Transformation validation for SSA programs

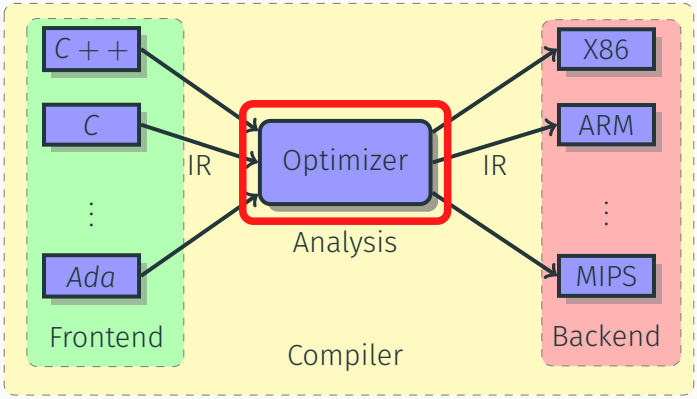
An approach using value graphs à la Paul Govereau

Thomas Rubiano

SAV2019

Motivations

Architecture



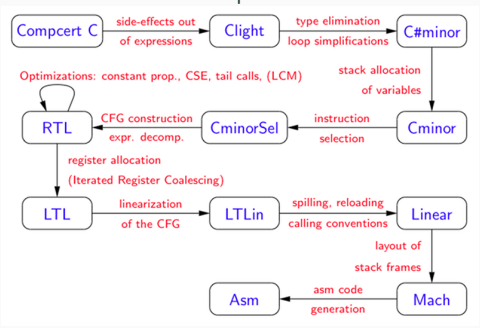
Optimizing compilers

$ clang -O3

```
Pass Arguments: -tti -targetlibinfo -tbaa -scoped-noalias -assumption-cache-tracker -profile-summary-info -forceattrs
      -inferattrs -callsite-splitting -ipsccp -called-value-propagation -globalopt -domtree -mem2reg -deadargelim
      -domtree -basicaa -aa -loops -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -instcombine -simplifycfg
      -basiccg -globals-aa -prune-eh -inline -functionattrs -argpromotion -domtree -sroa -basicaa -aa -memoryssa
      -early-cse-memssa -domtree -basicaa -aa -lazy-value-info -jump-threading -correlated-propagation -simplifycfg
      -domtree -basicaa -aa -loops -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -instcombine
      -libcalls-shrinkwrap -loops -branch-prob -block-freq -lazy-branch-prob -lazy-block-freq -opt-remark-emitter
      -pgo-memop-opt -domtree -basicaa -aa -loops -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -tailcallelim
      -simplifycfg -reassociate -domtree -loops -loop-simplify -lcssa-verification -lcssa -basicaa -aa
      -scalar-evolution -loop-rotate -licm -loop-unswitch -simplifycfg -domtree -basicaa -aa -loops -lazy-branch-prob
      -lazy-block-freq -opt-remark-emitter -instcombine -loop-simplify -lcssa-verification -lcssa -scalar-evolution
      -indvars -loop-idiom -loop-deletion -loop-unroll -mldst-motion -aa -memdep -lazy-branch-prob -lazy-block-freq
      -opt-remark-emitter -gvn -basicaa -aa -memdep -memcpyopt -sccp -domtree -demanded-bits -bdce -basicaa -aa -loops
      -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -instcombine -lazy-value-info -jump-threading
      -correlated-propagation -domtree -basicaa -aa -memdep -dse -loops -loop-simplify -lcssa-verification -lcssa -aa
      -scalar-evolution -licm -postdomtree -adce -simplifycfg -domtree -basicaa -aa -loops -lazy-branch-prob
      -lazy-block-freq -opt-remark-emitter -instcombine -barrier -elim-avail-extern -basiccg -rpo-functionattrs
      -globalopt -globaldce -basiccg -globals-aa -float2int -domtree -loops -loop-simplify -lcssa-verification -lcssa
      -basicaa -aa -scalar-evolution -loop-rotate -loop-accesses -lazy-branch-prob -lazy-block-freq -opt-remark-emitter
      -loop-distribute -branch-prob -block-freq -scalar-evolution -basicaa -aa -loop-accesses -demanded-bits
      -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -loop-vectorize -loop-simplify -scalar-evolution -aa
      -loop-accesses -loop-load-elim -basicaa -aa -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -instcombine
      -simplifycfg -domtree -loops -scalar-evolution -basicaa -aa -demanded-bits -lazy-branch-prob -lazy-block-freq
      -opt-remark-emitter -slp-vectorizer -opt-remark-emitter -instcombine -loop-simplify -lcssa-verification -lcssa
      -scalar-evolution -loop-unroll -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -instcombine -loop-simplify
      -lcssa-verification -lcssa -scalar-evolution -licm -alignment-from-assumptions -strip-dead-prototypes -globaldce
      -constmerge -domtree -loops -branch-prob -block-freq -loop-simplify -lcssa-verification -lcssa -basicaa -aa
      -scalar-evolution -branch-prob -block-freq -loop-sink -lazy-branch-prob -lazy-block-freq -opt-remark-emitter
      -instsimplify -div-rem-pairs -simplifycfg
```

