SSAFire : Formalizing Monadic Gated SSA and its Optimizations

This material is based upon work supported by grant ANR 14-CE28-0004.

Thomas Rubiano joint work with Delphine Demange

18 juin 2020

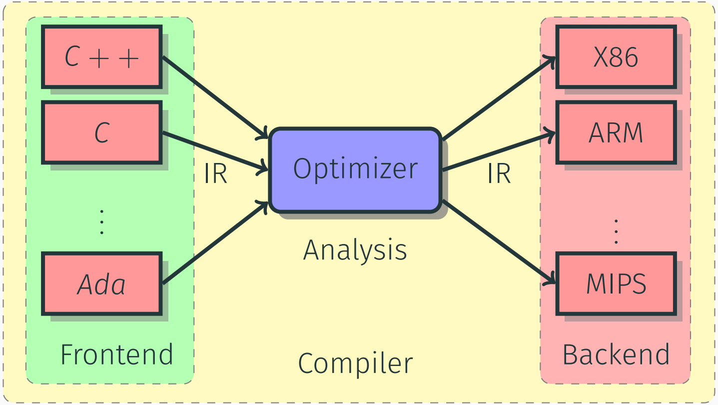
IRISA
Université Rennes 1

Outline

Outline

Problem : How to verify Optimizing compilers ?

Compilers look like



Problem : How to verify Optimizing compilers ?

Common Compiler **optimizing pipeline** be like

```
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
```

Most transformations need analysis of the **dependencies** between instructions

Complex and interdependent transformations may imply bugs

Problem : How to verify Optimizing compilers ?

How to formally verify those transformations ?

Verified Compilers : Theorem

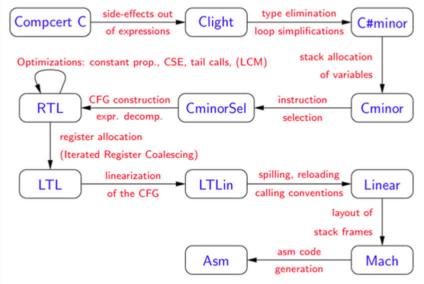
$$\textit{Source} \longrightarrow \textit{Compiler} \longrightarrow \textit{Assembler}$$

$$\forall \ \textit{behaviours} \ \ B \notin \textit{wrong}, \ \ \textit{Assembler} \Downarrow B \ \ \Rightarrow \ \ \textit{Source} \Downarrow B$$

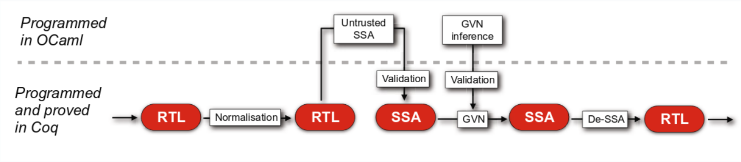
Verified Compilers : Intermediate Representations for verification purpose

- IRs **decompose** compilation : simulation simplification & modularity
- **simplify analyses and transformations**

Compcert

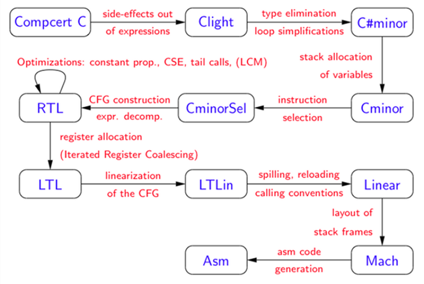


CompcertSSA

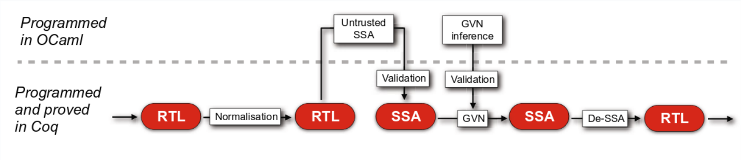


- IRs **decompose** compilation : simulation simplification & modularity
- **simplify analyses and transformations**

Compcert



CompcertSSA

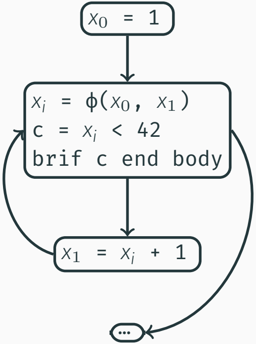


Simplifying transformation expression = simplifying verification

Transformation proofs techniques need to focus on dependences and values

SSA

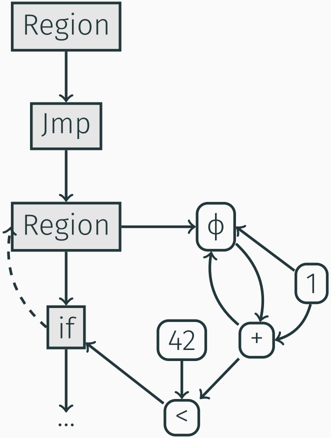
Rosen et al. (1988)



Basic dependencies
But depends on **control flow graph** (CFG)
(now used in many compilers)

Sea-of-Nodes

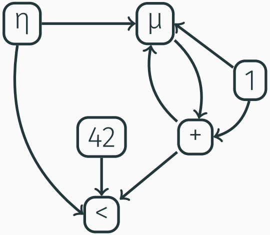
(Cliff Click 1995)



Sequentialized dependence graph
Regions reflect the CFG
(Used in HotSpot- Graal and LibFirm compilers)

(Monadic) Gated SSA

(Ottenstein et al. 1990)



Program Dependence Graph+SSA
Including control flow informations
Monadic : includes variables representing memory
(That is SSAFire I present next...)

Semantic correctness of aggressive, global optimizations

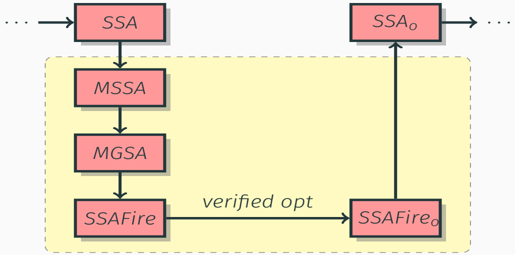
Most of verification efforts use SSA heavily based on CFG
It **relies on dominance** relation to recover dependence informations

In Monadic Gated SSA **dominance is no longer required** when semantically reasoning about optimizations correctness

What do we want?

We want the *transparency* of such dependence graph but also **simple, scalable and elegant proofs of transformations.**

⇒ A Program Dependence Graph with **operational** semantics.



Outline

Context

SSA and extensions

SSAFire : Syntax and Semantic

SSAFire : optimizations

Experiments

Conclusion

SSA [Rosen & al 1988] and Program Dependence Graph [Ferrante & al 1987]

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

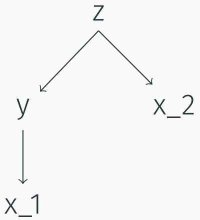
Source program

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

SSA program

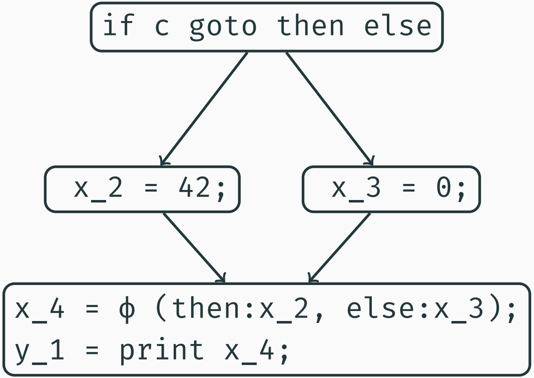
```
x_1 = 0
y = x_1 + 1
x_2 = 1
z = x_2 + y
```

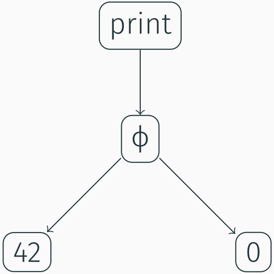
Dependence Graph



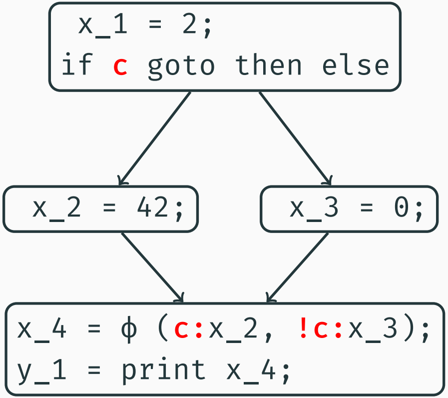
SSA = ϕ [Rosen & al 1988]

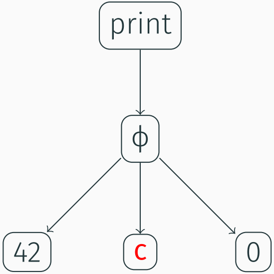
- ϕ-function **choose the right value** depending on the control flow
- Make the dependences explicit





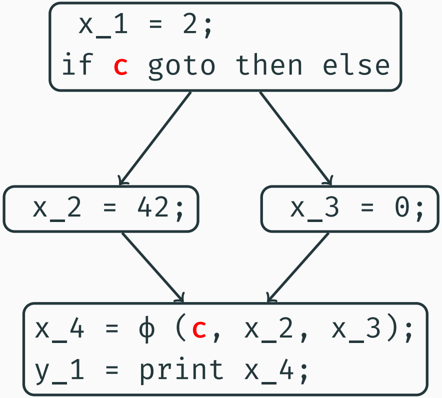
Include control information in ϕ

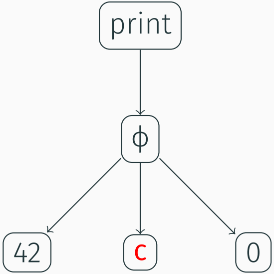




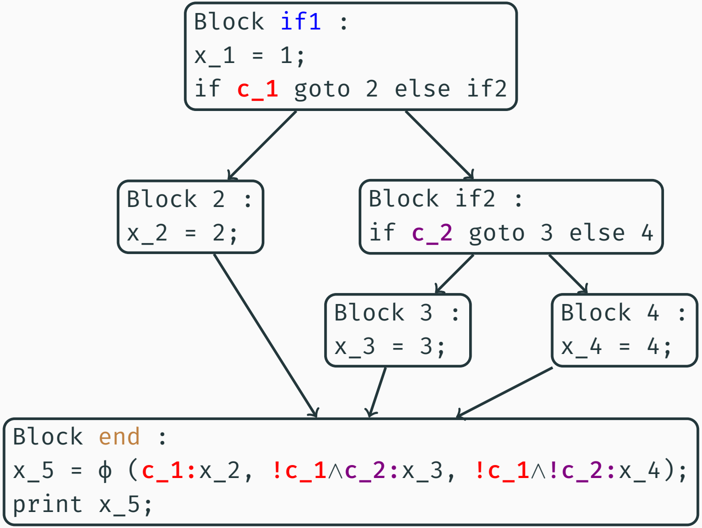
GSA : Gated Single Assignment [Ballance & al.1990]

We can simplify notation when it's a ϕ with only 2 variables



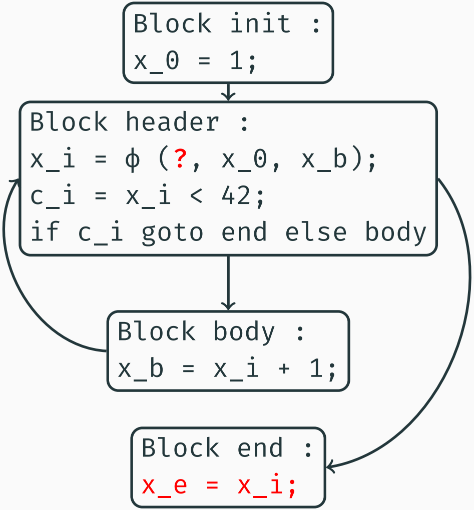


Tarjan ‘unambiguous path expression‘ from end’s immediat dominator to end.

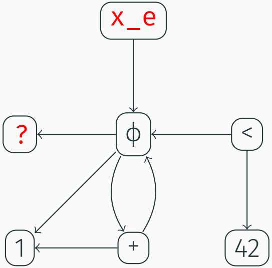


Gating Loops (μ & η)

What guard for ϕ? How to choose between **initialization** and **iteration**?

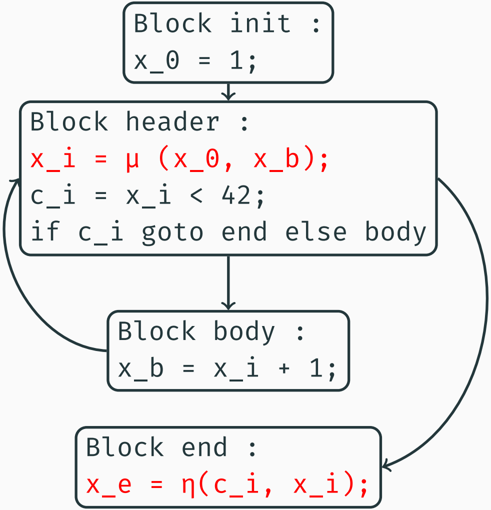


How controlling **x_e**?

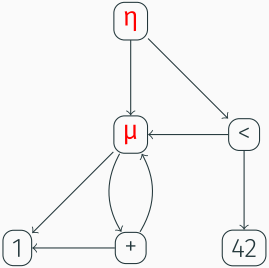


Gating Loops (μ & η)

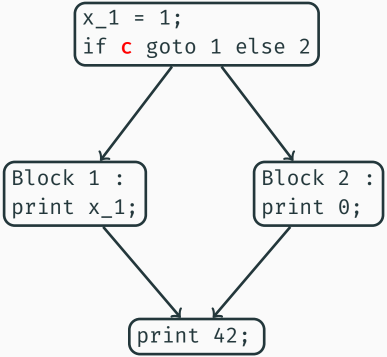
μ initializes and defines variables modified in loops.

