Mechanised Semantics for Gated Static Single Assignment

Yann Herklotz[1] Delphine Demange[2] Sandrine Blazy[2]

CPP'23, 16th January

[1] Imperial College London

[2] IRISA, Inria, CNRS, Univ de Rennes

Overview

1. Refresher on SSA
2. Translation from SSA to GSA
3. Proof of SSA to GSA Translation
4. Summary

Refresher on SSA

Refresher on SSA

Introduced in late 80’s [Alpern et al., 1988]

Now widely adopted in compiler community
GCC, LLVM, Java HotSpot JIT…

SSA: Variables with *unique* definition point

Refresher on SSA

Introduced in late 80’s [Alpern et al., 1988]

Now widely adopted in compiler community

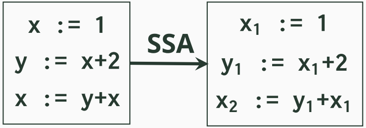
GCC, LLVM, Java HotSpot JIT…

SSA: Variables with *unique* definition point

Straight-line code

Definitions: fresh variable, version number

Uses: rename variable, pick right version



Refresher on SSA

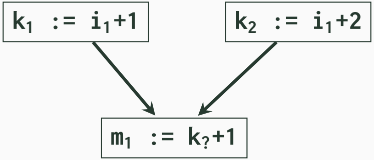
Introduced in late 80’s [Alpern et al., 1988]

Now widely adopted in compiler community
GCC, LLVM, Java HotSpot JIT…

SSA: Variables with *unique* definition point

Straight-line code
Definitions: fresh variable, version number
Uses: rename variable, pick right version

Control-flow join points
Which version should be used? Depends!



Refresher on SSA

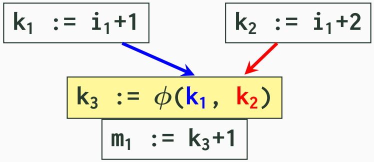
Introduced in late 80’s [Alpern et al., 1988]

Now widely adopted in compiler community
GCC, LLVM, Java HotSpot JIT...

SSA: Variables with *unique* definition point

Straight-line code
Definitions: fresh variable, version number
Uses: rename variable, pick right version

Control-flow join points
Which version should be used? Depends!
Dedicated instruction: $k_3 := \phi(k_1, k_2)$
Based on control-flow, select right argument



Benefits and Shortcomings of SSA

SSA strengths

CFG-based representation: simple operational semantics

ϕ-instructions already capture def/use dependencies

Benefits and Shortcomings of SSA

SSA strengths

CFG-based representation: simple operational semantics

ϕ-instructions already capture def/use dependencies

SSA weaknesses

Semantics of ϕ-instructions depends on control-flow

Non-local semantics of ϕ-instructions

Benefits and Shortcomings of SSA

SSA strengths

CFG-based representation: simple operational semantics

ϕ-instructions already capture def/use dependencies

SSA weaknesses

Semantics of ϕ-instructions depends on control-flow

Non-local semantics of ϕ-instructions

Gated SSA: Use gates to turn control into data-dependencies

Local execution of gates replacing ϕ-instructions

Gated SSA: New Instructions

Gated SSA: extends ϕ-instructions with gates

Simple join points:

$$r_d \leftarrow \gamma(\overrightarrow{(p_i, r_i)})$$

Predicate p_i discriminate arguments, local choice

Gated SSA: New Instructions

Gated SSA: extends ϕ-instructions with gates

Simple join points: $r_d \leftarrow \gamma(\overrightarrow{(p_i, r_i)})$

Predicate p_i discriminate arguments, local choice

Loop-header join point: $r_d \leftarrow \mu(r_0, r_i)$

Idea: no adequate predicate for iterations

Introduce a special node, with built-in looping semantics

Gated SSA: New Instructions

Gated SSA: extends ϕ-instructions with gates

Simple join points: $r_d \leftarrow \gamma(\overrightarrow{(p_i, r_i)})$

Predicate p_i discriminate arguments, local choice

Loop-header join point: $r_d \leftarrow \mu(r_0, r_i)$

Idea: no adequate predicate for iterations

Introduce a special node, with built-in looping semantics

Loop exit point: $r_d \leftarrow \eta(p, r_s)$

Idea: decouple loop-carried variable from end-of-loop usage

Gate p signals when r_s has reached a stable value

Gated SSA: State of affairs

Numerous variants of Gated SSA

Each come with own notion of dependencies

No formal semantics, partial and informal prose

Gated SSA: State of affairs

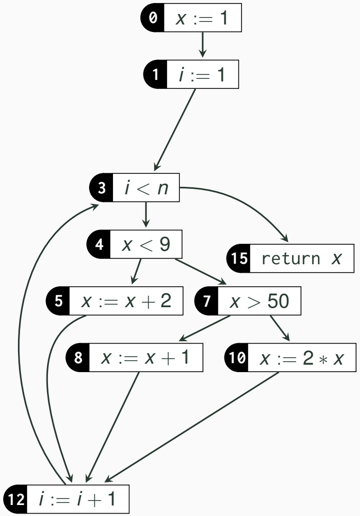
Numerous variants of Gated SSA
Each come with own notion of dependencies
No formal semantics, partial and informal prose

Our Contributions

- Describe a specification and control-flow semantics for Gated SSA.
- Focus on the control-flow independent semantics of gates.
- Describe implementation and proof in CompCertSSA.

Translation from SSA to GSA

Gated SSA (GSA): Example Generation

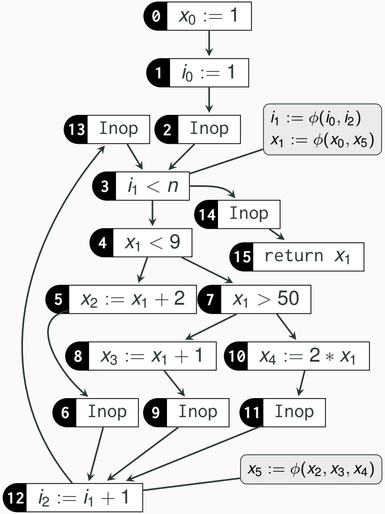


RTL

Control-flow graph for the following program:

```
int f(int n) {
  int x = 1;
  for (int i = 1; i < n; i++)
    if (x < 9) x = x + 2;
    else if (x > 50) x = x + 1;
    else x = 2 * x;
  return x;
}
```

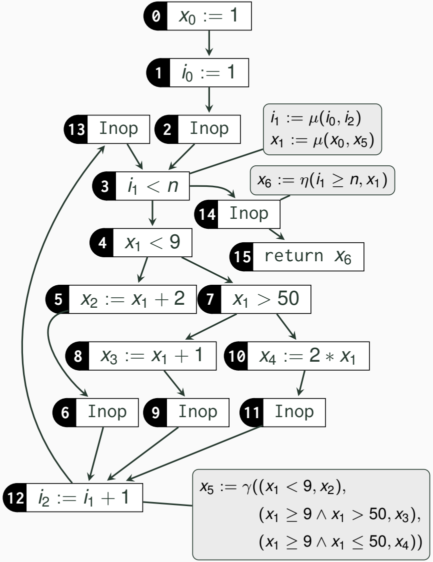
Gated SSA (GSA): Example Generation



SSA

- Additional nop instructions are inserted to normalise control-flow graph.
- Variable assignments are made unique.
- Existing SSA Generation inserts ϕ-instructions.

Gated SSA (GSA): Example Generation



GSA

- Replace ϕ-instructions by μ- and γ-instructions, then insert η-instructions.
- Predicates use normal syntactic elements.

Translating from SSA to GSA

Single-source path expression problem

"Find, for each vertex v, a regular expression $P(s, v)$ which represents the set of all paths in G from s to v." — [Tarjan, 1981]

Translating from SSA to GSA

Single-source path expression problem

"Find, for each vertex v, a regular expression $P(s, v)$ which represents the set of all paths in G from s to v." — [Tarjan, 1981]

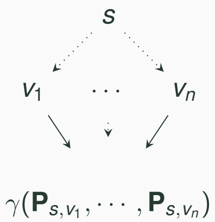
- We translate path expressions to predicates.
- Path expression $P(s, v)$ become predicate $\mathbf{P}_{s,v}$.