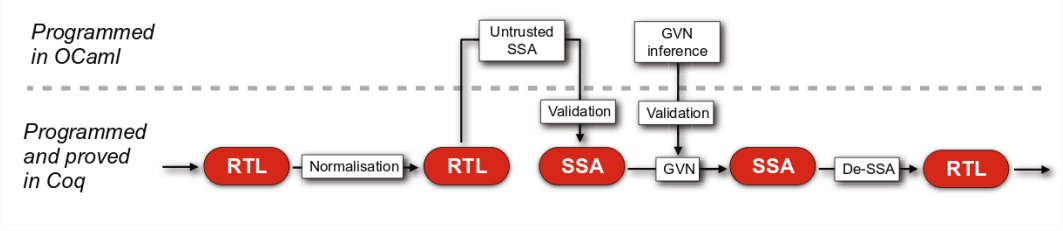
Intermediate Representations can help !

- IRs **decompose** compilation : simplification & modularity (proofs Compcert)
- Property : **simplify analyses and transformations** (SSA)
- **Design** and **juggle** with IRs (helped with validators : CompcertSSA, LLVM pipeline)
- **Refine** static analyses



CompcertSSA : a open door!

Opportunity for new analysis/optimizations and new IRs



Transformation SSA validated a posteriori

Focus on dependences and values

SSA

(1988)

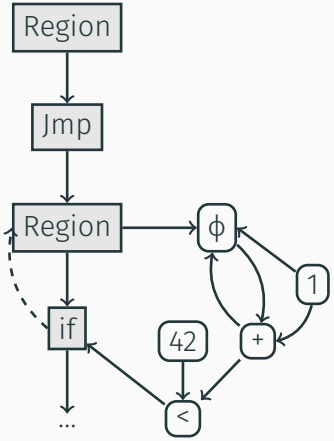
```
    ...
    x = 1

loop :
    ß = ϕ(x, x')
    x' = ß + 1
    c = x' < 42
    if c loop exit

exit :
    ...
```

Sea-of-Nodes

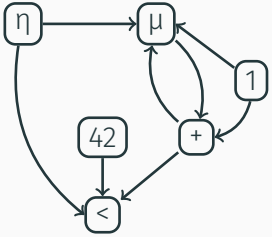
(Cliff Click 1995)



(HotSpot- Graal, LibFirm)

Synchronous Value Graph (Gated SSA)

(Paul Govereau 2011)



that's what I present now...

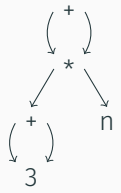
Value Graphs for validation ?

Idea : Paul Govereau LLVM-MD

Maximal sharing on graphs

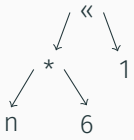
Simple example :

```
x_1 = 3 + 3
x_2 = a * x_1
x_3 = x_2 + x_2
```



Optimized simple example :

```
y_1 = a * 6
y_2 = y_1 << 1
```



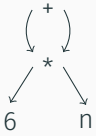
General $(3 + 3)$ and compiler-related $(x + x)$ reductions make the graphs converging

Idea : Paul Govereau LLVM-MD

Maximal sharing on graphs

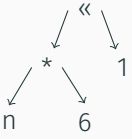
Simple example :

```
x_1 = 3 + 3
x_2 = a * x_1
x_3 = x_2 + x_2
```



Optimized simple example :

```
y_1 = a * 6
y_2 = y_1 << 1
```



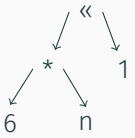
General $(3 + 3)$ and compiler-related $(x + x)$ reductions make the graphs converging

Idea : Paul Govereau LLVM-MD

Maximal sharing on graphs

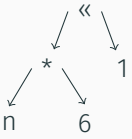
Simple example :

```
x_1 = 3 + 3
x_2 = a * x_1
x_3 = x_2 + x_2
```



Optimized simple example :

```
y_1 = a * 6
y_2 = y_1 << 1
```



General $(3 + 3)$ and compiler-related $(x + x)$ reductions make the graphs converging

Context (1) : SSA

Each variable is assigned **exactly once**, every variable is defined before it is used : *referential transparency*

Source program

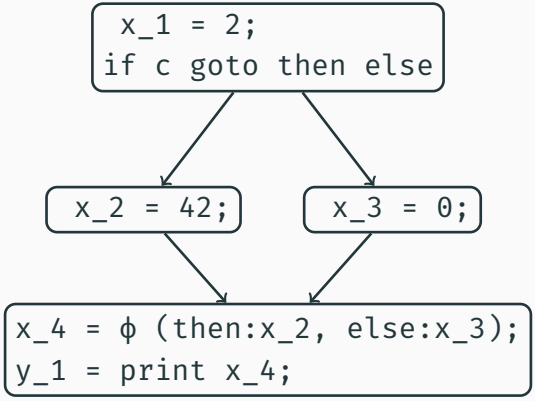
```
x = 0
y = x + 1
x = 1
z = x + 1
```

SSA program

```
x_1 = 0
y = x_1 + 1
x_2 = 1
z = x_2 + 1
```

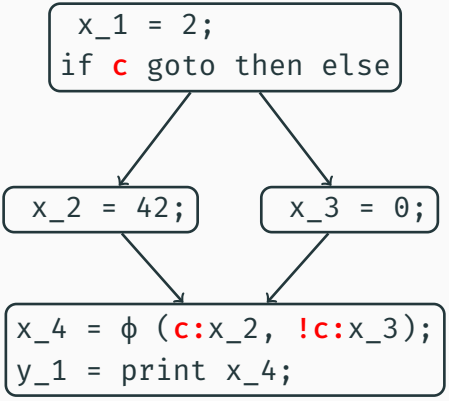
Context (2) SSA = ϕ

- ϕ-function **choose the right value** depending on the control flow
- Make the dependences explicit