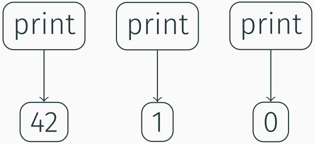


η sets the out value with the guard



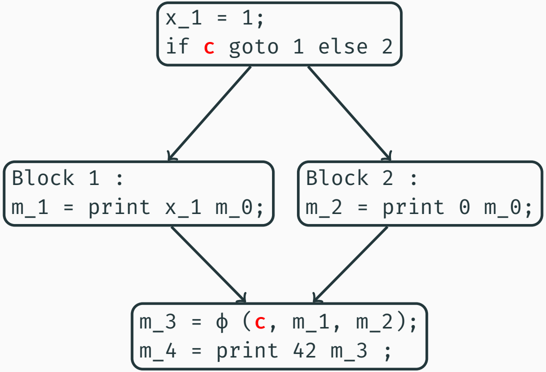
How do we **control** effects dependencies ?

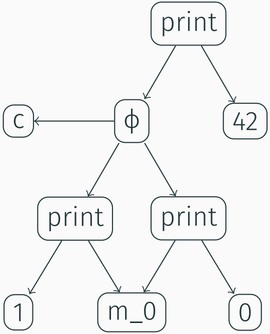




State dependency : Monadic SSA

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

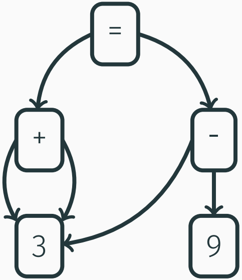




Outline

Constants, operators, comparisons...

$$\begin{array}{lcl} \text{constant literals} & \in & \text{Consts} = \{\textit{false}, \textit{true}, \ldots, -1, 0, 1, \ldots\} \\ \text{operators} & \in & \text{Ops} = \{+, -, \ldots\} \\ \text{comparisons} & \in & \text{Conds} = \{=, <, \text{not}, \ldots\} \end{array}$$



Dependence Graph : arrows show what a node needs to be evaluable

Abstract Syntax : two kind of terms

nodes id	n, n_i, n_c, n_l	$\in$	$\mathcal{N}$
term graphs	$g \in \mathcal{G}$	$=$	$\mathcal{N} \hookrightarrow \mathcal{T}_\mathrm{v} \cup \mathcal{T}_\mathrm{m}$

scalar terms

$$
\begin{aligned}
\mathcal{T}_\mathrm{v} \ni vt ::= \quad & \mathrm{var} \\
| \quad & \mathrm{cst}\ k \\
| \quad & \mathrm{op}\ o\ [n_1, \ldots, n_j] \mid \mathrm{cond}\ c\ [n_1, \ldots, n_j] \\
| \quad & \eta\ n_c\ n \\
| \quad & \phi\ (\gamma_s, n)_i \\
| \quad & \mu_b\ \gamma_a\ n_i\ n_l
\end{aligned}
$$

memory terms

$$
\begin{aligned}
\mathcal{T}_\mathrm{m} \ni mt ::= \quad & \mathrm{mvar} \\
| \quad & \mathrm{print}\ n\ m \\
| \quad & \mathrm{m}\eta\ n_c\ m \\
| \quad & \mathrm{return}\ \gamma_a\ n\ m \\
| \quad & \mathrm{m}\phi\ \gamma_a\ (\gamma_s, m)_i \\
| \quad & \mathrm{m}\mu_b\ \gamma_a\ m_i\ m_l
\end{aligned}
$$

Differenciate “scalar terms” and “memory terms”

Gates : activation ≠ selection

nodes id	n, n_i, n_c, n_l	$\in$	$\mathcal{N}$
gates in DNF	$\gamma, \gamma_a, \gamma_s$	$\in$	$\wp(\wp(\mathcal{N}))$

scalar terms

$$
\begin{array}{rl}
\mathcal{T}_V \ni vt ::= & \text{var} \\
\mid & \text{cst } k \\
\mid & \text{op } o\ [n_1, \ldots, n_j] \mid \text{cond } c\ [n_1, \ldots, n_j] \\
\mid & \eta\ n_c\ n \\
\mid & \phi\ (\gamma_s, n)_i \\
\mid & \mu_b\ \gamma_a\ n_i\ n_l
\end{array}
$$

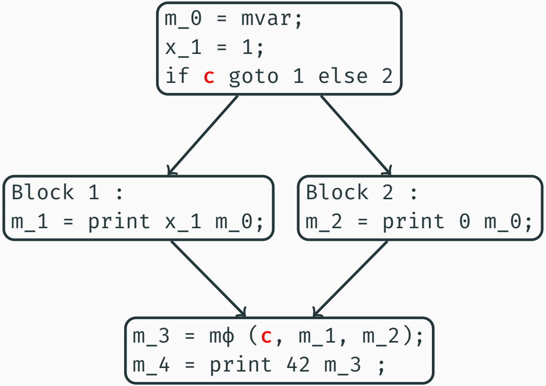
memory terms

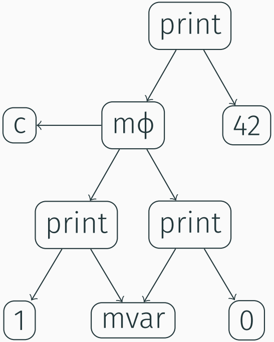
$$
\begin{array}{rl}
\mathcal{T}_m \ni mt ::= & \text{mvar} \\
\mid & \text{obs } n\ m \\
\mid & m\eta\ n_c\ m \\
\mid & \text{ret } \gamma_a\ n\ m \\
\mid & m\phi\ \gamma_a\ (\gamma_s, m)_i \\
\mid & m\mu_b\ \gamma_a\ m_i\ m_l
\end{array}
$$

The gates (in DNF form) : for “selection” but also “activation”

Activation gates

We've seen that the choice of the right "trace" is made by $\mathtt{m}_\phi$...

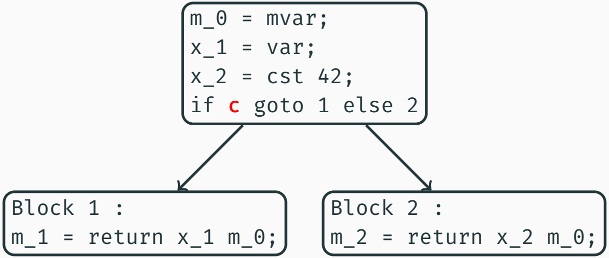




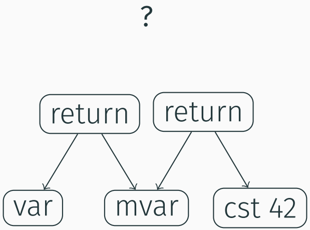
Other cases where we don't want to have several memory-variables defined at same state !

SSAFire has no information in order to choose between the two **`return`**

MSSA CFG



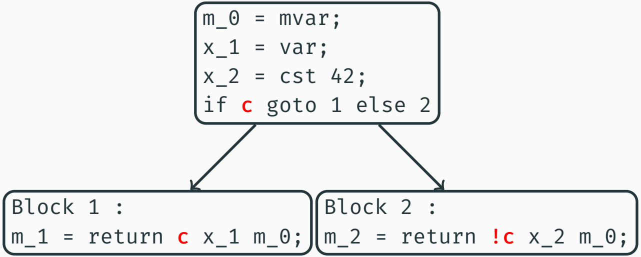
SSAFire



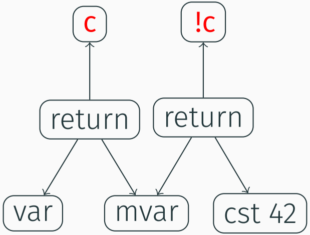
Also necessary of $m\phi$, $m\mu_b$ and μ_b nodes...

Activation gates embed control information into nodes using same memory

MSSA CFG



SSAFire



Also necessary of $m\phi$, $m\mu_b$ and μ_b nodes...

Syntax : $_\mu$block

$$
\begin{array}{lrcl}
\text{nodes id} & n, n_i, n_c, n_l & \in & \mathcal{N} \\
\text{block id} & b & \in & \mathcal{B} \\
\text{gates in DNF} & \gamma, \gamma_a, \gamma_s & \in & \wp(\wp(\mathcal{N}))
\end{array}
$$

scalar terms

$$
\begin{array}{rcl}
\mathcal{T}_\mathrm{V} \ni vt ::= & & \text{var} \\
& | & \text{cst } k \\
& | & \text{op } o\ [n_1, \ldots, n_j] \mid \text{cond } c\ [n_1, \ldots, n_j] \\
& | & \eta\ n_c\ n \\
& | & \phi\ (\gamma_s, n)_i \\
& | & \mu_b\ \gamma_a\ n_i\ n_l
\end{array}
$$

memory terms

$$
\begin{array}{rcl}
\mathcal{T}_\mathrm{m} \ni mt ::= & & \text{mvar} \\
& | & \text{print } n\ m \\
& | & \text{m}\eta\ n_c\ m \\
& | & \text{return } \gamma_a\ n\ m \\
& | & \text{m}\phi\ \gamma_a\ (\gamma_s, m)_i \\
& | & \text{m}\mu_b\ \gamma_a\ m_i\ m_l
\end{array}
$$

"scalar terms" and "memory terms"

Syntax : $_\mu$block

nodes id	n, n_i, n_c, n_l	$\in$	$\mathcal{N}$
block id	b	$\in$	$\mathcal{B}$
gates in DNF	$\gamma, \gamma_a, \gamma_s$	$\in$	$\wp(\wp(\mathcal{N}))$

scalar terms

$$\begin{aligned}
\mathcal{T}_\mathrm{V} \ni vt ::= \quad & \mathrm{var} \\
\mid \quad & \mathrm{cst}\ k \\
\mid \quad & \mathrm{op}\ o\ [n_1, \ldots, n_j] \mid \mathrm{cond}\ c\ [n_1, \ldots, n_j] \\
\mid \quad & \eta\ n_c\ n \\
\mid \quad & \phi\ (\gamma_s, n)_i \\
\mid \quad & \mu_b\ \gamma_a\ n_i\ n_l
\end{aligned}$$

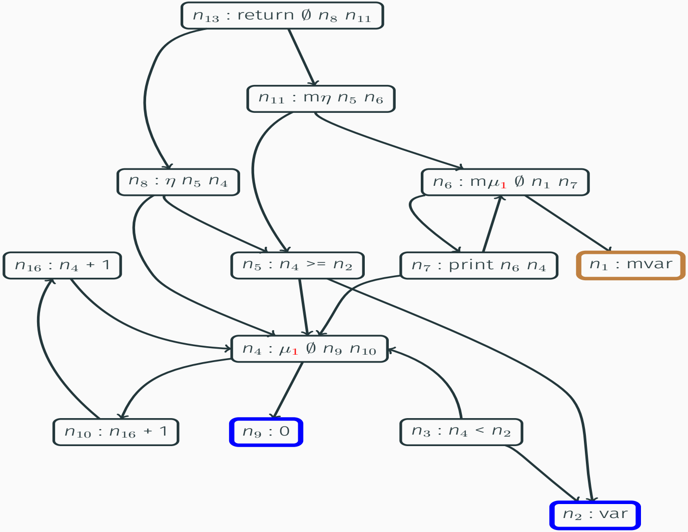
memory terms

$$\begin{aligned}
\mathcal{T}_\mathrm{m} \ni mt ::= \quad & \mathrm{mvar} \\
\mid \quad & \mathrm{print}\ n\ m \\
\mid \quad & \mathrm{m}\eta\ n_c\ m \\
\mid \quad & \mathrm{return}\ \gamma_a\ n\ m \\
\mid \quad & \mathrm{m}\phi\ \gamma_a\ (\gamma_s, m)_i \\
\mid \quad & \mathrm{m}\mu_b\ \gamma_a\ m_i\ m_l
\end{aligned}$$

$_\mu$block : `synchronises` $_\mu$nodes of the same loop

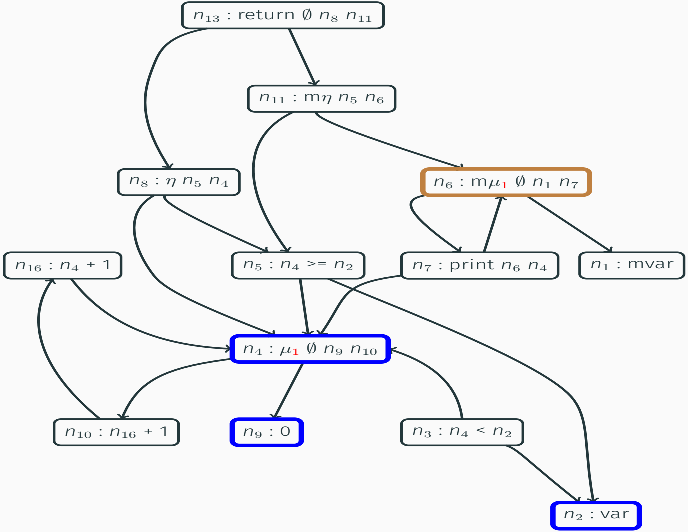
μblock

```
int main(int n){
    int i=0;
    while (i<n){
      print i;
      i++;
      i++;
    }
    return i;
}
```



Constants, input values and state are leaves

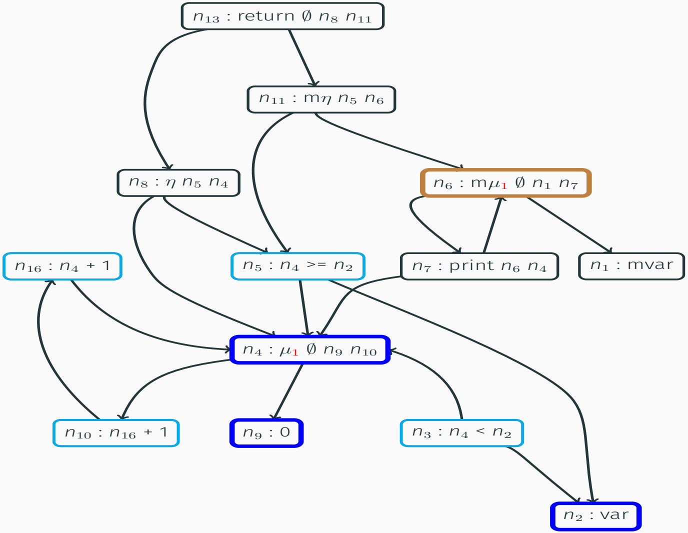
```
int main(int n){
    int i=0;
 →  while(i<n){
      print i;
      i++;
      i++;
    }
    return i;
}
```



Initialization of μ_1 nodes; Constants always evaluable but mvar "consumed"

μblock

```
int main(int n){
    int i=0;
    while(i<n){
      print i;
      i++;
      i++;
    }
    return i;
}
```



μ block synchronisation : μ_x evaluates only when all $(m)\mu_x$ are evaluable

SSAFIRE semantic evaluations and transition relations

We define a program state (or configuration) as :

$$\sigma = (n_m, \rho)$$

Where n_m is the current memory-state node
and ρ a map from value-state nodes to their value

Transition relation (or step) is defined as :

$$(m, g) \models \sigma_1 \xrightarrow{[v_1, \ldots, v_j]} \sigma_2$$

Where $[v_1, \ldots, v_j]$ is an effect sequence
(here simplified to a list of printed values...)