Translating from SSA to GSA

Single-source path expression problem

"Find, for each vertex v, a regular expression $P(s, v)$ which represents the set of all paths in G from s to v." — [Tarjan, 1981]

For every future γ node, get a path-expression from the dominator s to each of its predecessors v_1, v_2, ..., v_n.



Proof of SSA to GSA Translation

How do We Verify These Opaque Predicates?

- Path expression algorithm is not formalised.
- Validate predicates in gates after-the-fact.

How do We Verify These Opaque Predicates?

- Path expression algorithm is not formalised.
- Validate predicates in gates after-the-fact.

Main issues

- Reasoning about predicates is global and dynamic.
- Reason about executed and non-executed paths.

How do We Verify These Opaque Predicates?

- Path expression algorithm is not formalised.
- Validate predicates in gates after-the-fact.

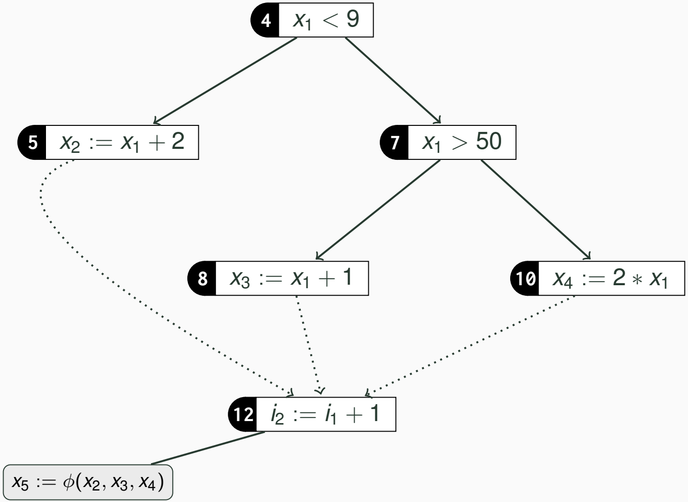
Main issues

- Reasoning about predicates is global and dynamic.
- Reason about executed and non-executed paths.

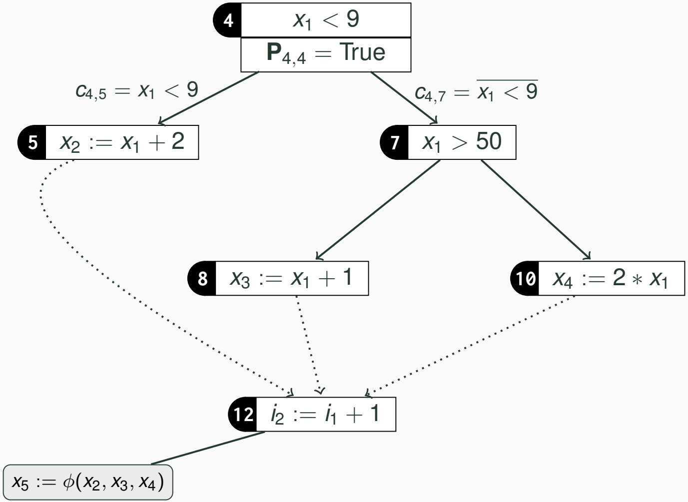
Key intuition

- Build local correctness rules about predicates for every node.
- Use them to build a proof about the evaluation of predicates.
- *Key properties*: coherence $\wedge$ mutual independence $\implies$ validity.