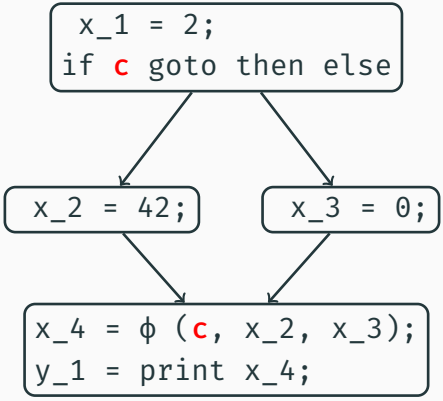
GSA : Gated Single Assignment [Ballance & al. SIGPLAN'90]

- Include control information in ϕ
- Simplify symbolic analyses



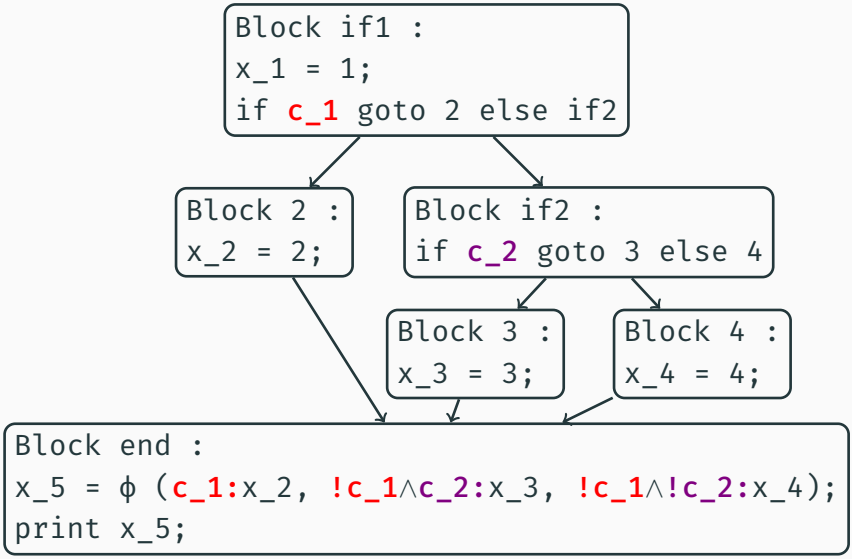
GSA : Gated Single Assignment [Ballance & al. SIGPLAN'90]

- Include control information in ϕ
- Simplify symbolic analyses



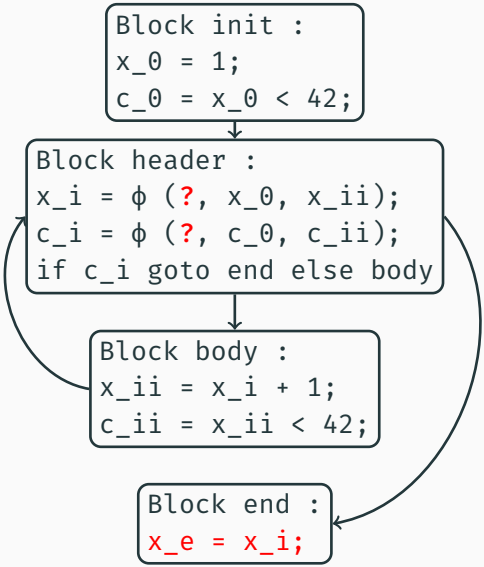
Gating ϕ

Gathering conditions for each predecessor until reaching immediat dominator



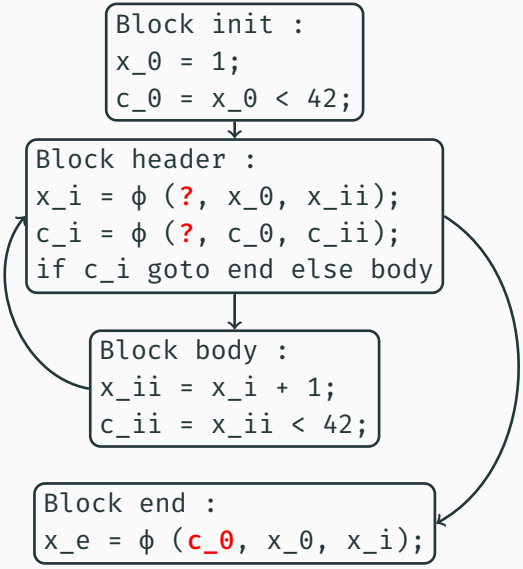
Gating Loops (μ & η)

A symbolic value for `x_e`? What guard for ϕ?