State nodes

$$\sigma = (n_m, \rho)$$

State nodes are nodes defining a program state :

Memory-state nodes define the state's memory

$$\in \{\texttt{return}, \texttt{m}\phi, \texttt{m}\mu, \texttt{m}\eta, \texttt{mvar}\}$$

State nodes

$$\sigma = (n_m, \rho)$$

State nodes are nodes defining a program state :

Memory-state nodes define the state's memory

$$\in \{\texttt{return}, \texttt{m}\phi, \texttt{m}\mu, \texttt{m}\eta, \texttt{mvar}\}$$

Value-state nodes define the state's values

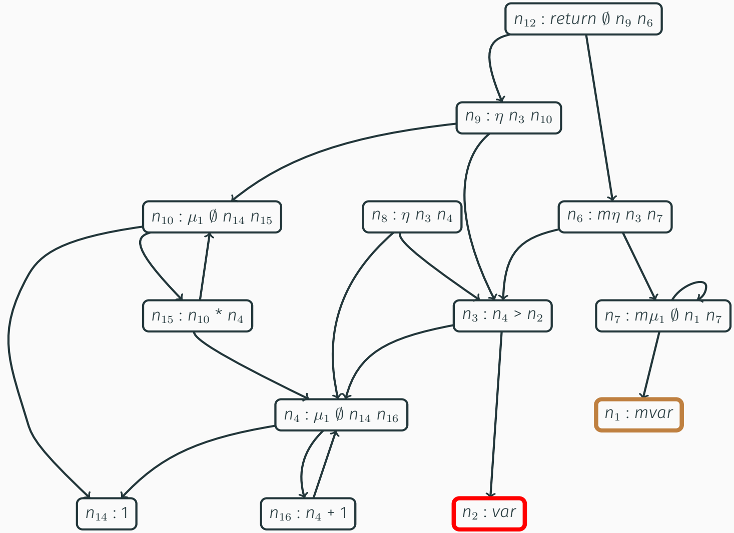
$$\in \{\mu, \eta, \texttt{var}\}$$

Value nodes

Value nodes are all other nodes, their value for the current state can be computed from the States nodes.

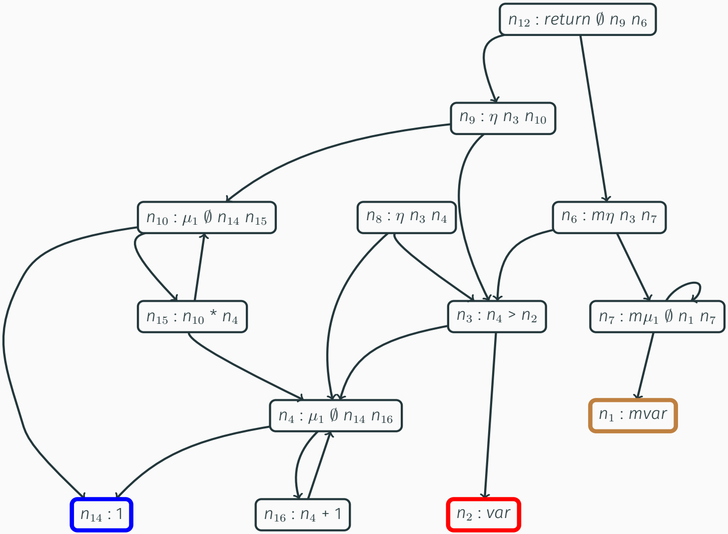
$$\in \{\texttt{cst}, \texttt{op}, \texttt{cond}, \phi, \texttt{print}\}$$

```
→int main(int n){
    int i=1;
    int fact=1;
    while (i<=n) {
        fact=fact*i;
        i=i+1;
    }
    return fact;
}
```



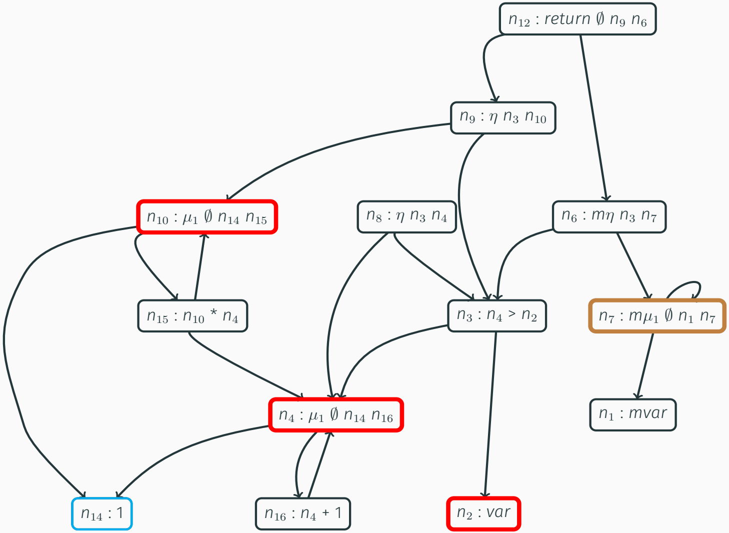
Initial configuration $\sigma_0 = (n_1, [n_2 \to 2])$

```
int main(int n){
  int i=1;
  int fact=1;
  while (i<=n) {
    fact=fact*i;
    i=i+1;
  }
  return fact;
}
```



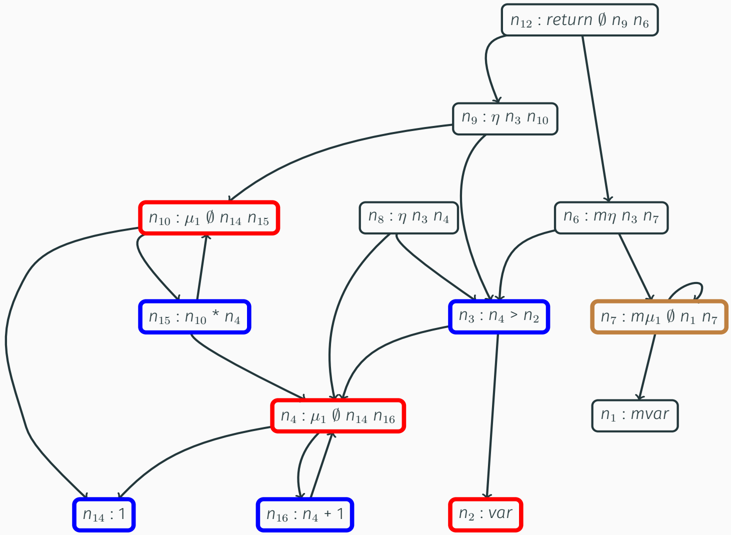
State node evaluation calls value node evaluation using σ_0

```
int main(int n){
    int i=1;
    int fact=1;
→   while (i<=n) {
        fact=fact*i;
        i=i+1;
    }
    return fact;
}
```



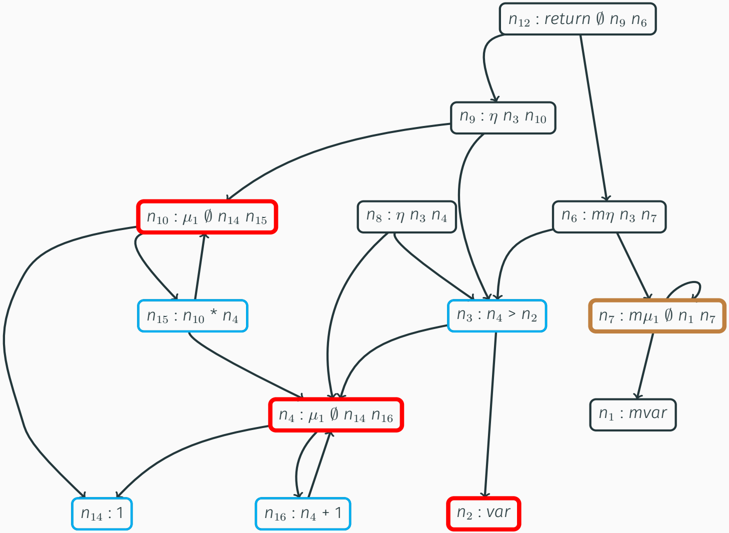
$$\sigma_1 = (n_7, [n_2 \to 2; n_4 \to 1; n_{10} \to 1])$$

```
int main(int n){
  int i=1;
  int fact=1;
  while (i<=n){
    fact=fact*i;
    i=i+1;
  }
  return fact;
}
```



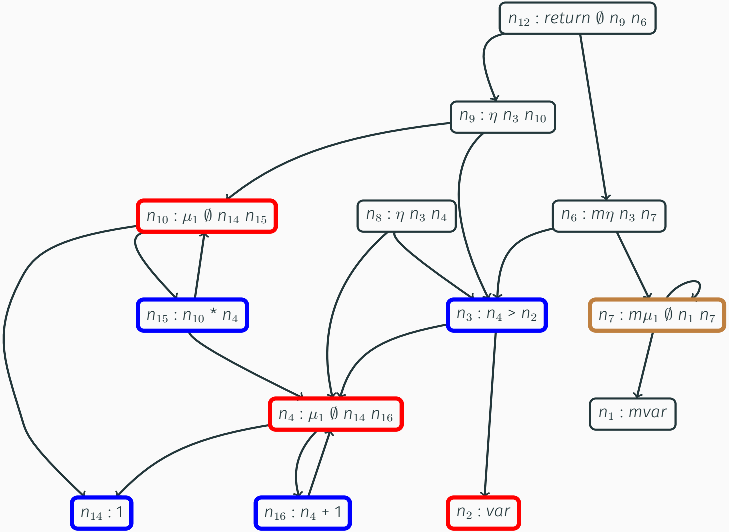
$$\sigma_1 \longrightarrow \sigma_2$$

```
int main(int n){
    int i=1;
    int fact=1;
→   while (i<=n) {
        fact=fact*i;
        i=i+1;
    }
    return fact;
}
```



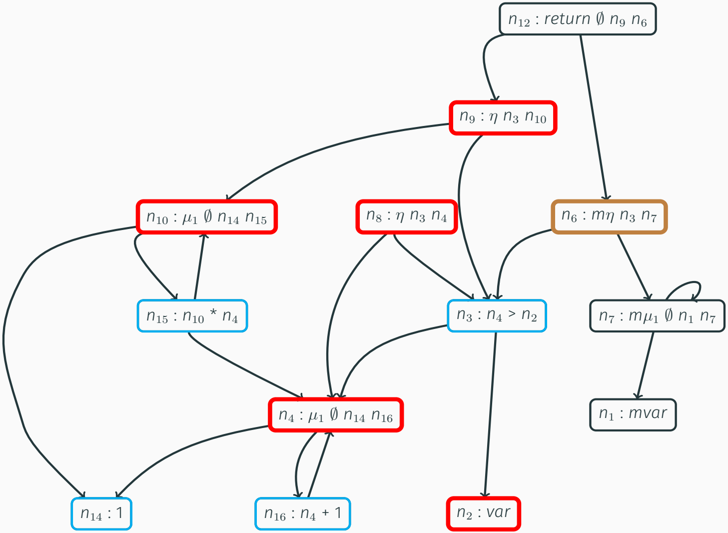
$$\sigma_2 = (n_7, [n_2 \to 2; n_4 \to 2; n_{10} \to 1])$$

```
int main(int n){
  int i=1;
  int fact=1;
  while(i<=n){
     fact=fact*i;
     i=i+1;
  }
  return fact;
}
```



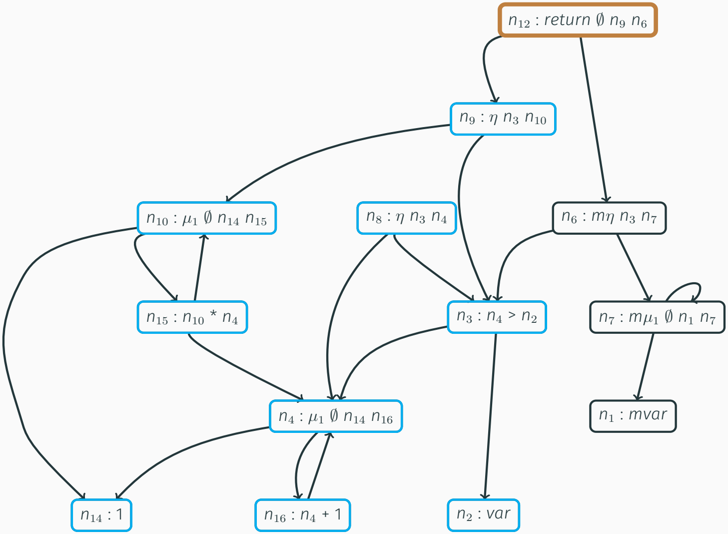
$$\sigma_2 \longrightarrow \sigma_3$$

```
int main(int n){
   int i=1;
   int fact=1;
   while (i<=n) {
      fact=fact*i;
      i=i+1;
   }
   return fact;
}
```



$$\sigma_3 = (n_6, [n_2 \to 2; n_4 \to 3; n_{10} \to 2; n_8 \to 3; n_9 \to 2])$$

```
int main(int n){
    int i=1;
    int fact=1;
    while (i<=n) {
        fact=fact*i;
        i=i+1;
    }
→   return fact;
}
```