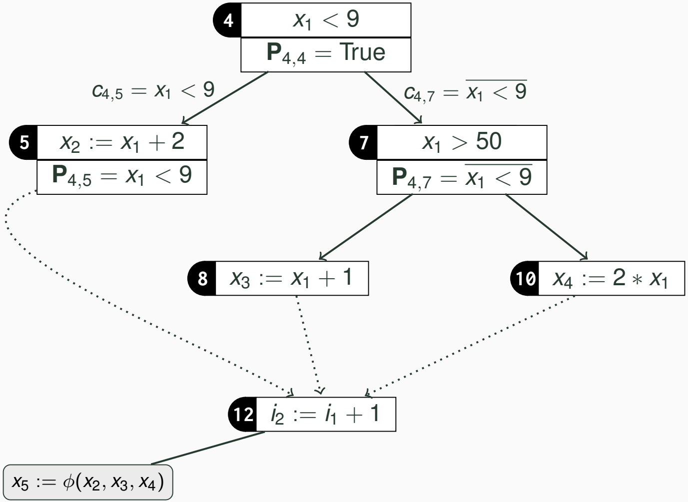
Predicate Generation: Example



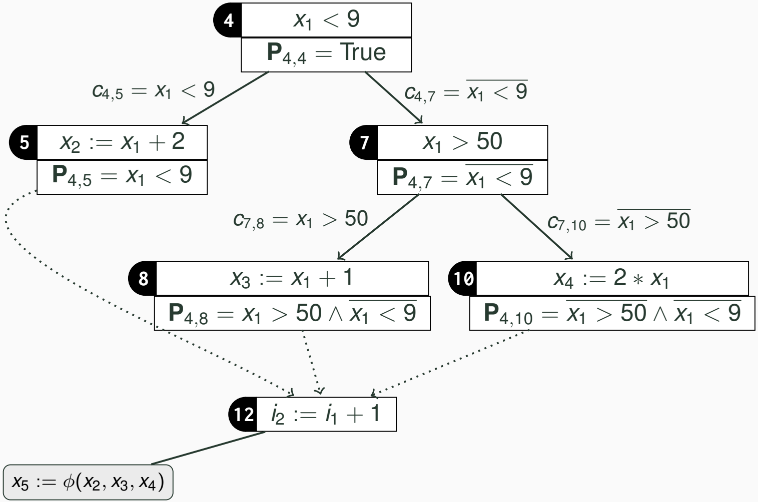
Predicate Generation: Example



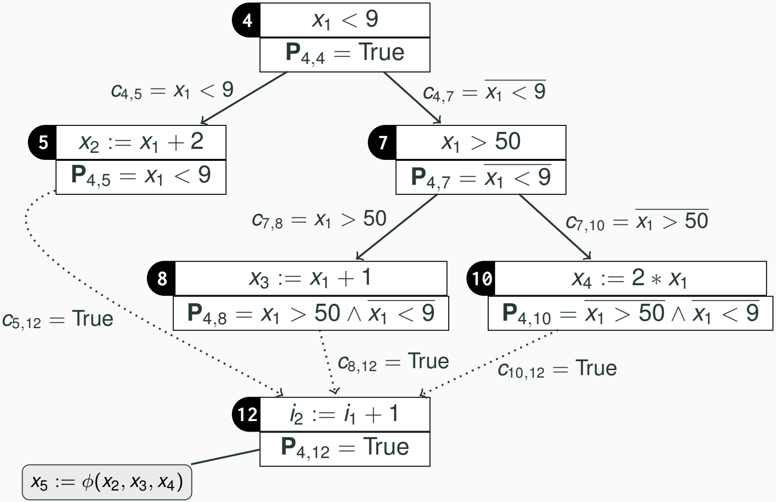
Predicate Generation: Example



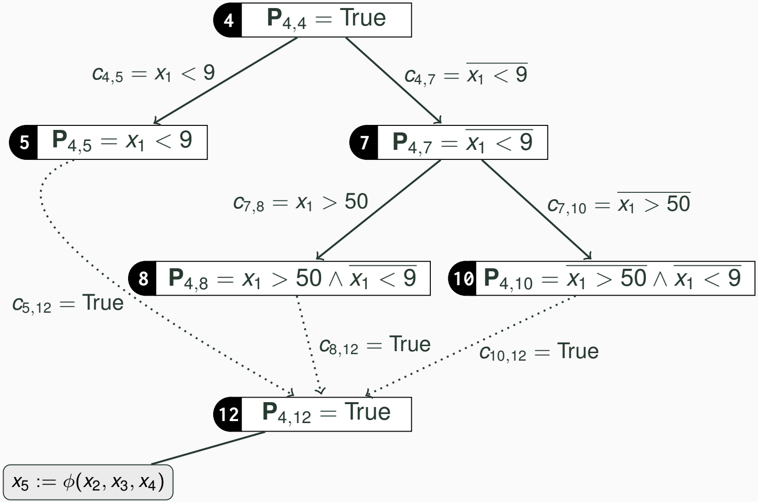
Predicate Generation: Example



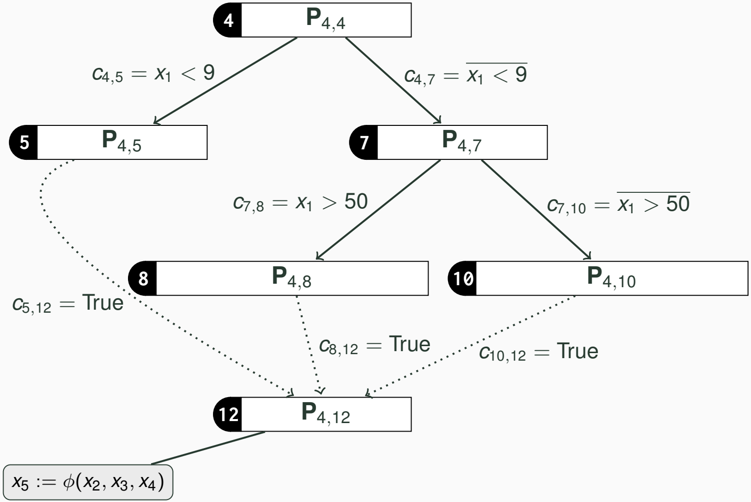
Predicate Generation: Example



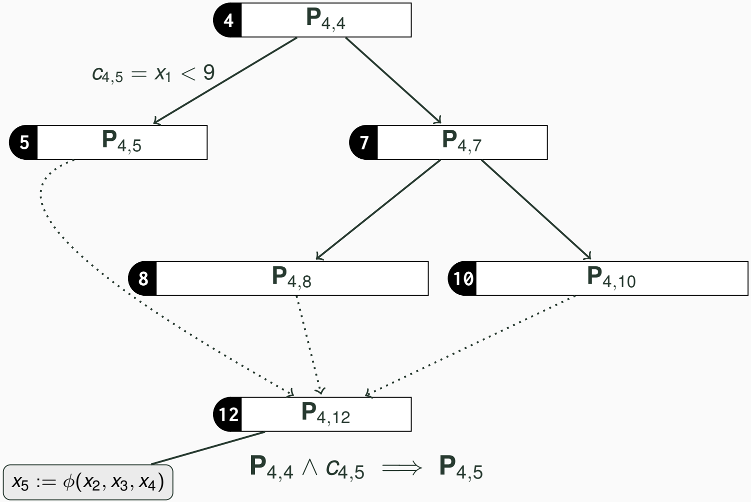
Coherence Property: Example



Coherence Property: Example

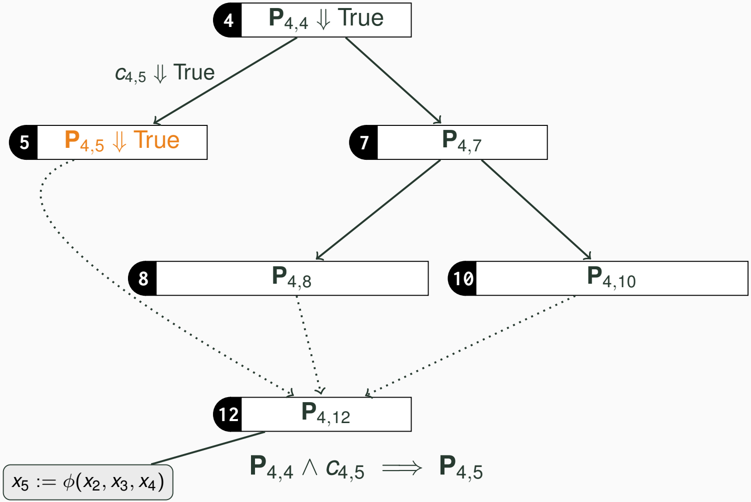


Coherence Property: Example



$$P_{4,4} \wedge c_{4,5} \implies P_{4,5}$$

Validity Property: Example



$$\mathbf{P}_{4,4} \wedge c_{4,5} \implies \mathbf{P}_{4,5}$$

Invariant: $\mathbf{P}_{4,i} \Downarrow \mathit{True}$

Using an SMT Solver to Check Properties

Want to prove the following correct

$$\mathbf{P}_{4,4} \wedge c_{4,5} \implies \mathbf{P}_{4,5}$$

Using an SMT Solver to Check Properties

Want to prove the following correct

$$\mathbf{P}_{4,4} \wedge c_{4,5} \implies \mathbf{P}_{4,5}$$

Use Three-Valued Logic and SMT Solver show unsat

$$\neg(\mathbf{P}_{4,4} \wedge c_{4,5} \rightarrow_{Ł} \mathbf{P}_{4,5})$$

Using Three-Valued Łukasiewicz Logic:

Syntactic elements in predicates might not be evaluable.

Using an SMT Solver to Check Properties