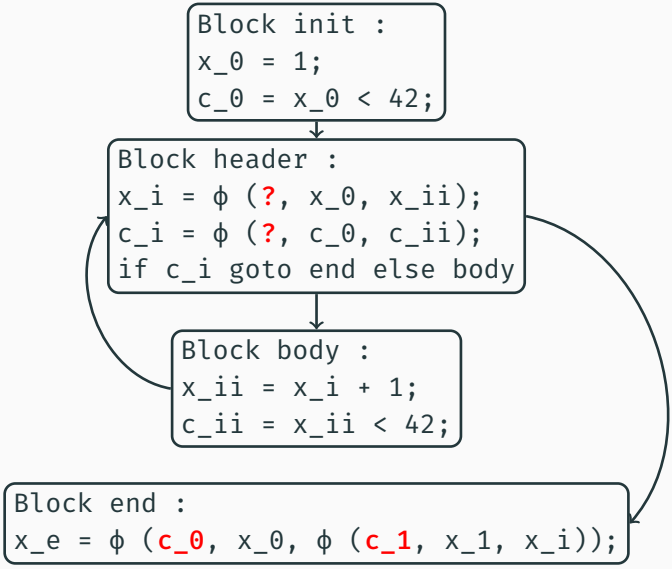
Gating Loops (μ & η)

If no iteration it should be ...

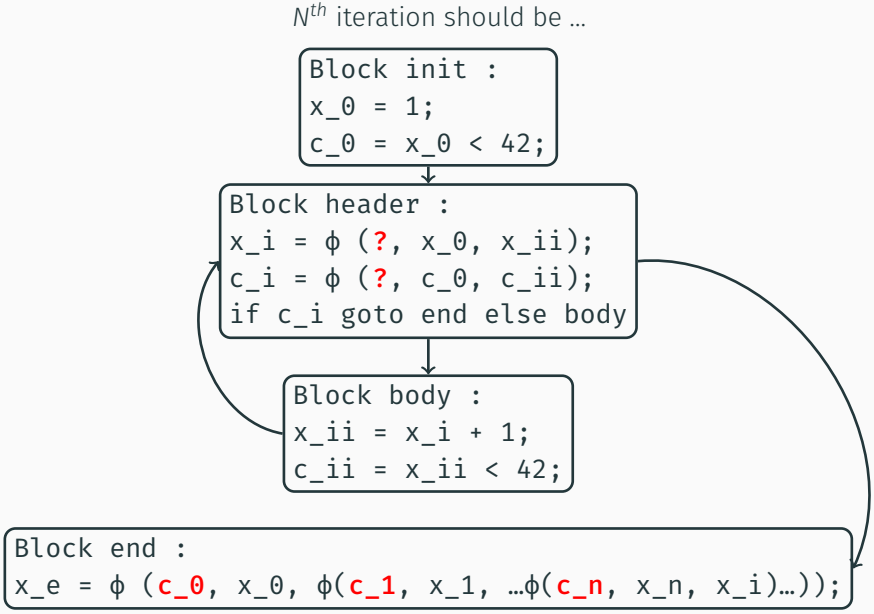


Gating Loops (μ & η)

First iteration should be ...

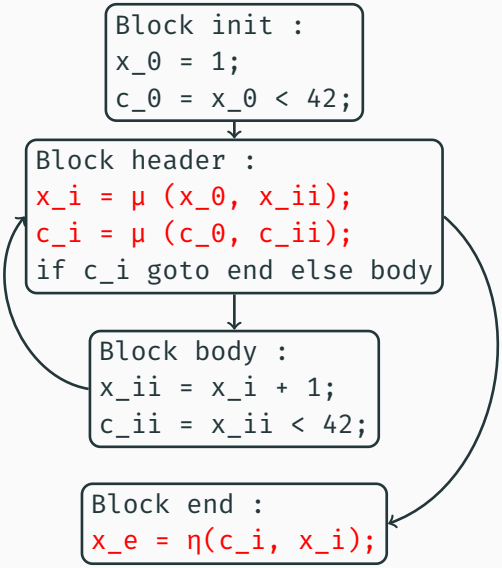


Gating Loops (μ & η)

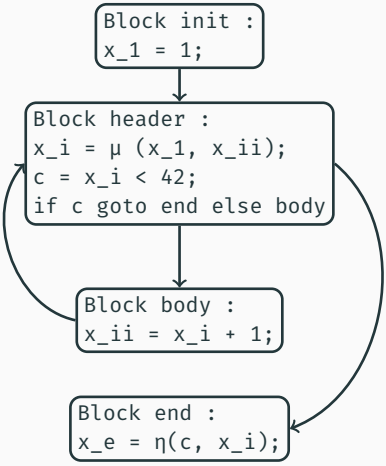


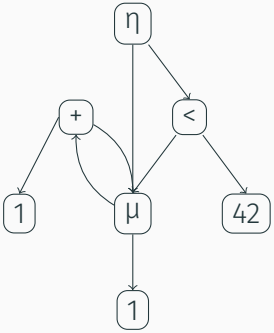
Gating Loops (μ & η)

μ initializes and defines variables modified in loops. η sets the out value with the guard.



