Gated SSA

Mechanised Semantics for Gated Static Single Assignment

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

CAS Seminar, 31 October 2022

[1] IRISA & Inria de l'Université de Rennes

[2] Imperial College London

Overview

1. Refresher on SSA
2. Proof of SSA to GSA Translation
3. Summary and On-going Work

Refresher on SSA

About GSA and 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

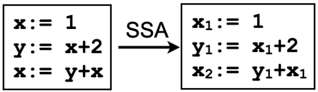
About GSA and SSA

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



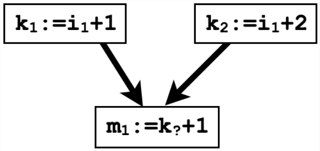
About GSA and SSA

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!



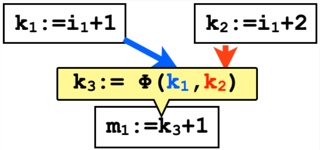
About GSA and SSA

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 $x_3 \leftarrow \phi(x_1, x_2)$
Based on control-flow, select right argument



From SSA to Gated SSA

SSA strengths

CFG-based representation: simple operational semantics

ϕ-functions already capture def/use dependencies

From SSA to Gated SSA

SSA strengths

CFG-based representation: simple operational semantics

ϕ-functions already capture def/use dependencies

SSA weaknesses

Semantics of ϕ-functions depends on control-flow

Non-local semantics of ϕ-functions: $\mathcal{S}(f, l, rs)$ not enough

Some dependencies are still implicit

From SSA to Gated SSA

SSA strengths

CFG-based representation: simple operational semantics

ϕ-functions already capture def/use dependencies

SSA weaknesses

Semantics of ϕ-functions depends on control-flow

Non-local semantics of ϕ-functions: $\mathcal{S}(f, l, rs)$ not enough

Some dependencies are still implicit

Gated SSA: gates turn control-dep. into data-dep.

Building block of Program Dependence Web [Ottenstein et al., 1990]

Ignore some dependencies [Havlak, 1994]

Symbolic analysis for parallelizing compiler [Tu and Padua, 1995]

Gated SSA: New Instructions

Gated SSA: extends ϕ-instructions with gates

Simple join points:

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

Gates p_i discriminate arguments, local choice

Pure data-dependency

Gated SSA: New Instructions

Gated SSA: extends ϕ-instructions with gates

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

Gates p_i discriminate arguments, local choice

Pure data-dependency

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

Idea: no adequate gate for iterations

Introduce a special node, with built-in looping semantics

Analyze loop-carried dependencies

Gated SSA: New Instructions

Gated SSA: extends ϕ-instructions with gates

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

Gates p_i discriminate arguments, local choice

Pure data-dependency

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

Idea: no adequate gate for iterations

Introduce a special node, with built-in looping semantics

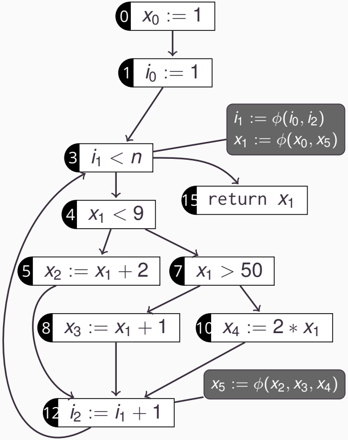
Analyze loop-carried dependencies

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

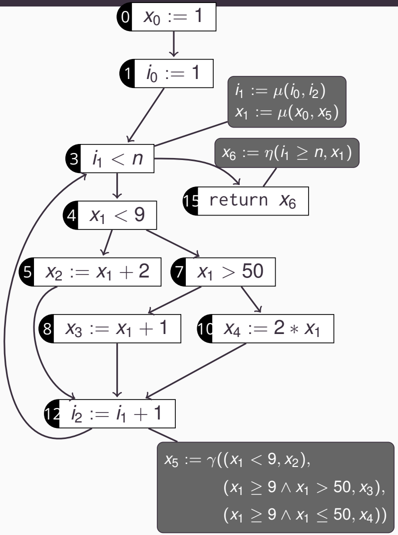
Idea: decouple loop-carried variable from end-of-loop usage

Gate p signals when r_s has reached a stable value

Gated SSA (GSA): example



SSA



GSSA: extends ϕ-instr. with gates

Gated SSA: State of Affairs

Recent usages

HLS, GPU code gen., parallelizing compilers

Non-verified translation validation for LLVM [Tristan et al., 2011]

Key component, alas not described in papers!