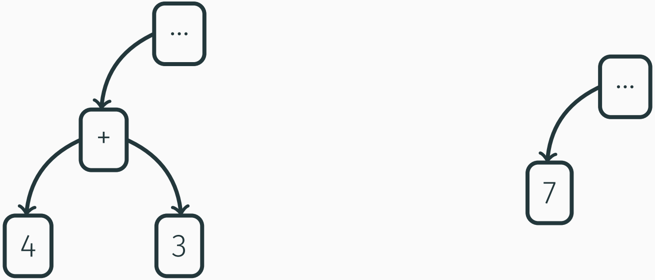


$$\sigma_4 = (n_{12}, 2)$$

Outline

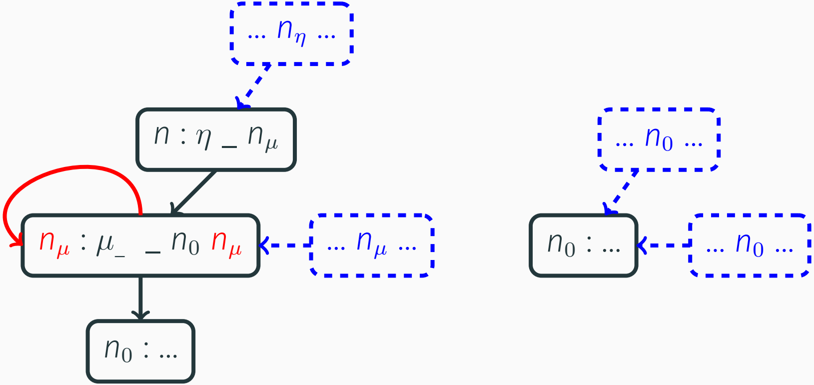
Constant folding

$$\text{CF1}\ \frac{g(n) = \text{op } o\ [n_{arg_1}, \ldots, n_{arg_j}] \qquad \forall i, g(n_{arg_i}) = \text{cst } v_i}{g \rightsquigarrow_n g[n \leftarrow \text{cst } [\![o]\!] v_1 \ldots v_k]}$$



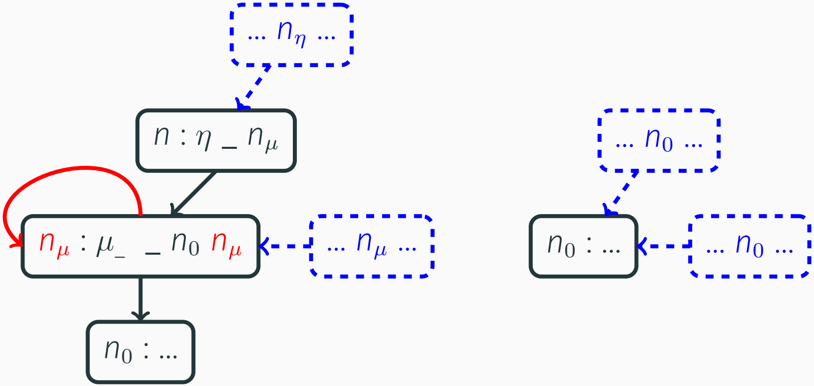
Transformation of node n into a precomputed constant

$$\textsc{Licm1}\ \frac{g(n) = \eta\ \_\ n_\mu \qquad g(n_\mu) = \mu_\_\ \_\ n_0\ n_\mu}{g \rightsquigarrow_n g[n_\mu/n_0][n/n_0]}$$



Loop invariant : itself as iteration argument

$$\textsc{Licm1}\ \frac{g(n) = \eta\ \_\ n_\mu \qquad g(n_\mu) = \mu_\_\ \_\ n_0\ n_\mu}{g \rightsquigarrow_n g[n_\mu/n_0][n/n_0]}$$



Replacement of n_μ and n by initial value n_0

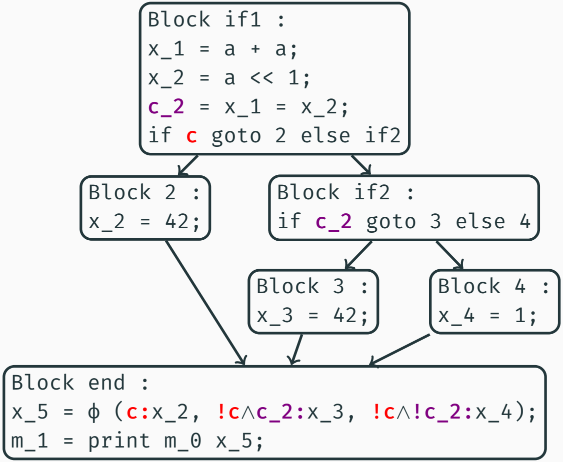
Branch Merging

$$\text{BM}\;\frac{g(n) = \phi\ (\gamma_s, n_s)_{i \in I} \qquad (\gamma'_s)_i = \left(\{\gamma_{sj} \mid j \in I, n_{sj} = n_{si}\}\right)_i \qquad t_\phi = \phi\ (\gamma'_{si}, n_{si})}{g \rightsquigarrow_n g[n \leftarrow t_\phi]}$$

$n : \phi\ [\ldots;(\gamma_{sj}, n_e);\ldots;(\gamma_{si}, n_e);\ldots]$

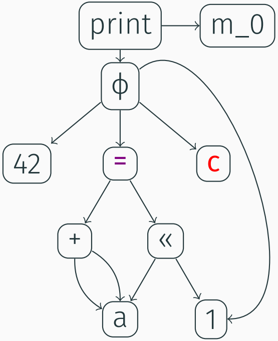
$n : \phi\ [\ldots;(\gamma_{sj} \vee \gamma_{si}, n_e);\ldots]$

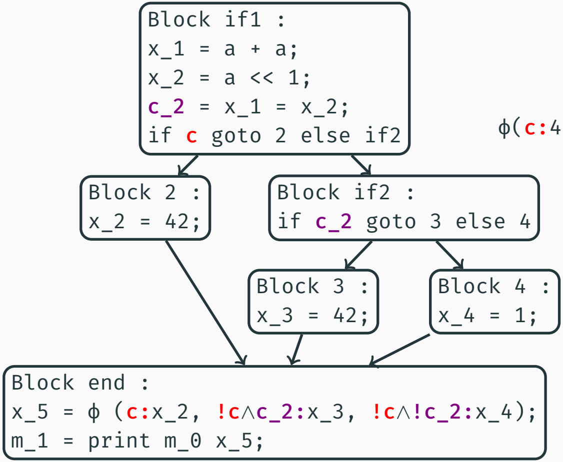
Merge branch : two branches returning same value



Focus on ϕ instruction :

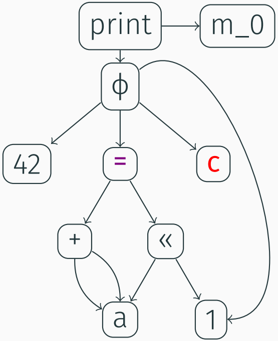
ϕ(c:x_2, !c∧c_2:x_3, !c∧!c_2:x_4)



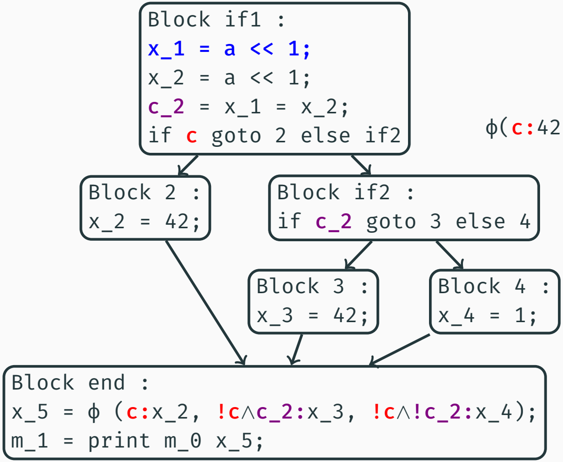


a + a ⇝ a << 1

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

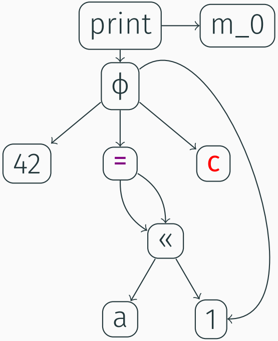


Atomic transformation : quick simplified example

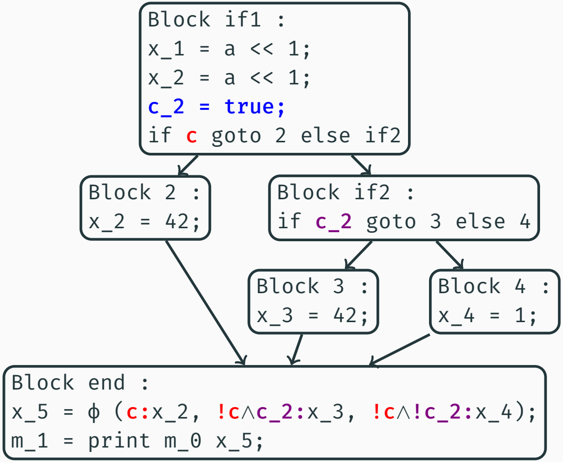


a = a ⇝ true

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

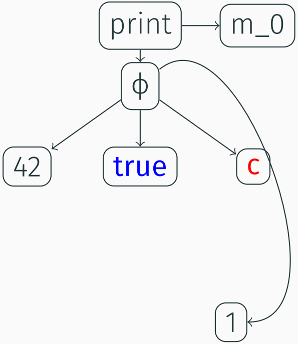


Atomic transformation : quick simplified example

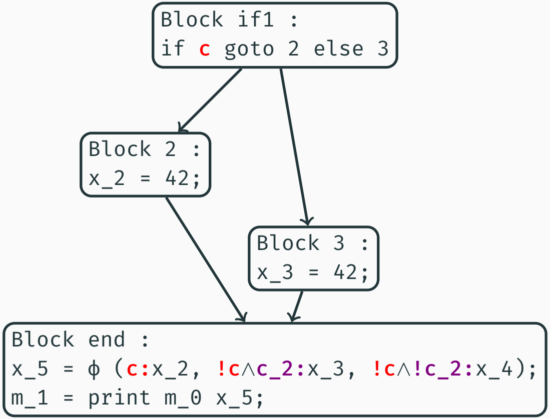


ϕ(…, false: x, …) ⇝ ϕ(…, …)

ϕ(c:42, !c∧(true):42, !c∧!(true):1)

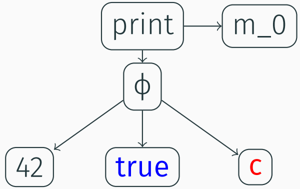


Atomic transformation : quick simplified example

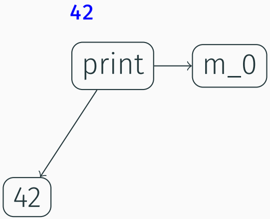


ϕ(…:x, …:x, …:x, …) ⇝ x

ϕ(c:42, !c∧(true):42)



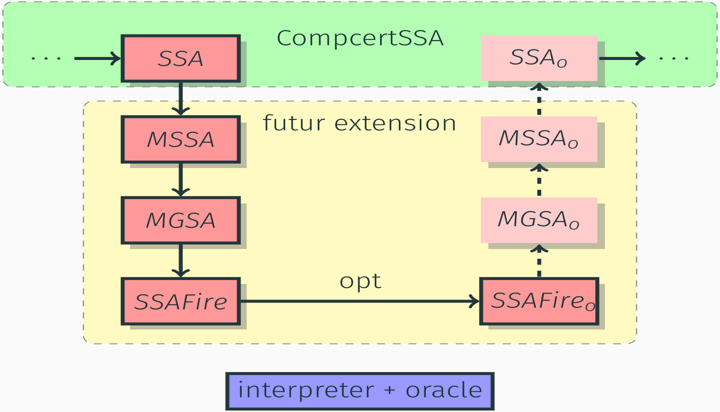
Atomic transformation : quick simplified example




```
Block end :
m_1 = print m_0 42;
```

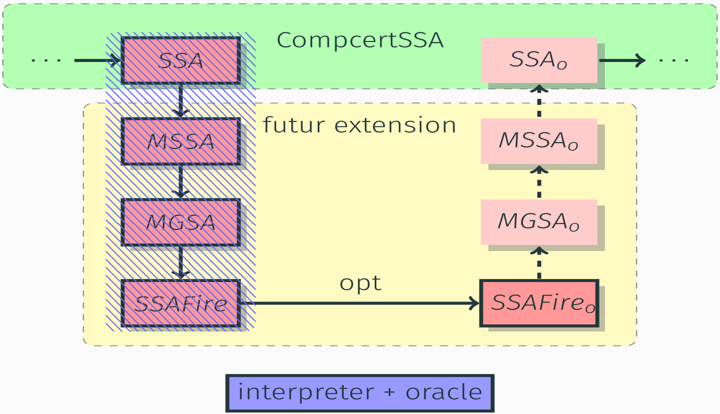
Outline

SSAFire implementation : A prototype using CompcertSSA as frontend



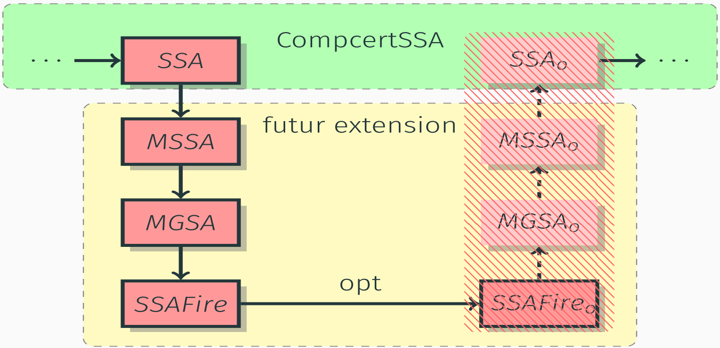
A **Caml** prototype using CompcertSSA as **C front-end**

SSAFire implementation : A prototype using CompcertSSA as frontend



Translation from SSA to SSAFIRE ... (not proven yet)

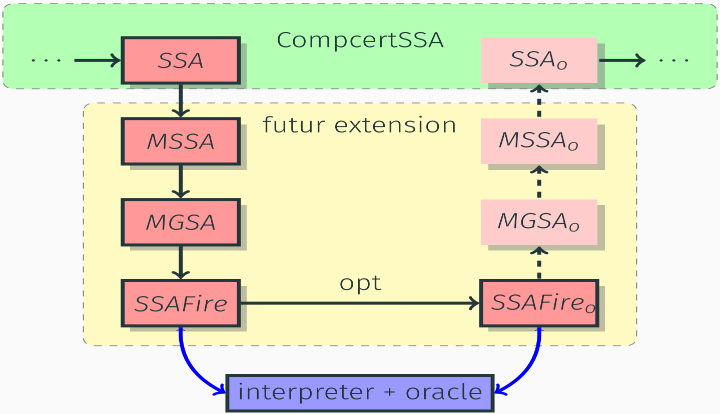
SSAFire implementation : A prototype using CompcertSSA as frontend



interpreter + oracle

SSAFIRE **deconstruction** to SSA is not done yet...