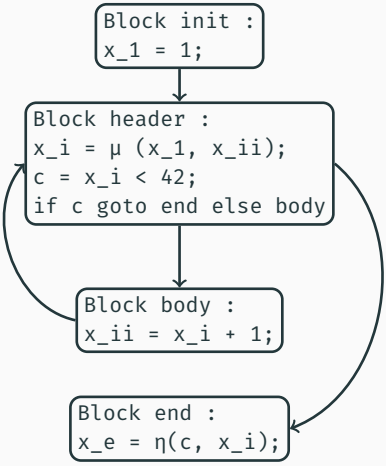
Synchronous Value Graph : building and hash-consing

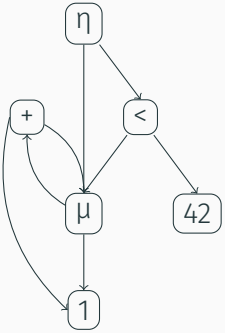




GVN for free !

Synchronous Value Graph : building and hash-consing

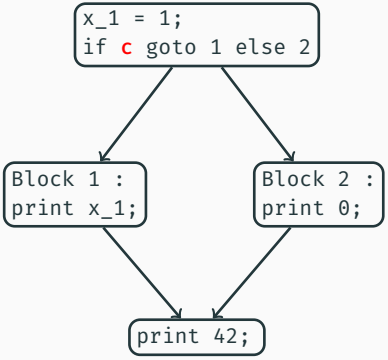


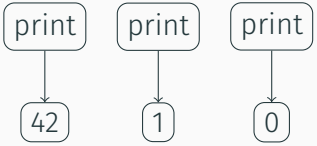


GVN for free !

Side effects

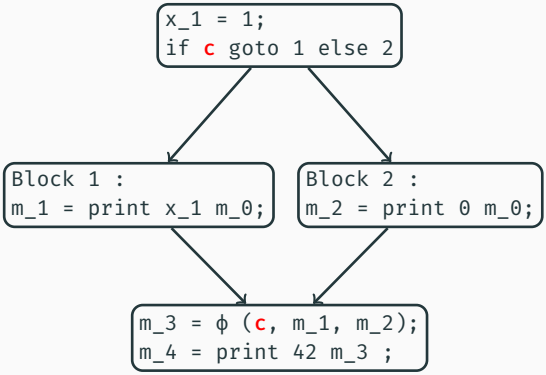
How do we **control** side effects ?

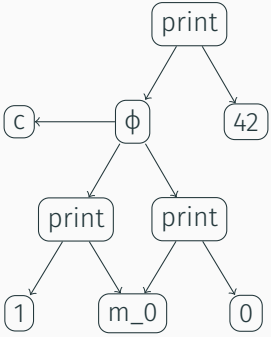




State dependency : MSSA

Introduction of abstract state variable `m` catching control dependencies between effects.





Reduction/rewrite rules

General rules

- `a = a` ↓ `true`
- `a ≠ a` ↓ `false`
- …

Optimization specific rules

- `a + a` ↓ `a << 1`
- `a * 4` ↓ `a << 2`
- `ϕ(…, true: x, …)` ↓ `x`
- `ϕ(…, false: x, …)` ↓ `ϕ(…, …)`
- `ϕ(…:x, …:x, …:x, …)` ↓ `x`
- `η(false, x, y)` ↓ `x`
- `η(c, μ(x, x))` ↓ `x`
- `η(c, y=μ(x, y))` ↓ `x`

Reduction/rewrite rules

General rules

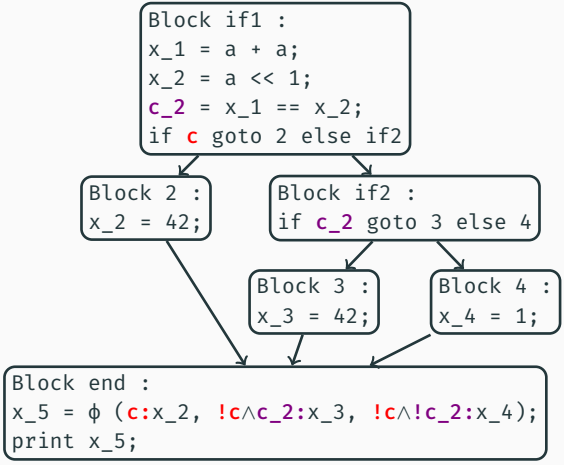
- `a = a` ↓ `true`
- `a ≠ a` ↓ `false`
- …

Optimization specific rules

- `a + a` ↓ `a << 1`
- `a * 4` ↓ `a << 2`
- `ϕ(…, true: x, …)` ↓ `x`
- `ϕ(…, false: x, …)` ↓ `ϕ(…, …)`
- `ϕ(…:x, …:x, …:x, …)` ↓ `x`
- `η(false, x, y)` ↓ `x`
- `η(c, μ(x, x))` ↓ `x`
- `η(c, y=μ(x, y))` ↓ `x`

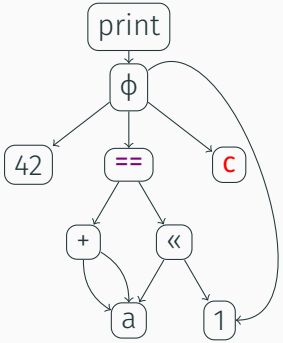
Mirror compiler optimization pipeline (transformations order matter)!

