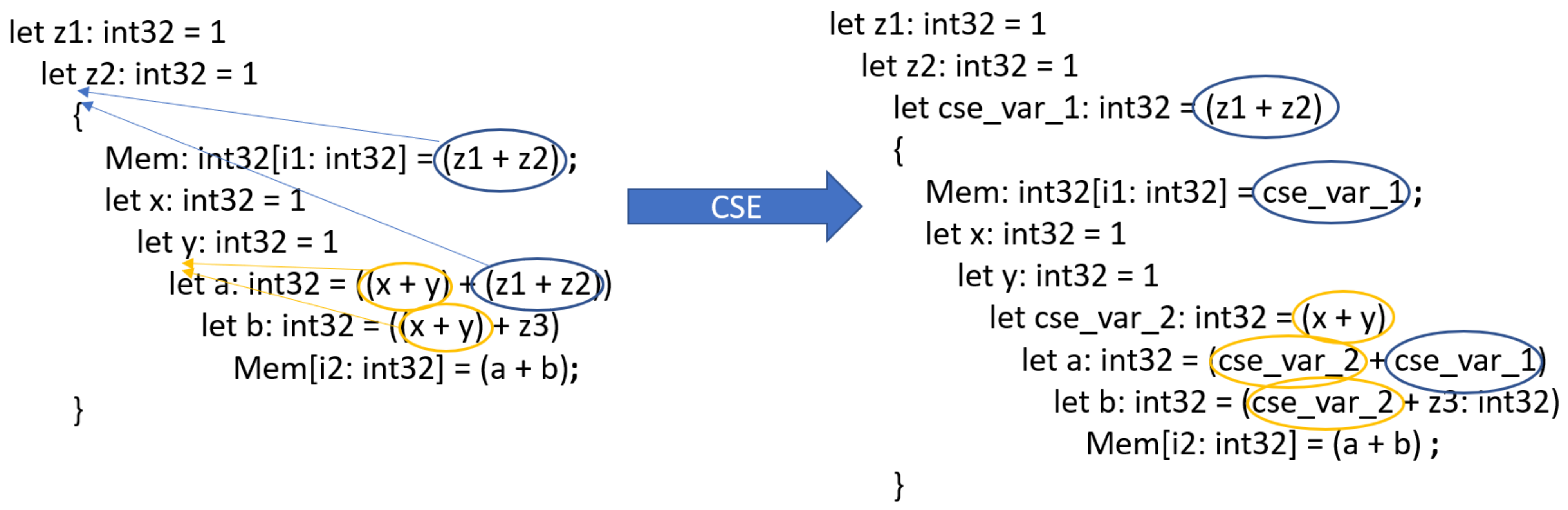
- One could for instance consider to identify expressions modulo commutativity (identifying for instance (x+y) with (y+x)), modulo associativity (identifying for instance (x+y)+z with x+(y+z)), etc
  - That would allow to common-out even more

- Replacing just a single function will be the only thing needed in order to do that:
  - bool EquivalentTerms(const PrimExpr& a, const PrimExpr& b)
  - Typical way to implement such extensions would be to compute a canonical representant of 'a' and a canonical representant of 'b' and to then compare them with the strict syntactical equality

9 - Results and conclusion (1/3)

Results

On a handwritten program:

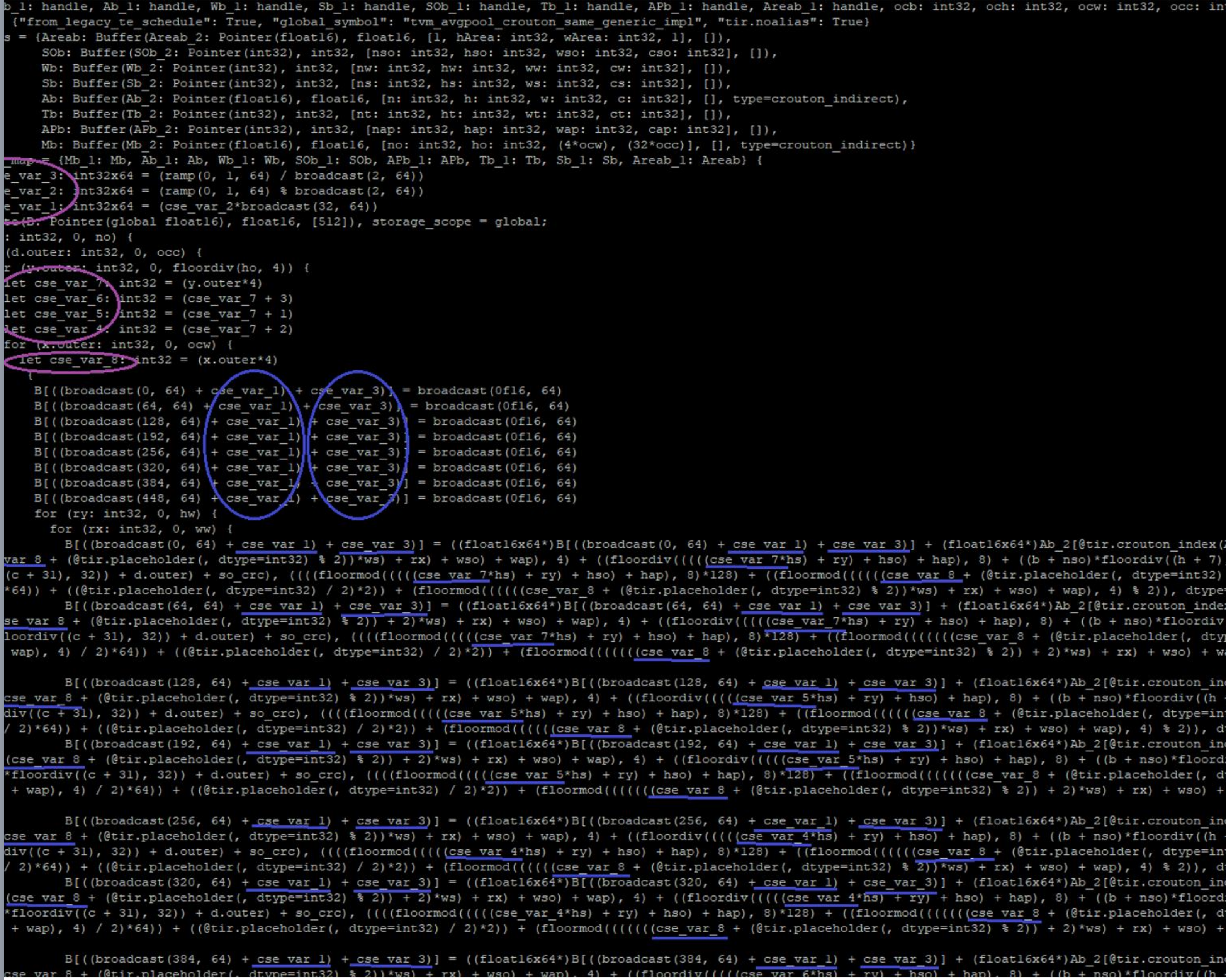


```
let z1: int32 = 1
   let z2: int32 = 1
      {
         Mem: int32[i1: int32] = (z1 + z2);
         let x: int32 = 1
            let y: int32 = 1
               let a: int32 = ((x + y) + (z1 + z2))
                  let b: int32 = ((x + y) + z3)
                     Mem[i2: int32] = (a + b);
      }
```

```
let z1: int32 = 1
   let z2: int32 = 1
      let cse_var_1: int32 = (z1 + z2)
      {
         Mem: int32[i1: int32] = cse_var_1;
         let x: int32 = 1
            let y: int32 = 1
               let cse_var_2: int32 = (x + y)
                  let a: int32 = (cse_var_2 + cse_var_1)
                     let b: int32 = (cse_var_2 + z3: int32)
                        Mem[i2: int32] = (a + b) ;
      }
```