SSAFire implementation : A prototype using CompcertSSA as frontend



Can **interpret** any SSAFire programs and compare behaviours

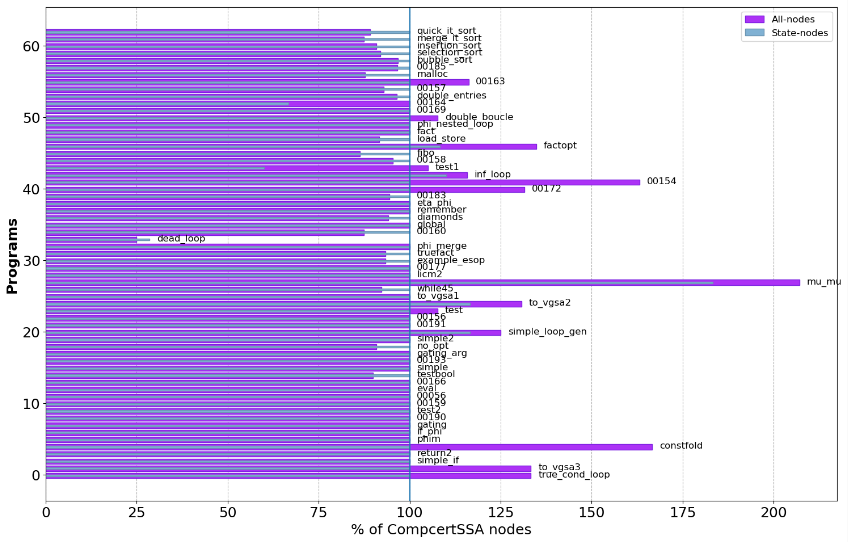
Empirical validation

Experimental validation using an **oracle** on our test-suite (62 relevant programs)

Because transformations are **atomic** we can run any possible **finite** pipeline and observe that **behaviours are preserved**

Because we cannot compare execution time without deconstruction, we compare programs sizes...

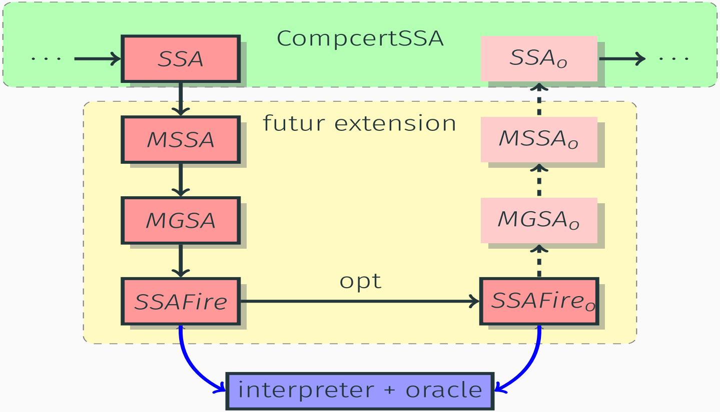
Compcert VS SSAFire (fair comparison)



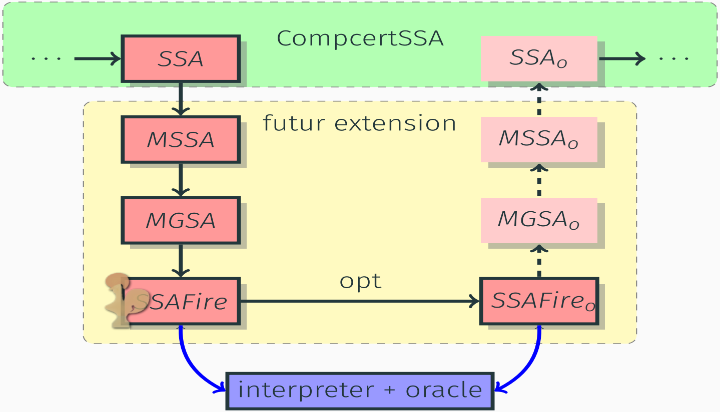
Compcert optimized translated into SSAFire Versus optimized by SSAFire transformations

Outline

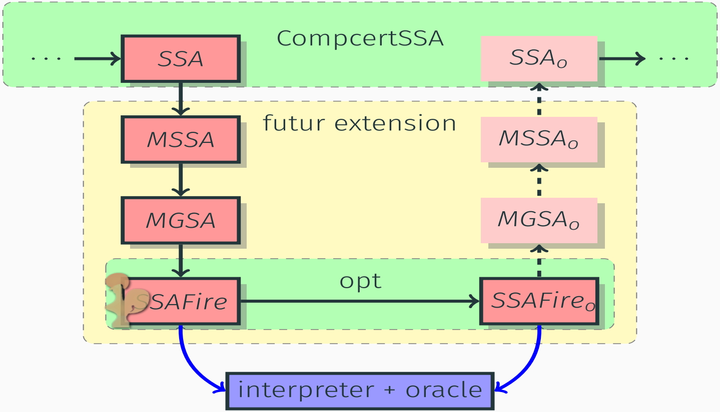
A prototype **validating experimentally** the given SSAFIRE **operational semantic**



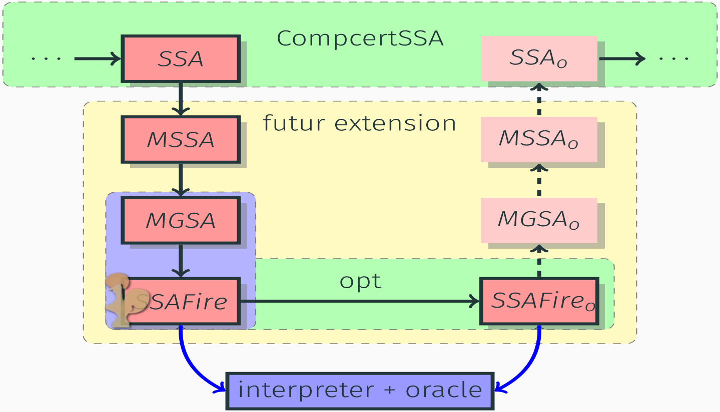
Proven **determinism** on SSAFIRE (I didn't present how and with which restrictions)



"Easy" to express then prove (work in progress) complex transformations

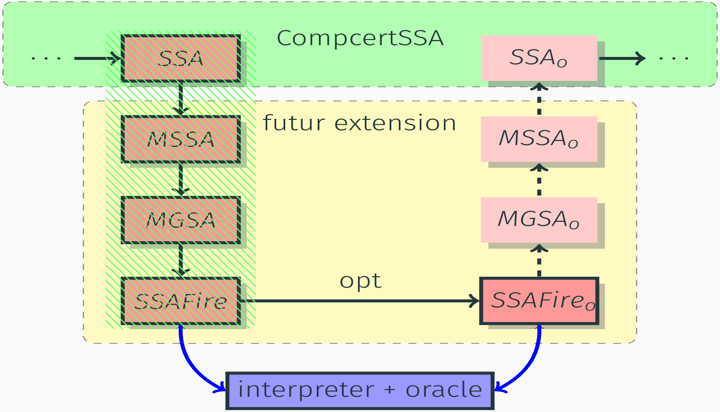


Current work : Adding memory instructions `load` and `store` to SSAFIRE

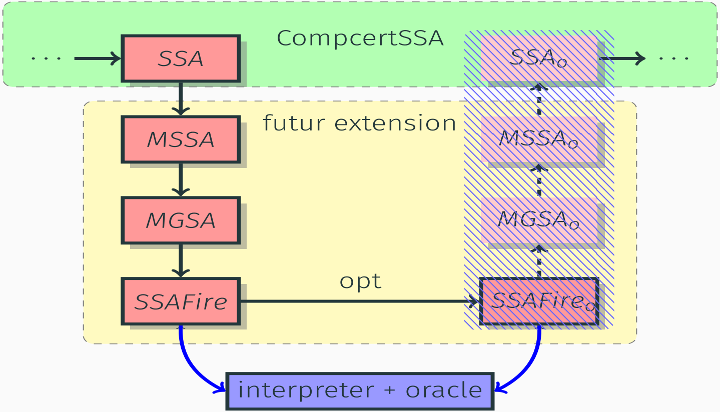


TODO

Prove semantic preservation of CompcertSSA translation to SSAFire



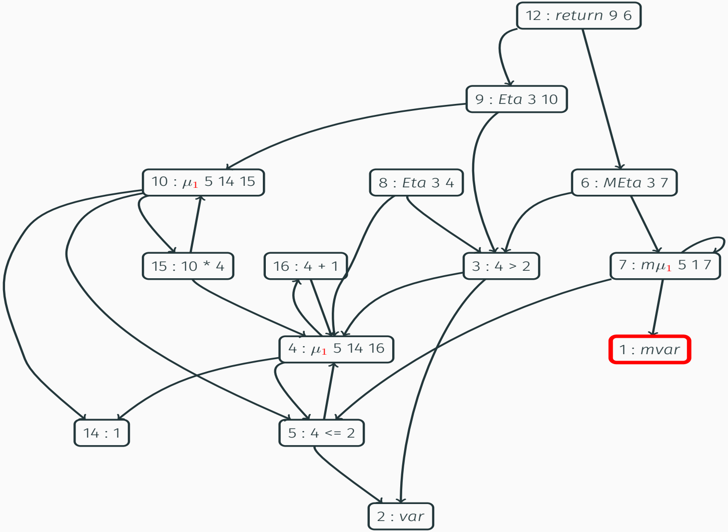
Regeneration of SSA without deoptimizing...



Questions ?

Well-formedness conditions

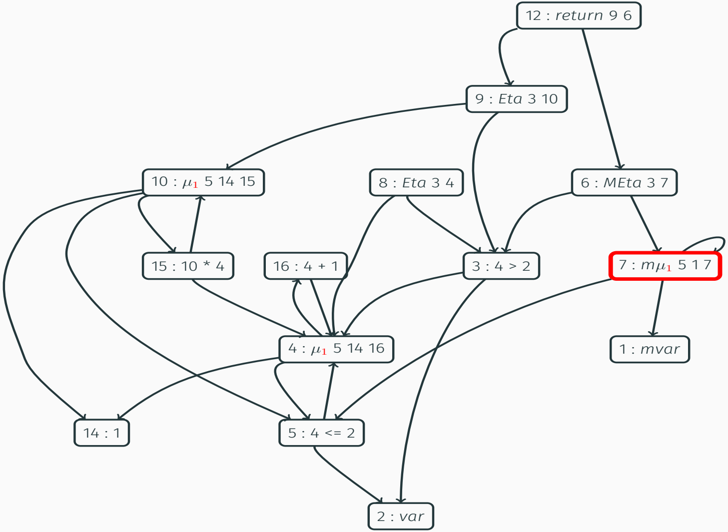
```
int main(int n){
   int i=1;
   int fact=1;
   while (i<=n) {
      fact=fact*i;
      i=i+1;
   }
   return fact;
}
```



Only one **mvar** in the graph

Well-formedness conditions

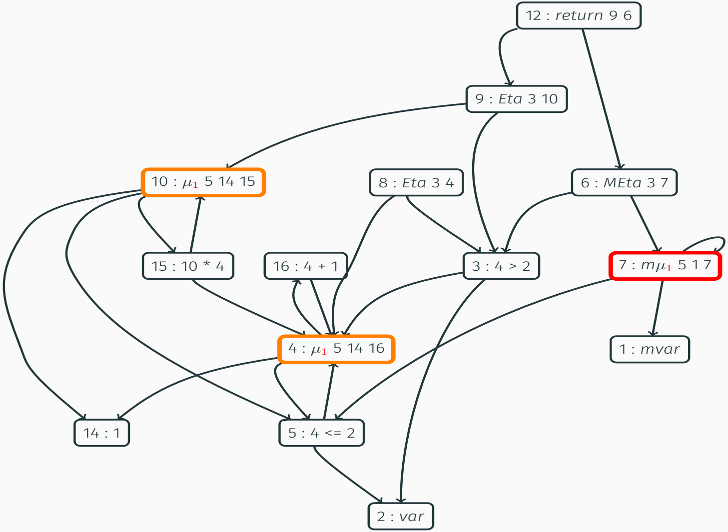
```
int main(int n){
   int i=1;
   int fact=1;
   while (i<=n) {
      fact=fact*i;
      i=i+1;
   }
   return fact;
}
```



At most one mμ per block

Well-formedness conditions

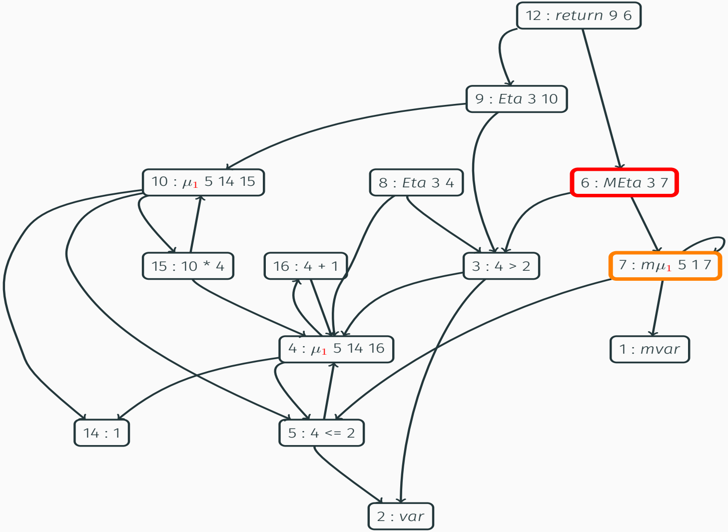
```
int main(int n){
   int i=1;
   int fact=1;
   while (i<=n) {
      fact=fact*i;
      i=i+1;
   }
   return fact;
}
```



If there is a μ a **m**μ must exist in the same block

Well-formedness conditions

```
int main(int n){
   int i=1;
   int fact=1;
   while (i<=n) {
      fact=fact*i;
      i=i+1;
   }
   return fact;
}
```