Reduction/rewrite rules : quick example

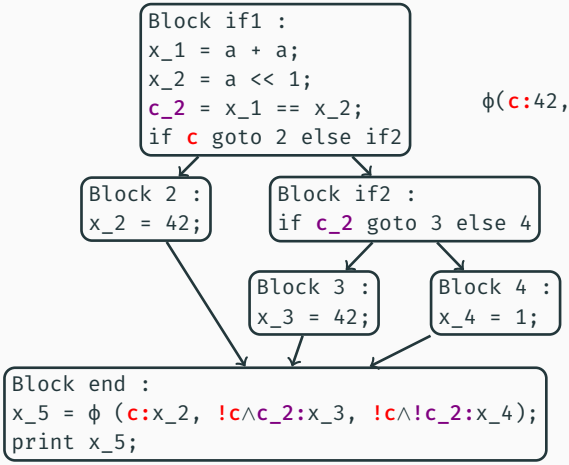


Simplify

ϕ(c:x_2, !c∧c_2:x_3, !c∧!c_2:x_4)

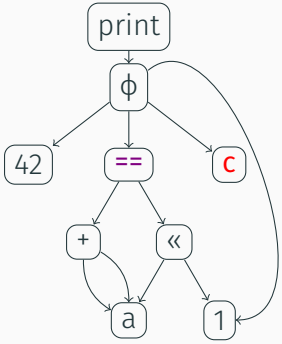


Reduction/rewrite rules : quick example

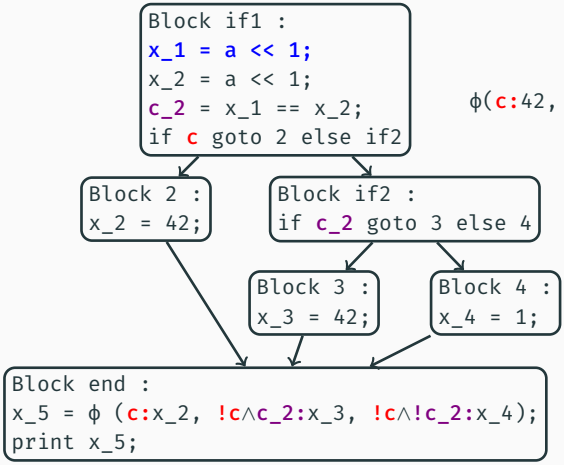


Next rule : `a + a` ↓ `a << 1`

ϕ(c:42, !c∧(a+a)==(a<<1):42, !c∧!(a+a)==(a<<1):1)

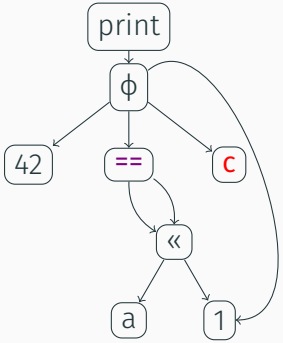


Reduction/rewrite rules : quick example

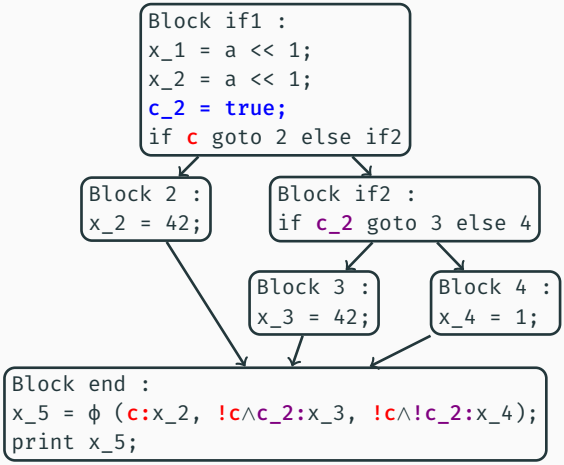


Next rule : `a = a ↓ true`

`ϕ(c:42, !c∧(a<<1)==(a<<1):42, !c∧!(a<<1)==(a<<1):1)`

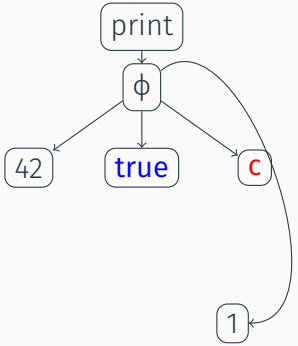


Reduction/rewrite rules : quick example

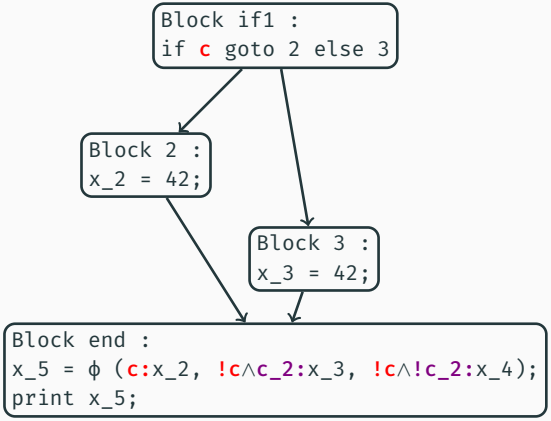


Next rule : ϕ(…, false: x, …)