Want to prove the following correct

$$\mathbf{P}_{4,4} \land c_{4,5} \implies \mathbf{P}_{4,5}$$

Use Three-Valued Logic and SMT Solver show unsat

$$\neg(\mathbf{P}_{4,4} \land c_{4,5} \rightarrow_{Ł} \mathbf{P}_{4,5})$$

Using Three-Valued Łukasiewicz Logic:

Syntactic elements in predicates might not be evaluable.

Generate low-level formula for SMTCoq and veriT to obtain validated SMT Check.

Summary

Summary and Future Work

Implementation within CompCertSSA

- Gated SSA: syntax and semantics
- Correct generation of Gated SSA
- Prove global validity of predicates using coherence and mutual independence.

Summary and Future Work

Implementation within CompCertSSA

- Gated SSA: syntax and semantics
- Correct generation of Gated SSA
- Prove global validity of predicates using coherence and mutual independence.

Limitations

- Conditions dependent on memory not supported in predicates.
- GSA predicate validation quite slow with validated SMT solver.

Summary and Future Work

Implementation within CompCertSSA

- Gated SSA: syntax and semantics
- Correct generation of Gated SSA
- Prove global validity of predicates using coherence and mutual independence.

Limitations

- Conditions dependent on memory not supported in predicates.
- GSA predicate validation quite slow with validated SMT solver.

Future work: Pure data-flow semantics, proof of Tarjan's SSPE, well-formed GSA.

Thank You, Any Questions?

Paper

Artefact

Gated SSA: New Instructions

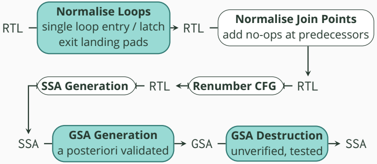
Gated SSA: extends ϕ-instructions with gates

Simple join points: $r_d \leftarrow \gamma(\overrightarrow{(p_i, r_i)})$
Predicate p_i discriminate arguments, local choice

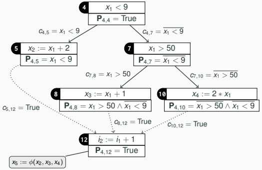
Loop-header join point: $r_d \leftarrow \mu(r_0, r_i)$
Idea: no adequate predicate for iterations
Introduce a special node, with built-in looping semantics

Loop exit point: $r_d \leftarrow \eta(p, r_s)$
Idea: decouple loop-carried variable from end-of-loop usage
Gate p signals when r_s has reached a stable value

5



Predicate Generation: Example



10

Using an SMT Solver to Check Properties

Want to prove the following correct

$$\mathbf{P}_{4,4} \wedge c_{4,5} \implies \mathbf{P}_{4,5}$$

Use Ternary Logic and SMT Solver show unsat

$$\neg(\mathbf{P}_{4,4} \wedge c_{4,5} \rightarrow_{Ł} \mathbf{P}_{4,5})$$

Using Ternary Łukasiewicz Logic:

Syntactic elements in predicates might not be evaluable.

Generate low-level formula for SMTCoq and veriT to obtain validated SMT Check.