- This handwritten program is a unit test of the pass

Example of output program after CSE on a real layer



```
b_1: handle, Ab_1: handle, Wb_1: handle, Sb_1: handle, SOb_1: handle, Tb_1: handle, APb_1: handle, Areab_1: handle, ocb: int32, och: int32, ocw: int32, occ: in
 {"from_legacy_te_schedule": True, "global_symbol": "tvm_avgpool_crouton_same_generic_impl", "tir.noalias": True}
s = {Areab: Buffer(Areab_2: Pointer(float16), float16, [1, hArea: int32, wArea: int32, 1], []),
     SOb: Buffer(SOb_2: Pointer(int32), int32, [nso: int32, hso: int32, wso: int32, cso: int32], []),
     Wb: Buffer(Wb_2: Pointer(int32), int32, [nw: int32, hw: int32, ww: int32, cw: int32], []),
     Sb: Buffer(Sb_2: Pointer(int32), int32, [ns: int32, hs: int32, ws: int32, cs: int32], []),
     Ab: Buffer(Ab_2: Pointer(float16), float16, [n: int32, h: int32, w: int32, c: int32], [], type=crouton_indirect),
     Tb: Buffer(Tb_2: Pointer(int32), int32, [nt: int32, ht: int32, wt: int32, ct: int32], []),
     APb: Buffer(APb_2: Pointer(int32), int32, [nap: int32, hap: int32, wap: int32, cap: int32], []),
     Mb: Buffer(Mb_2: Pointer(float16), float16, [no: int32, ho: int32, (4*ocw), (32*occ)], [], type=crouton_indirect)}
_map = {Mb_1: Mb, Ab_1: Ab, Wb_1: Wb, SOb_1: SOb, APb_1: APb, Tb_1: Tb, Sb_1: Sb, Areab_1: Areab} {
e_var_3: int32x64 = (ramp(0, 1, 64) / broadcast(2, 64))
e_var_2: int32x64 = (ramp(0, 1, 64) % broadcast(2, 64))
e_var_1: int32x64 = (cse_var_2*broadcast(32, 64))
te(B: Pointer(global float16), float16, [512]), storage_scope = global;
: int32, 0, no) {
(d.outer: int32, 0, occ) {
r (y.outer: int32, 0, floordiv(ho, 4)) {
let cse_var_7: int32 = (y.outer*4)
let cse_var_6: int32 = (cse_var_7 + 3)
let cse_var_5: int32 = (cse_var_7 + 1)
let cse_var_4: int32 = (cse_var_7 + 2)
for (x.outer: int32, 0, ocw) {
  let cse_var_8: int32 = (x.outer*4)
   {
    B[((broadcast(0, 64) + cse_var_1) + cse_var_3)] = broadcast(0f16, 64)
    B[((broadcast(64, 64) + cse_var_1) + cse_var_3)] = broadcast(0f16, 64)
    B[((broadcast(128, 64) + cse_var_1) + cse_var_3)] = broadcast(0f16, 64)
    B[((broadcast(192, 64) + cse_var_1) + cse_var_3)] = broadcast(0f16, 64)
    B[((broadcast(256, 64) + cse_var_1) + cse_var_3)] = broadcast(0f16, 64)
    B[((broadcast(320, 64) + cse_var_1) + cse_var_3)] = broadcast(0f16, 64)
    B[((broadcast(384, 64) + cse_var_1) + cse_var_3)] = broadcast(0f16, 64)
    B[((broadcast(448, 64) + cse_var_1) + cse_var_3)] = broadcast(0f16, 64)
    for (ry: int32, 0, hw) {
      for (rx: int32, 0, ww) {
        B[((broadcast(0, 64) + cse_var_1) + cse_var_3)] = ((float16x64*)B[((broadcast(0, 64) + cse_var_1) + cse_var_3)] + (float16x64*)Ab_2[@tir.crouton_index(
var_8 + (@tir.placeholder(, dtype=int32) % 2))*ws) + rx) + wso) + wap), 4) + ((floordiv((((cse_var_7*hs) + ry) + hso) + hap), 8) + ((b + nso)*floordiv((h + 7)
(c + 31), 32)) + d.outer) + so_crc), ((((floormod((((cse_var_7*hs) + ry) + hso) + hap), 8)*128) + ((floormod(((((cse_var_8 + (@tir.placeholder(, dtype=int32)
*64)) + ((@tir.placeholder(, dtype=int32) / 2)*2)) + (floormod(((((cse_var_8 + (@tir.placeholder(, dtype=int32) % 2))*ws) + rx) + wso) + wap), 4) % 2)), dtype
        B[((broadcast(64, 64) + cse_var_1) + cse_var_3)] = ((float16x64*)B[((broadcast(64, 64) + cse_var_1) + cse_var_3)] + (float16x64*)Ab_2[@tir.crouton_inde
se_var_8 + (@tir.placeholder(, dtype=int32) % 2)) + 2)*ws) + rx) + wso) + wap), 4) + ((floordiv((((cse_var_7*hs) + ry) + hso) + hap), 8) + ((b + nso)*floordiv
loordiv((c + 31), 32)) + d.outer) + so_crc), ((((floormod((((cse_var_7*hs) + ry) + hso) + hap), 8)*128) + ((floormod((((((cse_var_8 + (@tir.placeholder(, dty
 wap), 4) / 2)*64)) + ((@tir.placeholder(, dtype=int32) / 2)*2)) + (floormod((((((cse_var_8 + (@tir.placeholder(, dtype=int32) % 2)) + 2)*ws) + rx) + wso) + w

        B[((broadcast(128, 64) + cse_var_1) + cse_var_3)] = ((float16x64*)B[((broadcast(128, 64) + cse_var_1) + cse_var_3)] + (float16x64*)Ab_2[@tir.crouton_in