ϕ(c:42, !c∧(true):42, !c∧!(true):1)

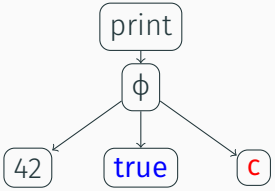


Reduction/rewrite rules : quick example



Next rule : ϕ(…:x, …:x, …:x, …) ↓ x

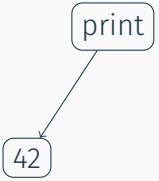
ϕ(c:42, !c∧(true):42)



Reduction/rewrite rules : quick example

Nothing to do ...

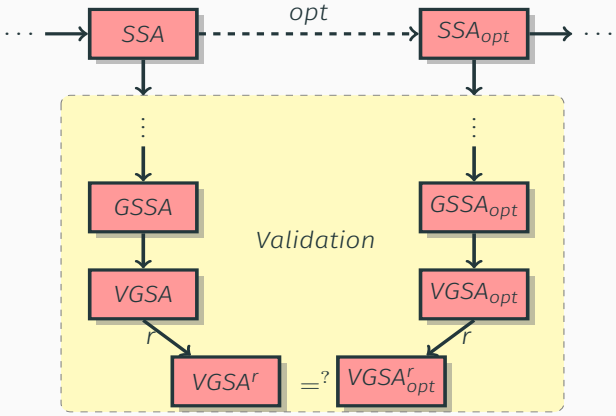
42




```
Block end :
print 42;
```

In CompcertSSA

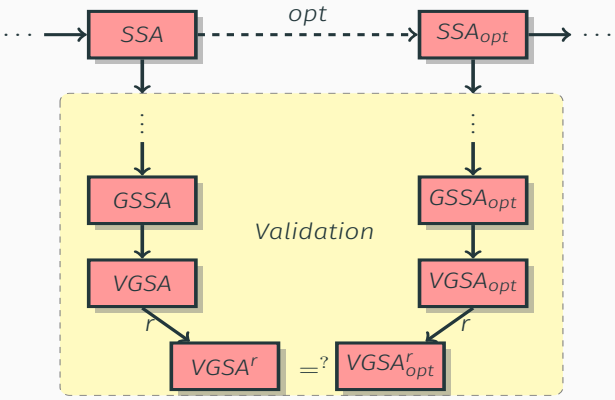
- Has been implemented on CompcertSSA (Ocaml)
- Compromises done on loops detection...
- Validate GVN and SCCP on our tests (when loop analysis is ok)



Conclusion

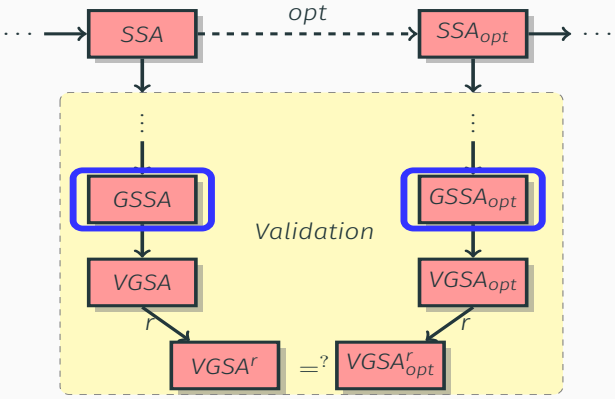
To be continued!

Formal part is still under construction...



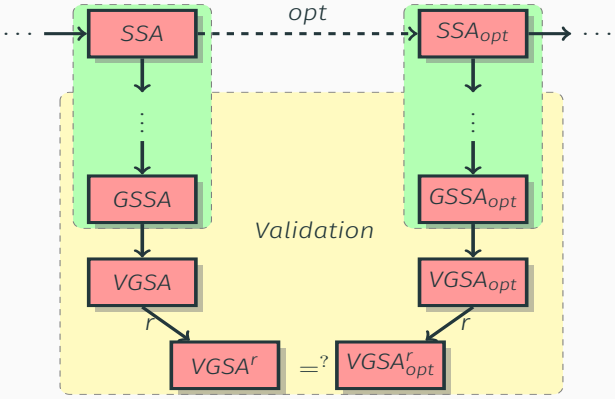
To be continued!

GSSA semantic OK!