If there is a **mμ** a corresponding **meta** must exist in the graph

Well-formedness conditions

All those conditions need to be **preserved** by transformations

Value evaluation

$$\text{CST}\ \frac{g(n) = \mathsf{cst}\ k}{(m_{in}, g) \models \sigma, n \downarrow k}$$

$$\text{OP}\ \frac{g(n) = \mathsf{op}\ o\ [n_{arg_1}, \ldots, n_{arg_j}] \quad \forall i, (m_{in}, g) \models \sigma, n_{arg_i} \downarrow v_i}{(m_{in}, g) \models \sigma, n \downarrow [\![o]\!] v_1 \ldots v_k}$$

$$\text{COND}\ \frac{g(n) = \mathsf{cond}\ c\ [n_{arg_1}, \ldots, n_{arg_j}] \quad \forall i, (m_{in}, g) \models \sigma, n_{arg_i} \downarrow v_i}{(m_{in}, g) \models \sigma, n \downarrow [\![c]\!] v_1 \ldots v_k}$$

$$\text{OBS}\ \frac{g(n) = \mathsf{obs}\ n_{arg}\ m_{arg} \quad (m_{in}, g) \models \sigma, n_{arg} \downarrow v \quad (m_{in}, g) \models \sigma, m_{arg} \downarrow \mathbf{M} \varnothing [t_1, \ldots, t_j]}{(m_{in}, g) \models \sigma, n \downarrow \mathbf{M} \varnothing [t_1, \ldots, t_j].v}$$

$$\text{PHIV}\ \frac{g(n) = \mathsf{phi}\ (\gamma, n_{arg})_{i \in I} \quad \forall l, (m_{in}, g) \models \sigma, \gamma_{i(k,l)} \downarrow tt \quad (m_{in}, g) \models \sigma, n_{arg_i} \downarrow v}{(m_{in}, g) \models \sigma, n \downarrow v}$$

$$\text{ST}_{\mathsf{v}}\ \frac{n \in \mathcal{N}_{st}^{g} \qquad g(n) \in \mathcal{T}_{\mathsf{v}}}{(m_{in}, g) \models (m, \rho), n \downarrow \rho(n)}$$

$$\text{ST}_{\mathsf{m}}\ \frac{m \in \mathcal{N}_{st}^{g} \qquad g(m) \in \mathcal{T}_{\mathsf{m}}}{(m_{in}, g) \models (m, \rho), m \downarrow \mathbf{M} \varnothing []}$$

State-evaluation Relation

$$\text{VAR}\;\frac{g(n) = (\mathrm{m})\mathrm{var} \quad (m_{in}, g) \models \sigma, n{\downarrow}nv}{(m_{in}, g) \models \sigma, n{\Downarrow}nv}$$

$$\text{ETA}\;\frac{g(n) = \mathrm{eta}\ n_c\ n_{arg} \quad (m_{in}, g) \models \sigma, n_c{\downarrow}tt \quad (m_{in}, g) \models \sigma, n_{arg}{\downarrow}v}{(m_{in}, g) \models \sigma, n{\Downarrow}v}$$

$$\text{MPHI}\;\frac{g(m) = \mathrm{mphi}\ \gamma\ (\gamma_{arg}, m_{arg})_{i \in I} \quad \forall l, (m_{in}, g) \models \sigma, \gamma_{(j,l)}{\downarrow}tt \quad \forall l, (m_{in}, g) \models \sigma, \gamma_{arg_i\,(k,l)}{\downarrow}tt \quad (m_{in}, g) \models \sigma, m_{arg_i}{\downarrow}mv}{(m_{in}, g) \models \sigma, m{\Downarrow}mv}$$

$$\text{RET}\;\frac{g(m) = \mathrm{ret}\ \gamma\ n_{arg}\ m_{arg} \quad \forall l, (m_{in}, g) \models \sigma, \gamma_{(k,l)}{\downarrow}tt \quad (m_{in}, g) \models \sigma, n_{arg}{\downarrow}v \quad (m_{in}, g) \models \sigma, m_{arg}{\downarrow}\mathbf{M}\ \varnothing\ [t_1, \ldots, t_j]}{(m_{in}, g) \models \sigma, m{\Downarrow}\mathbf{M}\ v\ [t_1, \ldots, t_j]}$$

$$\text{MULOOP}\;\frac{g(n) = \mathrm{mu}_b\ \gamma\ n_i\ n_c\ n_l \quad \forall l, (m_{in}, g) \models \sigma, \gamma_{(j,l)}{\downarrow}tt \quad \forall n, g(n) = (\mathrm{m})\mathrm{mu}_b\ \_\ \_\ \_\ n_l \Rightarrow (m_{in}, g) \models \sigma, n_l{\downarrow}nv_l \quad (m_{in}, g) \models \sigma, n_c{\downarrow}ff \quad (m_{in}, g) \models \sigma, n_l{\downarrow}nv}{(m_{in}, g) \models \sigma, n{\Downarrow}nv}$$

$$\text{MUINIT}\;\frac{g(n) = \mathrm{mu}_b\ \gamma\ n_i\ n_c\ n_l \quad \forall l, (m_{in}, g) \models \sigma, \gamma_{(j,l)}{\downarrow}tt \quad \forall n, g(n) = (\mathrm{m})\mathrm{mu}_b\ \_\ n_i\ \_\ \_ \Rightarrow (m_{in}, g) \models \sigma, n_i{\downarrow}nv_i \quad \neg(\forall n, g(n) = (\mathrm{m})\mathrm{mu}_b\ \_\ \_\ \_\ n_l \Rightarrow (m_{in}, g) \models \sigma, n_l{\downarrow}nv_l) \quad (m_{in}, g) \models \sigma, n_i{\downarrow}nv}{(m_{in}, g) \models \sigma, n{\Downarrow}nv}$$

SSAFIRE transition relation

$$\text{Step}\;\frac{\rho'(n) \triangleq \begin{cases} v & \text{if } n \in \mathcal{N}^{g}_{st}, \quad g(n) \in \mathcal{T}_{\mathsf{V}}, \quad (m_{in}, g) \models (m, \rho), n \Downarrow v \\ \rho(n) & \text{otherwise} \end{cases} \qquad m' \in \max^{\preceq^m_g}(\{m_d \mid (m_{in}, g) \models (m, \rho), m_d \Downarrow \mathbf{M}\ \_\ \_\}) \qquad (m_{in}, g) \models (m, \rho), m' \Downarrow \mathbf{M}\ \varnothing\ [t_1, \ldots, t_j]}{(m_{in}, g) \models (m, \rho) \xrightarrow{[t_1, \ldots, t_j]} (m', \rho')}$$

$$\text{mvar} \preceq^m_g \ldots \textit{mstate nodes} \ldots \preceq^m_g \text{meta}$$

SSAFIRE transition relation

$$\rho'(n) \triangleq \begin{cases} v & \text{if } n \in \mathcal{N}_{st}^{g}, \quad g(n) \in \mathcal{T}_{\mathsf{V}}, \quad (m_{in}, g) \models (m, \rho), n \Downarrow v \\ \rho(n) & \text{otherwise} \end{cases}$$

$$\textsc{Step}\ \frac{m' \in \max^{\preceq_g^m}(\{m_d \mid (m_{in}, g) \models (m, \rho), m_d \Downarrow \mathbf{M}\ \_\ \_\}) \qquad (m_{in}, g) \models (m, \rho), m' \Downarrow \mathbf{M}\ \varnothing\ [t_1, \ldots, t_j]}{(m_{in}, g) \models (m, \rho) \xrightarrow{[t_1, \ldots, t_j]} (m', \rho')}$$

mvar $\preceq_g^m$ $\ldots$ *mstate nodes* $\ldots$ $\preceq_g^m$ meta

Proved deterministic

Abstract syntax

constant literals	k	$\in$	$\mathsf{Consts} = \{\mathit{ff}, \mathit{tt}, \dots, -1, 0, 1, \dots\}$
operators	o	$\in$	$\mathsf{Ops} = \{\mathsf{mov}, \mathsf{add}, \dots\}$
comparisons	c	$\in$	$\mathsf{Conds} = \{\mathsf{eq}, \mathsf{neq}, \mathsf{not}, \dots\}$

nodes id	n, n_i, n_c, n_l	$\in$	$\mathcal{N}$
block id	b	$\in$	$\mathcal{B}$
gates in DNF	$\gamma, \gamma_a, \gamma_s$	$\in$	$\wp(\wp(\mathcal{N}))$

programs	$p \in \mathcal{P}$	$=$	$\mathcal{N} \times \mathcal{G}$
code or term graphs	$g \in \mathcal{G}$	$=$	$\mathcal{N} \hookrightarrow \mathcal{T}_\mathsf{V} \cup \mathcal{T}_\mathsf{m}$

scalar terms

$$\begin{aligned}\mathcal{T}_\mathsf{V} \ni vt ::=\ & \mathsf{var} \\ |\ & \mathsf{cst}\ k \\ |\ & \mathsf{op}\ o\ [n_1, \dots, n_j] \mid \mathsf{cond}\ c\ [n_1, \dots, n_j] \\ |\ & \mathsf{eta}\ n_c\ n \\ |\ & \mathsf{phi}\ (\gamma_s, n)_i \\ |\ & \mathsf{mu}_b\ \gamma_a\ n_c\ n_i\ n_l\end{aligned}$$

memory terms

$$\begin{aligned}\mathcal{T}_\mathsf{m} \ni mt ::=\ & \mathsf{mvar} \\ |\ & \mathsf{obs}\ n\ m \\ |\ & \mathsf{meta}\ n_c\ m \\ |\ & \mathsf{ret}\ \gamma_a\ n\ m \\ |\ & \mathsf{mphi}\ \gamma_a\ (\gamma_s, m)_i \\ |\ & \mathsf{mmu}_b\ \gamma_a\ n_c\ m_i\ m_l\end{aligned}$$

Usual stuff : `constants, operators, comparisons…`

Abstract syntax

constant literals	k	$\in$	$\text{Consts} = \{ff, tt, \ldots, -1, 0, 1, \ldots\}$
operators	o	$\in$	$\text{Ops} = \{\text{mov}, \text{add}, \ldots\}$
comparisons	c	$\in$	$\text{Conds} = \{\text{eq}, \text{neq}, \text{not}, \ldots\}$

nodes id	n, n_i, n_c, n_l	$\in$	$\mathcal{N}$	programs	$p \in \mathcal{P}$	$=$	$\mathcal{N} \times \mathcal{G}$
block id	b	$\in$	$\mathcal{B}$	code or			
gates in DNF	$\gamma, \gamma_a, \gamma_s$	$\in$	$\wp(\wp(\mathcal{N}))$	term graphs	$g \in \mathcal{G}$	$=$	$\mathcal{N} \hookrightarrow \mathcal{T}_\text{v} \cup \mathcal{T}_\text{m}$

scalar terms

$$\begin{aligned} \mathcal{T}_\text{v} \ni vt ::= \; & \text{var} \\ | \; & \text{cst } k \\ | \; & \text{op } o \, [n_1, \ldots, n_j] \mid \text{cond } c \, [n_1, \ldots, n_j] \\ | \; & \text{eta } n_c \; n \\ | \; & \text{phi } (\gamma_s, n)_i \\ | \; & \text{mu}_b \; \gamma_a \; n_c \; n_i \; n_l \end{aligned}$$

memory terms

$$\begin{aligned} \mathcal{T}_\text{m} \ni mt ::= \; & \text{mvar} \\ | \; & \text{obs } n \; m \\ | \; & \text{meta } n_c \; m \\ | \; & \text{ret } \gamma_a \; n \; m \\ | \; & \text{mphi } \gamma_a \; (\gamma_s, m)_i \\ | \; & \text{mmu}_b \; \gamma_a \; n_c \; m_i \; m_l \end{aligned}$$

Differenciate “scalar terms” and “memory terms”

Abstract syntax

constant literals	k	$\in$	$\text{Consts} = \{ff, tt, \ldots, -1, 0, 1, \ldots\}$
operators	o	$\in$	$\text{Ops} = \{\text{mov}, \text{add}, \ldots\}$
comparisons	c	$\in$	$\text{Conds} = \{\text{eq}, \text{neq}, \text{not}, \ldots\}$

nodes id	n, n_i, n_c, n_l	$\in$	$\mathcal{N}$	programs	$p \in \mathcal{P}$	$=$	$\mathcal{N} \times \mathcal{G}$
block id	b	$\in$	$\mathcal{B}$	code or			
gates in DNF	$\gamma, \gamma_a, \gamma_s$	$\in$	$\wp(\wp(\mathcal{N}))$	term graphs	$g \in \mathcal{G}$	$=$	$\mathcal{N} \hookrightarrow \mathcal{T}_v \cup \mathcal{T}_m$

scalar terms

$$
\begin{aligned}
\mathcal{T}_v \ni vt ::=\ & \text{var} \\
\mid\ & \text{cst } k \\
\mid\ & \text{op } o\ [n_1, \ldots, n_j] \mid \text{cond } c\ [n_1, \ldots, n_j] \\
\mid\ & \text{eta } n_c\ n \\
\mid\ & \text{phi } (\gamma_s, n)_i \\
\mid\ & \text{mu}_b\ \gamma_a\ n_c\ n_i\ n_l
\end{aligned}
$$

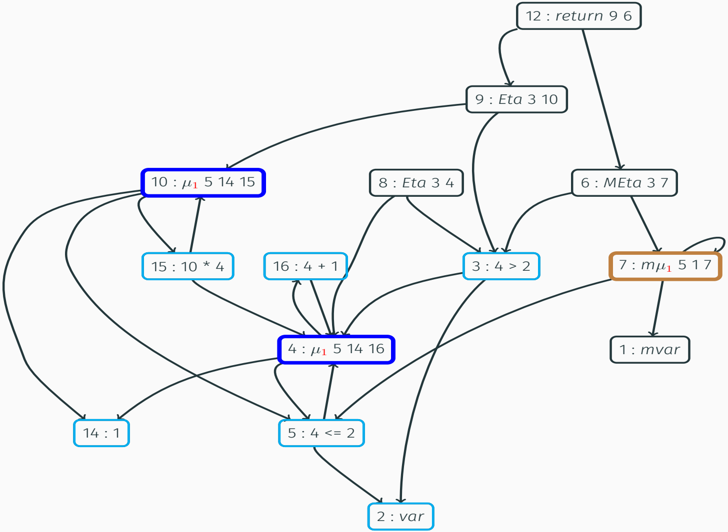
memory terms

$$
\begin{aligned}
\mathcal{T}_m \ni mt ::=\ & \text{mvar} \\
\mid\ & \text{obs } n\ m \\
\mid\ & \text{meta } n_c\ m \\
\mid\ & \text{ret } \gamma_a\ n\ m \\
\mid\ & \text{mphi } \gamma_a\ (\gamma_s, m)_i \\
\mid\ & \text{mmu}_b\ \gamma_a\ n_c\ n_i\ n_l
\end{aligned}
$$

μblock : `synchronises` μnodes of the same loop

μblock

```
int main(int n){
   int i=1;
   int fact=1;
   while (i<=n) {
      fact=fact*i;
      i=i+1;
   }
   return fact;
}
```



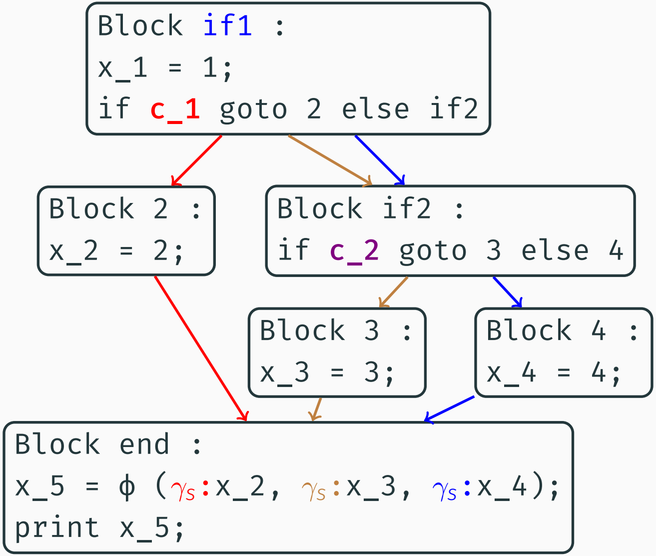
μblock : links with mμ and helps to choose between **initialization** or **iteration**

Implies some **well-formedness conditions**...