13

Semantics of Gated SSA

Eta

$$\frac{i = r_d \leftarrow \eta(q, r) \qquad rs \models_p q \Downarrow 1 \qquad b_\eta \vdash rs \overset{\mathcal{E}}{\rightsquigarrow} rs'}{\lfloor i :: b_\eta \rfloor \vdash rs \overset{\mathcal{E}}{\rightsquigarrow} rs'[r_d \mapsto rs(r)]}$$

Merge$_\gamma$

$$\frac{i = r_d \leftarrow \gamma(\overrightarrow{(q, r)}) \qquad rs \models_p q_n \Downarrow 1 \qquad b_\mathcal{M}, k \vdash rs \overset{\mathcal{M}}{\rightsquigarrow} rs'}{i :: b_\mathcal{M}, k \vdash rs \overset{\mathcal{M}}{\rightsquigarrow} rs'[r_d \mapsto rs(r_n)]}$$

Merge$_\mu$

$$\frac{i = r_d \leftarrow \mu(r_0, r_1) \qquad k \in \{0, 1\} \qquad b_\mathcal{M}, k \vdash rs \overset{\mathcal{M}}{\rightsquigarrow} rs'}{i :: b_\mathcal{M}, k \vdash rs \overset{\mathcal{M}}{\rightsquigarrow} rs'[r_d \mapsto rs(r_k)]}$$

NJoin

$$\frac{f.\mathcal{I}(l) = \lfloor \texttt{Inop}(l') \rfloor \qquad f \not\curlyvee l' \qquad f.\mathcal{E}(l) \vdash rs \overset{\mathcal{E}}{\rightsquigarrow} rs'}{\vdash \mathcal{S}(f, l, rs) \rightarrow \mathcal{S}(f, l', rs')}$$

Join

$$\frac{f.\mathcal{I}(l) = \lfloor \texttt{Inop}(l') \rfloor \qquad f \curlyvee l' \qquad f.\mathcal{M}(l') = \lfloor b_\mathcal{M} \rfloor \qquad f.\mathcal{E}(l) \vdash rs \overset{\mathcal{E}}{\rightsquigarrow} rs' \qquad \text{preds}(l')_k = l \qquad b_\mathcal{M}, k \vdash rs' \overset{\mathcal{M}}{\rightsquigarrow} rs''}{\vdash \mathcal{S}(f, l, rs) \rightarrow \mathcal{S}(f, l', rs'')}$$

References i

Alpern, B., Wegman, M. N., and Zadeck, F. K. (1988).
Detecting equality of variables in programs.
In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, page 1–11, New York, NY, USA. Association for Computing Machinery.

Arenaz, M., Amoedo, P., and Touriño, J. (2008).
Efficiently building the gated single assignment form in codes with pointers in modern optimizing compilers.
In Luque, E., Margalef, T., and Benítez, D., editors, *Euro-Par 2008 – Parallel Processing*, pages 360–369, Berlin, Heidelberg. Springer Berlin Heidelberg.

Derrien, S., Marty, T., Rokicki, S., and Yuki, T. (2020).
Toward speculative loop pipelining for high-level synthesis.
IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 39(11):4229–4239.

References ii

Havlak, P. (1994).
Construction of thinned gated single-assignment form.
In Banerjee, U., Gelernter, D., Nicolau, A., and Padua, D., editors, *Languages and Compilers for Parallel Computing*, pages 477–499, Berlin, Heidelberg. Springer Berlin Heidelberg.

Ottenstein, K. J., Ballance, R. A., and MacCabe, A. B. (1990).
The program dependence web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages.
In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, page 257–271, New York, NY, USA. Association for Computing Machinery.

Sampaio, D., Martins, R., Collange, C., and Pereira, F. M. Q. (2012).
Divergence analysis with affine constraints.
In *2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing*, pages 67–74.

Tarjan, R. E. (1981).
Fast algorithms for solving path problems.
J. ACM, 28(3):594–614.

References iii

Tristan, J.-B., Govereau, P., and Morrisett, G. (2011).
Evaluating value-graph translation validation for LLVM.
In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 295–305, New York, NY, USA. Association for Computing Machinery.

Tu, P. and Padua, D. (1995).
Gated ssa-based demand-driven symbolic analysis for parallelizing compilers.
In *Proceedings of the 9th International Conference on Supercomputing*, ICS '95, page 414–423, New York, NY, USA. Association for Computing Machinery.