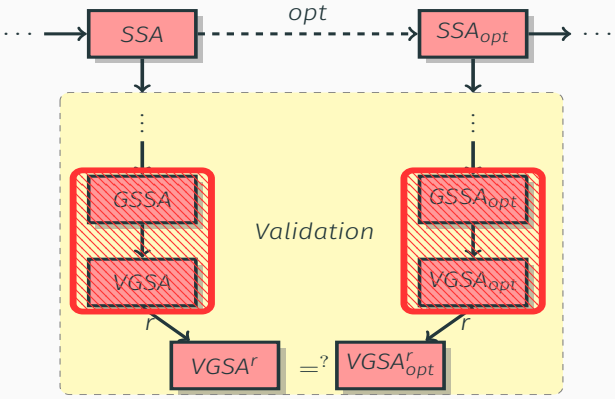
To be continued!

Connecting to the SSA semantic should be ok...



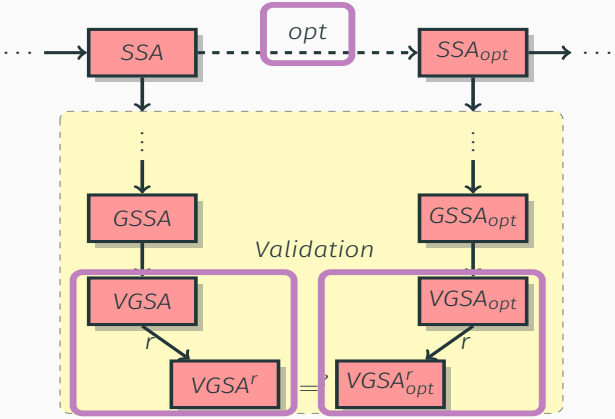
To be continued!

VGSA semantic not correct...



To be continued!

Validate more rules regarding to new optimizations



Questions?

Sémantique GSSA

$$\text{condT} : \frac{P(b)[pc] = (Icond\ cond\ args\ ifso\ ifnot) \quad evalcond\ cond\ args - Vr = True}{P \vdash (b, pc, Vr, m) \Rightarrow (ifso, 0, Vr, m)}$$

$$\text{condF} : \frac{P(b)[pc] = (Icond\ cond\ args\ ifso\ ifnot) \quad Vr(args) = vargs\ evalcond\ cond\ vargs = False}{P \vdash (b, pc, Vr, m) \Rightarrow (ifnot, 0, Vr, m)}$$

$$\text{phis} : \frac{P(b)[pc] = (Iphis\ memphi\ phis) \quad memphi = (((cs_1, m_1), \ldots, (cs_n, m_n)), m') \quad phi_k \in phis,\ phi_k = (((cs_{k_1}, r_{k_1}), \ldots, (cs_{k_n}, r_{k_n})), res_k) \quad t \in [1 \ldots n]\ m = m_t\ Vr(r_{k_t}) = v_{k_t}}{P \vdash (b, pc, Vr, m) \Rightarrow (b, pc + 1, \forall (k, t)\ Vr[res_k \leftarrow v_{k_t}], m)}$$

$$\text{mus} : \frac{P(b)[pc] = (Imu\ memu\ mus) \quad memu = (m_0, m_i, m') \quad mu_k \in mus,\ mu_k = (r_{k_0}, r_{k_i}, res_k) \quad m = m_0 \Rightarrow Vr(r_{k_0}) = v_k \quad m = m_i \Rightarrow Vr(r_{k_i}) = v_k}{P \vdash (b, pc, Vr, m) \Rightarrow (b, pc + 1, \forall k Vr[res_k \leftarrow v_k], m')}$$

$$\text{etas} : \frac{P(b)[pc] = (Ieta\ etas) \quad eta_k \in etas,\ eta_k = (c_k, r_k, res_k) \quad evalcond\ c_k = True,\ Vr(r_k) = v_k}{P \vdash (b, pc, Vr, m) \Rightarrow (b, pc + 1, \forall k Vr[res_k \leftarrow v_k], m)}$$

$$\text{obs} : \frac{P(b)[pc] = (Iobs\ r\ res\ m\ m') \quad Vr(r) = vr \quad call\ vr\ m\ vres\ e}{P \vdash (b, pc, Vr, m) \overset{e}{\Rightarrow} (b, pc + 1, Vr[res \leftarrow vres], m')}$$

$$\text{return} : \frac{P(b)[pc] = (Ireturn\ r\ m\ m')}{P \vdash (b, pc, Vr, m) \Rightarrow (Vr(r), m')}$$

Sémantique VGSA (data-flow)

$$\text{evt-step}\ \frac{eval\_n(i)(n) = ((\_, true), e)}{G, P \vdash (i, eval\_n(i)) \overset{e}{\Rightarrow} (i+1, eval\_n(i+1))}$$

$$\epsilon\text{-step}\ \frac{eval\_n(i)(n) = (\_, \epsilon)}{G, P \vdash (i, eval\_n(i)) \Rightarrow (i+1, eval\_n(i+1))}$$

Le langage

```
type instructions =
| Icst of int * reg
| Iop of op * reg * reg * reg
| Icond of cond * reg list * label * label
| Iphis of ((cond list * memreg) list * memreg) * ((cond list * reg) list * reg) list
| Imus of (memreg * memreg * memreg) * (reg * reg * reg) list
| Ieta of cond * reg * reg
| Iobs of reg * reg * memreg * memreg
| Ijmp of label
| Ireturn of reg option * memreg * memreg
```