All those conditions need to be **preserved** by transformations

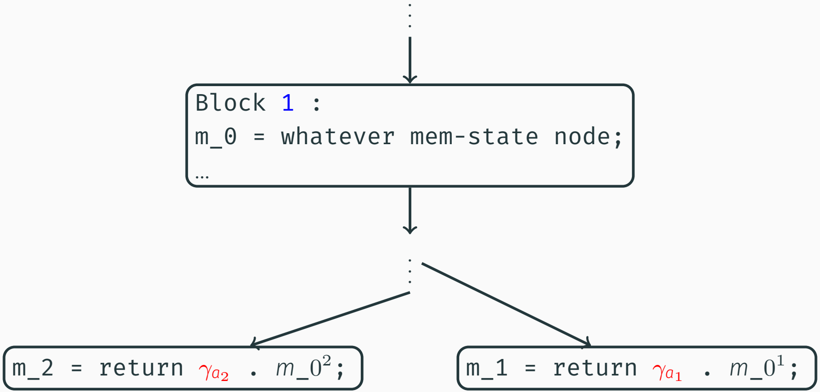
Check Syntactically Well-gated SSAFIRE

Exclusive selection gates



Check Syntactically Well-gated SSAFire

Exclusive activation gates γ_{a_1} and γ_{a_2}



Where `m_0` is the *first common root memory-state* of m_0^1 and m_0^2.

Check Syntactically Well-gated SSAFire

Syntactic well-gatedness implies **semantic exclusivity**

Semantic exclusivity and *well-formedness conditions* are **preserverd by transformations**

Necessary for **determinism**!

Atomic transformations (subset examples)

$$\text{CF1}\ \frac{g(n) = \mathsf{op}\ o\ [n_{arg_1}, \ldots, n_{arg_j}] \qquad \forall i, g(n_{arg_i}) = \mathsf{cst}\ v_i}{g \rightsquigarrow_n g[n \leftarrow \mathsf{cst}\ [\![o]\!] v_1 \ldots v_k]}$$

$$\text{CPP}\ \frac{g(n) = \mathsf{phi}\ (\gamma_s, n_{arg})_i \qquad \mathsf{open}(g, n, \gamma_{s_i})}{g \rightsquigarrow_n g[n / n_{arg_i}]}$$

$$\text{LD}\ \frac{g(n) = \mathsf{eta}\ n_c\ n_\mu \qquad g(n_\mu) = \mathsf{mu}\_\ \gamma\ n_0\ n_c\ \_ \qquad \mathsf{open}(g, n_\mu, \gamma) \qquad \mathsf{open}(g, n, \{\!\{n_c\}\!\})}{g \rightsquigarrow_n g[n_\mu / n_o][n_\mu \leftarrow \epsilon][n / n_o]}$$

$$\text{LICM1}\ \frac{g(n) = \mathsf{eta}\ n_c\ n_\mu \qquad g(n_\mu) = \mathsf{mu}\_\ \_\ n_0\ \_\ n_\mu}{g \rightsquigarrow_n g[n_\mu / n_0][n / n_0]}$$

$$\text{BE}\ \frac{g(n) = \mathsf{phi}\ (\gamma_s, n_s)_{i \in I} \qquad I_c = \{i \mid \mathsf{closed}(g, n, \gamma_{s_i}), i \in I\}}{g \rightsquigarrow_n g[n \leftarrow \mathsf{phi}\ (\gamma_s, n_s)_{i \in I \setminus I_c}]}$$

$$\text{BM}\ \frac{g(n) = \mathsf{phi}\ (\gamma_s, n_s)_{i \in I} \qquad (\gamma'_s)_i = \left(\{\gamma_{s_j} \mid j \in I, n_{s_j} = n_{s_i}\}\right)_i \qquad t_\phi = \mathsf{phi}\ (\gamma'_{s_i}, n_{s_i})}{g \rightsquigarrow_n g[n \leftarrow t_\phi]}$$

$$\text{SH}\ \frac{g(n_1) = t \qquad g(n_2) = t \qquad \varepsilon \triangleq \epsilon \text{ if } t \in \mathcal{T}_V, \mathsf{m}\epsilon \text{ otherwise}}{g \rightsquigarrow_{n_1} g[n_1 / n_2][n_2 \leftarrow \varepsilon]}$$

Local transformations

Atomic transformations (subset examples)

$$\text{CF1}\ \frac{g(n) = \text{op } o\ [n_{arg_1}, \dots, n_{arg_j}] \quad \forall i, g(n_{arg_i}) = \text{cst } v_i}{g \leadsto_n g[n \leftarrow \text{cst } [\![o]\!] v_1 \dots v_k]}$$

$$\text{CPP}\ \frac{g(n) = \text{phi } (\gamma_s, n_{arg})_i \quad \text{open}(g, n, \gamma_{s_i})}{g \leadsto_n g[n / n_{arg_i}]}$$

$$\text{LD}\ \frac{g(n) = \text{eta } n_c\ n_\mu \quad g(n_\mu) = \text{mu\_ } \gamma\ n_0\ n_c\ \_ \quad \text{open}(g, n_\mu, \gamma) \quad \text{open}(g, n, \{\!\{n_c\}\!\})}{g \leadsto_n g[n_\mu / n_o][n_\mu \leftarrow \epsilon][n / n_o]}$$

$$\text{LICM1}\ \frac{g(n) = \text{eta } n_c\ n_\mu \quad g(n_\mu) = \text{mu\_ } \_\ n_0\ \_\ n_\mu}{g \leadsto_n g[n_\mu / n_0][n / n_0]}$$

$$\text{BE}\ \frac{g(n) = \text{phi } (\gamma_s, n_s)_{i \in I} \quad I_c = \{i \mid \text{closed}(g, n, \gamma_{s_i}), i \in I\}}{g \leadsto_n g[n \leftarrow \text{phi } (\gamma_s, n_s)_{i \in I \setminus I_c}]}$$

$$\text{BM}\ \frac{g(n) = \text{phi } (\gamma_s, n_s)_{i \in I} \quad (\gamma_s')_i = \left(\{\gamma_{s_j} \mid j \in I, n_{s_j} = n_{s_i}\}\right)_i \quad t_\phi = \text{phi } (\gamma'_{s_i}, n_{s_i})}{g \leadsto_n g[n \leftarrow t_\phi]}$$

$$\text{SH}\ \frac{g(n_1) = t \quad g(n_2) = t \quad \varepsilon \triangleq \epsilon \text{ if } t \in \mathcal{T}_V, \text{m}\epsilon \text{ otherwise}}{g \leadsto_{n_1} g[n_1 / n_2][n_2 \leftarrow \varepsilon]}$$

Transformation of a node

Atomic transformations (subset examples)

$$\text{CF1}\ \frac{g(n) = \text{op } o\ [n_{arg_1}, \ldots, n_{arg_j}] \qquad \forall i, g(n_{arg_i}) = \text{cst } v_i}{g \rightsquigarrow_n g[n \leftarrow \text{cst } [\![o]\!] v_1 \ldots v_k]}$$

$$\text{CPP}\ \frac{g(n) = \text{phi } (\gamma_s, n_{arg})_i \qquad \text{open}(g, n, \gamma_{s_i})}{g \rightsquigarrow_n g[n/n_{arg_i}]}$$

$$\text{LD}\ \frac{g(n) = \text{eta } n_c\ n_\mu \qquad g(n_\mu) = \text{mu\_ } \gamma\ n_0\ n_c\ \_ \qquad \text{open}(g, n_\mu, \gamma) \quad \text{open}(g, n, \{\!\{n_c\}\!\})}{g \rightsquigarrow_n g[n_\mu/n_o][n_\mu \leftarrow \epsilon][n/n_o]}$$

$$\text{LICM1}\ \frac{g(n) = \text{eta } n_c\ n_\mu \qquad g(n_\mu) = \text{mu\_ } \_\ n_0\ \_\ n_\mu}{g \rightsquigarrow_n g[n_\mu/n_0][n/n_0]}$$

$$\text{BE}\ \frac{g(n) = \text{phi } (\gamma_s, n_s)_{i \in I} \qquad I_c = \{i \mid \text{closed}(g, n, \gamma_{s_i}), i \in I\}}{g \rightsquigarrow_n g[n \leftarrow \text{phi } (\gamma_s, n_s)_{i \in I \setminus I_c}]}$$

$$\text{BM}\ \frac{g(n) = \text{phi } (\gamma_s, n_s)_{i \in I} \qquad (\gamma_s')_i = \left(\{\gamma_{s_j} \mid j \in I, n_{s_j} = n_{s_i}\}\right)_i \qquad t_\phi = \text{phi } (\gamma_{s_i}', n_{s_i})}{g \rightsquigarrow_n g[n \leftarrow t_\phi]}$$

$$\text{SH}\ \frac{g(n_1) = t \qquad g(n_2) = t \qquad \varepsilon \triangleq \epsilon \text{ if } t \in \mathcal{T}_V, \text{m}\epsilon \text{ otherwise}}{g \rightsquigarrow_{n_1} g[n_1/n_2][n_2 \leftarrow \varepsilon]}$$

Replacement of a node by another

Atomic transformations (subset examples)

$$\text{CF1}\;\frac{g(n) = \text{op } o\ [n_{arg_1}, \ldots, n_{arg_j}] \qquad \forall i, g(n_{arg_i}) = \text{cst } v_i}{g \rightsquigarrow_n g[n \leftarrow \text{cst } [\![o]\!] v_1 \ldots v_k]}$$

$$\text{CPP}\;\frac{g(n) = \text{phi } (\gamma_s, n_{arg})_i \qquad \text{open}(g, n, \gamma_{s_i})}{g \rightsquigarrow_n g[n / n_{arg_i}]}$$

$$\text{LD}\;\frac{g(n) = \text{eta } n_c\ n_\mu \qquad g(n_\mu) = \text{mu\_ } \gamma\ n_0\ n_c\ \_ \qquad \text{open}(g, n_\mu, \gamma) \qquad \text{open}(g, n, \{\!\{n_c\}\!\})}{g \rightsquigarrow_n g[n_\mu / n_0][n_\mu \leftarrow \epsilon][n / n_0]}$$

$$\text{LICM1}\;\frac{g(n) = \text{eta } n_c\ n_\mu \qquad g(n_\mu) = \text{mu\_ } \_\ n_0\ \_\ n_\mu}{g \rightsquigarrow_n g[n_\mu / n_0][n / n_0]}$$

$$\text{BE}\;\frac{g(n) = \text{phi } (\gamma_s, n_s)_{i \in I} \qquad I_c = \{i \mid \text{closed}(g, n, \gamma_{s_i}), i \in I\}}{g \rightsquigarrow_n g[n \leftarrow \text{phi } (\gamma_s, n_s)_{i \in I \setminus I_c}]}$$

$$\text{BM}\;\frac{g(n) = \text{phi } (\gamma_s, n_s)_{i \in I} \qquad (\gamma'_s)_i = \left(\{\gamma_{s_j} \mid j \in I, n_{s_j} = n_{s_i}\}\right)_i \qquad t_\phi = \text{phi } (\gamma'_{s_i}, n_{s_i})}{g \rightsquigarrow_n g[n \leftarrow t_\phi]}$$

$$\text{SH}\;\frac{g(n_1) = t \qquad g(n_2) = t \qquad \varepsilon \triangleq \epsilon \text{ if } t \in \mathcal{T}_V, \text{ m}\epsilon \text{ otherwise}}{g \rightsquigarrow_{n_1} g[n_1 / n_2][n_2 \leftarrow \varepsilon]}$$