cse_var_8 + (@tir.placeholder(, dtype=int32) % 2))*ws) + rx) + wso) + wap), 4) + ((floordiv((((cse_var_5*hs) + ry) + hso) + hap), 8) + ((b + nso)*floordiv((h
div((c + 31), 32)) + d.outer) + so_crc), ((((floormod((((cse_var_5*hs) + ry) + hso) + hap), 8)*128) + ((floormod(((((cse_var_8 + (@tir.placeholder(, dtype=in
/ 2)*64)) + ((@tir.placeholder(, dtype=int32) / 2)*2)) + (floormod(((((cse_var_8 + (@tir.placeholder(, dtype=int32) % 2))*ws) + rx) + wso) + wap), 4) % 2)), d
        B[((broadcast(192, 64) + cse_var_1) + cse_var_3)] = ((float16x64*)B[((broadcast(192, 64) + cse_var_1) + cse_var_3)] + (float16x64*)Ab_2[@tir.crouton_in
(cse_var_8 + (@tir.placeholder(, dtype=int32) % 2)) + 2)*ws) + rx) + wso) + wap), 4) + ((floordiv(((((cse_var_5*hs) + ry) + hso) + hap), 8) + ((b + nso)*floord
*floordiv((c + 31), 32)) + d.outer) + so_crc), ((((floormod((((cse_var_5*hs) + ry) + hso) + hap), 8)*128) + ((floormod((((((cse_var_8 + (@tir.placeholder(, d
 + wap), 4) / 2)*64)) + ((@tir.placeholder(, dtype=int32) / 2)*2)) + (floormod((((((cse_var_8 + (@tir.placeholder(, dtype=int32) % 2)) + 2)*ws) + rx) + wso) +

        B[((broadcast(256, 64) + cse_var_1) + cse_var_3)] = ((float16x64*)B[((broadcast(256, 64) + cse_var_1) + cse_var_3)] + (float16x64*)Ab_2[@tir.crouton_in
cse_var_8 + (@tir.placeholder(, dtype=int32) % 2))*ws) + rx) + wso) + wap), 4) + ((floordiv((((cse_var_4*hs) + ry) + hso) + hap), 8) + ((b + nso)*floordiv((h
div((c + 31), 32)) + d.outer) + so_crc), ((((floormod((((cse_var_4*hs) + ry) + hso) + hap), 8)*128) + ((floormod(((((cse_var_8 + (@tir.placeholder(, dtype=in
/ 2)*64)) + ((@tir.placeholder(, dtype=int32) / 2)*2)) + (floormod(((((cse_var_8 + (@tir.placeholder(, dtype=int32) % 2))*ws) + rx) + wso) + wap), 4) % 2)), d
        B[((broadcast(320, 64) + cse_var_1) + cse_var_3)] = ((float16x64*)B[((broadcast(320, 64) + cse_var_1) + cse_var_3)] + (float16x64*)Ab_2[@tir.crouton_in
(cse_var_8 + (@tir.placeholder(, dtype=int32) % 2)) + 2)*ws) + rx) + wso) + wap), 4) + ((floordiv(((((cse_var_4*hs) + ry) + hso) + hap), 8) + ((b + nso)*floord
*floordiv((c + 31), 32)) + d.outer) + so_crc), ((((floormod((((cse_var_4*hs) + ry) + hso) + hap), 8)*128) + ((floormod((((((cse_var_8 + (@tir.placeholder(, d
 + wap), 4) / 2)*64)) + ((@tir.placeholder(, dtype=int32) / 2)*2)) + (floormod((((((cse_var_8 + (@tir.placeholder(, dtype=int32) % 2)) + 2)*ws) + rx) + wso) +

        B[((broadcast(384, 64) + cse_var_1) + cse_var_3)] = ((float16x64*)B[((broadcast(384, 64) + cse_var_1) + cse_var_3)] + (float16x64*)Ab_2[@tir.crouton_in
cse_var_8 + (@tir.placeholder(, dtype=int32) % 2))*ws) + rx) + wso) + wap), 4) + ((floordiv((((cse_var_6*hs) + ry) + hso) + hap), 8) + ((b + nso)*floordiv((h
```

- Declarations of the new variables introduced by the CSE pass are in purple
- Occurrences replaced are in blue

9 – Results and conclusion (3/3)

Conclusion

- We have implemented a CSE pass for TIR which simplifies many programs and which makes them run faster, independently of any hardware consideration

- The CSE pass implemented is very generic and customizable (one can easily change the predicate that decides if a computation should be introduced into a new variable)

- It can easily be extended to deal with any equivalence relation between terms instead of just the syntactical equality
  - it can probably even serve as a common infrastructure for performing value numbering optimizations

Acknowledgments

- Many thanks to the whole **Qualcomm® TVM** team and in particular to Krzysztof Parzyszek and Jyotsna Verma for their feedback and for encouraging me to upstream this new TIR pass.

- Thanks to @wrongtest on the Github repository for the example he wanted to see simplified by the CSE pass.

- Thanks to the entire **TVM community** in general for your work on such a great product!

Qualcomm

Thank you

Follow us on:

For more information, visit us at:

www.qualcomm.com & www.qualcomm.com/blog