Means that gate is syntactically always open

Atomic transformations (subset examples)

$$\text{CF1}\ \frac{g(n) = \mathsf{op}\ o\ [n_{arg_1}, \dots, n_{arg_j}] \quad \forall i, g(n_{arg_i}) = \mathsf{cst}\ v_i}{g \rightsquigarrow_n g[n \leftarrow \mathsf{cst}\ [\![o]\!] v_1 \dots v_k]}$$

$$\text{CPP}\ \frac{g(n) = \mathsf{phi}\ (\gamma_s, n_{arg})_i \quad \mathsf{open}(g, n, \gamma_{s_i})}{g \rightsquigarrow_n g[n / n_{arg_i}]}$$

$$\text{LD}\ \frac{g(n) = \mathsf{eta}\ n_c\ n_\mu \quad g(n_\mu) = \mathsf{mu}\_\ \gamma\ n_0\ n_c\ \_ \quad \mathsf{open}(g, n_\mu, \gamma) \quad \mathsf{open}(g, n, \{\!\{n_c\}\!\})}{g \rightsquigarrow_n g[n_\mu / n_o][n_\mu \leftarrow \epsilon][n / n_o]}$$

$$\text{LICM1}\ \frac{g(n) = \mathsf{eta}\ n_c\ n_\mu \quad g(n_\mu) = \mathsf{mu}\_\ \_\ n_0\ \_\ n_\mu}{g \rightsquigarrow_n g[n_\mu / n_0][n / n_0]}$$

$$\text{BE}\ \frac{g(n) = \mathsf{phi}\ (\gamma_s, n_s)_{i \in I} \quad I_c = \{i \mid \mathsf{closed}(g, n, \gamma_{s_i}), i \in I\}}{g \rightsquigarrow_n g[n \leftarrow \mathsf{phi}\ (\gamma_s, n_s)_{i \in I \setminus I_c}]}$$

$$\text{BM}\ \frac{g(n) = \mathsf{phi}\ (\gamma_s, n_s)_{i \in I} \quad (\gamma'_s)_i = \left(\{\gamma_{s_j} \mid j \in I, n_{s_j} = n_{s_i}\}\right)_i \quad t_\phi = \mathsf{phi}\ (\gamma'_{s_i}, n_{s_i})}{g \rightsquigarrow_n g[n \leftarrow t_\phi]}$$

$$\text{SH}\ \frac{g(n_1) = t \quad g(n_2) = t \quad \varepsilon \triangleq \epsilon \text{ if } t \in \mathcal{T}_V, \mathsf{m}\epsilon \text{ otherwise}}{g \rightsquigarrow_{n_1} g[n_1 / n_2][n_2 \leftarrow \varepsilon]}$$

Constant propagation : on phi

Atomic transformations (subset examples)

$$\text{CF1}\ \frac{g(n) = \text{op } o\ [n_{arg_1}, \ldots, n_{arg_j}] \quad \forall i, g(n_{arg_i}) = \text{cst } v_i}{g \rightsquigarrow_n g[n \leftarrow \text{cst } [\![o]\!] v_1 \ldots v_k]}$$

$$\text{CPP}\ \frac{g(n) = \text{phi } (\gamma_s, n_{arg})_i \quad \text{open}(g, n, \gamma_{s_i})}{g \rightsquigarrow_n g[n / n_{arg_i}]}$$

$$\text{LD}\ \frac{g(n) = \text{eta } n_c\ n_\mu \quad g(n_\mu) = \text{mu\_ } \gamma\ n_0\ n_c\ \_ \quad \text{open}(g, n_\mu, \gamma) \quad \text{open}(g, n, \{\!\{n_c\}\!\})}{g \rightsquigarrow_n g[n_\mu / n_0][n_\mu \leftarrow \epsilon][n / n_0]}$$

$$\text{LICM1}\ \frac{g(n) = \text{eta } n_c\ n_\mu \quad g(n_\mu) = \text{mu\_ } \_\ n_0\ \_\ n_\mu}{g \rightsquigarrow_n g[n_\mu / n_0][n / n_0]}$$

$$\text{BE}\ \frac{g(n) = \text{phi } (\gamma_s, n_s)_{i \in I} \quad I_c = \{i \mid \text{closed}(g, n, \gamma_{s_i}), i \in I\}}{g \rightsquigarrow_n g[n \leftarrow \text{phi } (\gamma_s, n_s)_{i \in I \setminus I_c}]}$$

$$\text{BM}\ \frac{g(n) = \text{phi } (\gamma_s, n_s)_{i \in I} \quad (\gamma'_s)_i = \left(\{\gamma_{s_j} \mid j \in I, n_{s_j} = n_{s_i}\}\right)_i \quad t_\phi = \text{phi } (\gamma'_{s_i}, n_{s_i})}{g \rightsquigarrow_n g[n \leftarrow t_\phi]}$$

$$\text{SH}\ \frac{g(n_1) = t \quad g(n_2) = t \quad \varepsilon \triangleq \epsilon \text{ if } t \in \mathcal{T}_V, \text{m}\epsilon \text{ otherwise}}{g \rightsquigarrow_{n_1} g[n_1 / n_2][n_2 \leftarrow \varepsilon]}$$

Loop deletion : Loop exit-condition always open

Atomic transformations (subset examples)

$$\text{CF1}\ \frac{g(n) = \mathsf{op}\ o\ [n_{arg_1}, \ldots, n_{arg_j}] \qquad \forall i, g(n_{arg_i}) = \mathsf{cst}\ v_i}{g \rightsquigarrow_n g[n \leftarrow \mathsf{cst}\ [\![o]\!] v_1 \ldots v_k]}$$

$$\text{CPP}\ \frac{g(n) = \mathsf{phi}\ (\gamma_s, n_{arg})_i \qquad \mathsf{open}(g, n, \gamma_{s_i})}{g \rightsquigarrow_n g[n / n_{arg_i}]}$$

$$\text{LD}\ \frac{g(n) = \mathsf{eta}\ n_c\ n_\mu \qquad g(n_\mu) = \mathsf{mu\_}\ \gamma\ n_0\ n_c\ \_ \qquad \mathsf{open}(g, n_\mu, \gamma) \qquad \mathsf{open}(g, n, \{\!\{n_c\}\!\})}{g \rightsquigarrow_n g[n_\mu / n_0][n_\mu \leftarrow \epsilon][n / n_0]}$$

$$\text{LICM1}\ \frac{g(n) = \mathsf{eta}\ n_c\ n_\mu \qquad g(\color{red}{n_\mu}) = \mathsf{mu\_}\ \_\ n_0\ \_\ \color{red}{n_\mu}}{g \rightsquigarrow_n g[n_\mu / n_0][n / n_0]}$$

$$\text{BE}\ \frac{g(n) = \mathsf{phi}\ (\gamma_s, n_s)_{i \in I} \qquad I_c = \{i \mid \mathsf{closed}(g, n, \gamma_{s_i}), i \in I\}}{g \rightsquigarrow_n g[n \leftarrow \mathsf{phi}\ (\gamma_s, n_s)_{i \in I \setminus I_c}]}$$

$$\text{BM}\ \frac{g(n) = \mathsf{phi}\ (\gamma_s, n_s)_{i \in I} \qquad (\gamma_s')_i = \left(\{\gamma_{s_j} \mid j \in I, n_{s_j} = n_{s_i}\}\right)_i \qquad t_\phi = \mathsf{phi}\ (\gamma_{s_i}', n_{s_i})}{g \rightsquigarrow_n g[n \leftarrow t_\phi]}$$

$$\text{SH}\ \frac{g(n_1) = t \qquad g(n_2) = t \qquad \varepsilon \triangleq \epsilon \text{ if } t \in \mathcal{T}_V, \mathsf{m}\epsilon \text{ otherwise}}{g \rightsquigarrow_{n_1} g[n_1 / n_2][n_2 \leftarrow \varepsilon]}$$

Loop invariant : itself as argument

Atomic transformations (subset examples)

$$\text{CF1}\ \frac{g(n) = \text{op } o\ [n_{arg_1}, \ldots, n_{arg_j}] \qquad \forall i, g(n_{arg_i}) = \text{cst } v_i}{g \leadsto_n g[n \leftarrow \text{cst } [\![o]\!] v_1 \ldots v_k]}$$

$$\text{CPP}\ \frac{g(n) = \text{phi } (\gamma_s, n_{arg})_i \qquad \text{open}(g, n, \gamma_{s_i})}{g \leadsto_n g[n/n_{arg_i}]}$$

$$\text{LD}\ \frac{g(n) = \text{eta } n_c\ n_\mu \qquad g(n_\mu) = \text{mu\_ } \gamma\ n_0\ n_c\ \_ \qquad \text{open}(g, n_\mu, \gamma) \qquad \text{open}(g, n, \{\!\{n_c\}\!\})}{g \leadsto_n g[n_\mu/n_0][n_\mu \leftarrow \epsilon][n/n_0]}$$

$$\text{LICM1}\ \frac{g(n) = \text{eta } n_c\ n_\mu \qquad g(n_\mu) = \text{mu\_ } \_\ n_0\ \_\ n_\mu}{g \leadsto_n g[n_\mu/n_0][n/n_0]}$$

$$\text{BE}\ \frac{g(n) = \text{phi } (\gamma_s, n_s)_{i \in I} \qquad I_c = \{i \mid \text{closed}(g, n, \gamma_{s_i}), i \in I\}}{g \leadsto_n g[n \leftarrow \text{phi } (\gamma_s, n_s)_{i \in I \setminus I_c}]}$$

$$\text{BM}\ \frac{g(n) = \text{phi } (\gamma_s, n_s)_{i \in I} \qquad (\gamma_s')_i = \left(\{\gamma_{s_j} \mid j \in I, n_{s_j} = n_{s_i}\}\right)_i \qquad t_\phi = \text{phi } (\gamma'_{s_i}, n_{s_i})}{g \leadsto_n g[n \leftarrow t_\phi]}$$

$$\text{SH}\ \frac{g(n_1) = t \qquad g(n_2) = t \qquad \varepsilon \triangleq \epsilon \text{ if } t \in \mathcal{T}_V, \text{m}\epsilon \text{ otherwise}}{g \leadsto_{n_1} g[n_1/n_2][n_2 \leftarrow \varepsilon]}$$

Dead branch : selection gate always closed

Atomic transformations (subset examples)

$$\text{CF1}\ \frac{g(n) = \text{op } o\ [n_{arg_1}, \ldots, n_{arg_j}] \qquad \forall i, g(n_{arg_i}) = \text{cst } v_i}{g \rightsquigarrow_n g[n \leftarrow \text{cst } [\![o]\!] v_1 \ldots v_k]}$$

$$\text{CPP}\ \frac{g(n) = \text{phi } (\gamma_s, n_{arg})_i \qquad \text{open}(g, n, \gamma_{s_i})}{g \rightsquigarrow_n g[n/n_{arg_i}]}$$

$$\text{LD}\ \frac{g(n) = \text{eta } n_c\ n_\mu \qquad g(n_\mu) = \text{mu\_ } \gamma\ n_0\ n_c\ \_ \qquad \text{open}(g, n_\mu, \gamma) \qquad \text{open}(g, n, \{\!\{n_c\}\!\})}{g \rightsquigarrow_n g[n_\mu/n_0][n_\mu \leftarrow \epsilon][n/n_0]}$$

$$\text{LICM1}\ \frac{g(n) = \text{eta } n_c\ n_\mu \qquad g(n_\mu) = \text{mu\_ } \_\ n_0\ \_\ n_\mu}{g \rightsquigarrow_n g[n_\mu/n_0][n/n_0]}$$

$$\text{BE}\ \frac{g(n) = \text{phi } (\gamma_s, n_s)_{i \in I} \qquad I_c = \{i \mid \text{closed}(g, n, \gamma_{s_i}), i \in I\}}{g \rightsquigarrow_n g[n \leftarrow \text{phi } (\gamma_s, n_s)_{i \in I \setminus I_c}]}$$

$$\text{BM}\ \frac{g(n) = \text{phi } (\gamma_s, n_s)_{i \in I} \qquad (\gamma'_s)_i = \left(\{\gamma_{s_j} \mid j \in I, n_{s_j} = n_{s_i}\}\right)_i \qquad t_\phi = \text{phi } (\gamma'_{s_i}, n_{s_i})}{g \rightsquigarrow_n g[n \leftarrow t_\phi]}$$

$$\text{SH}\ \frac{g(n_1) = t \qquad g(n_2) = t \qquad \varepsilon \triangleq \epsilon \text{ if } t \in \mathcal{T}_V, \text{ m}\epsilon \text{ otherwise}}{g \rightsquigarrow_{n_1} g[n_1/n_2][n_2 \leftarrow \varepsilon]}$$

Merge branch : two branches returning same value

Atomic transformations (subset examples)

$$\text{CF1}\ \frac{g(n) = \text{op } o\ [n_{arg_1}, \ldots, n_{arg_j}] \quad \forall i, g(n_{arg_i}) = \text{cst } v_i}{g \rightsquigarrow_n g[n \leftarrow \text{cst } [\![o]\!] v_1 \ldots v_k]}$$

$$\text{CPP}\ \frac{g(n) = \text{phi } (\gamma_s, n_{arg})_i \quad \text{open}(g, n, \gamma_{s_i})}{g \rightsquigarrow_n g[n / n_{arg_i}]}$$

$$\text{LD}\ \frac{g(n) = \text{eta } n_c\ n_\mu \quad g(n_\mu) = \text{mu\_ } \gamma\ n_0\ n_c\ \_ \quad \text{open}(g, n_\mu, \gamma) \quad \text{open}(g, n, \{\!\{n_c\}\!\})}{g \rightsquigarrow_n g[n_\mu / n_0][n_\mu \leftarrow \epsilon][n / n_0]}$$

$$\text{LICM1}\ \frac{g(n) = \text{eta } n_c\ n_\mu \quad g(n_\mu) = \text{mu\_ } \_\ n_0\ \_\ n_\mu}{g \rightsquigarrow_n g[n_\mu / n_0][n / n_0]}$$

$$\text{BE}\ \frac{g(n) = \text{phi } (\gamma_s, n_s)_{i \in I} \quad I_c = \{i \mid \text{closed}(g, n, \gamma_{s_i}), i \in I\}}{g \rightsquigarrow_n g[n \leftarrow \text{phi } (\gamma_s, n_s)_{i \in I \setminus I_c}]}$$

$$\text{BM}\ \frac{g(n) = \text{phi } (\gamma_s, n_s)_{i \in I} \quad (\gamma'_s)_i = \left(\{\gamma_{s_j} \mid j \in I, n_{s_j} = n_{s_i}\}\right)_i \quad t_\phi = \text{phi } (\gamma'_{s_i}, n_{s_i})}{g \rightsquigarrow_n g[n \leftarrow t_\phi]}$$

$$\text{SH}\ \frac{g(n_1) = t \quad g(n_2) = t \quad \varepsilon \triangleq \epsilon \text{ if } t \in \mathcal{T}_V, \text{ m}\epsilon \text{ otherwise}}{g \rightsquigarrow_{n_1} g[n_1 / n_2][n_2 \leftarrow \varepsilon]}$$

Sharing : two nodes syntactically equal