Numerous variants

Each come with own notion of dependencies

No reference implementation, no specification

No formal semantics, partial and informal prose

Gated SSA: State of Affairs

Recent usages

HLS, GPU code gen., parallelizing compilers

Non-verified translation validation for LLVM [Tristan et al., 2011]

Key component, alas not described in papers!

Numerous variants

Each come with own notion of dependencies

No reference implementation, no specification

No formal semantics, partial and informal prose

$\Rightarrow$ We need a specification and a semantics for this critical component

Gated SSA: State of Affairs

Recent usages

HLS, GPU code gen., parallelizing compilers

Non-verified translation validation for LLVM [Tristan et al., 2011]

Key component, alas not described in papers!

Numerous variants

Each come with own notion of dependencies

No reference implementation, no specification

No formal semantics, partial and informal prose

$\Rightarrow$ We need a specification and a semantics for this critical component

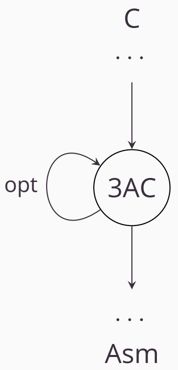
Disclaimer

Necessarily geared to application case

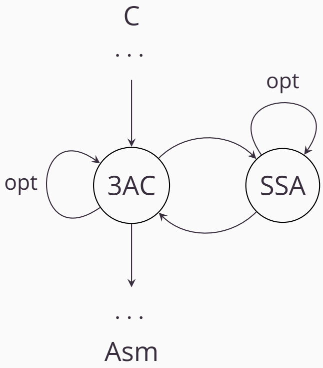
Baby steps: focus on gates and generation

No performance evaluation yet!

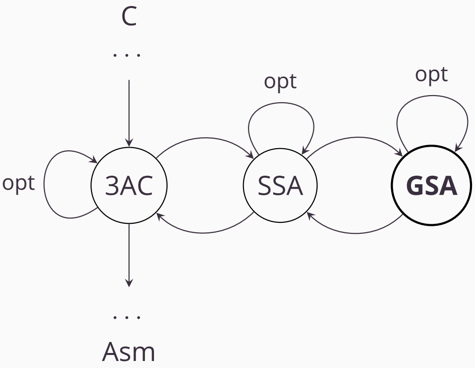
Overview of Implementation in CompCert



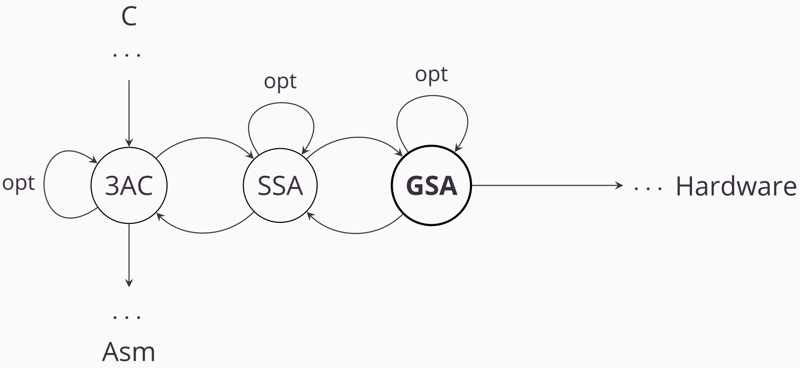
Overview of Implementation in CompCert



Overview of Implementation in CompCert



Overview of Implementation in CompCert



Proof of SSA to GSA Translation

Verified Compilers: Semantics Preservation

```
Theorem compiler_correct: forall P P' behavior,
  compiler P = OK P' ->
  prog_asm_exec P' behavior ->
  prog_src_exec P  behavior.
Proof. [...] Qed.
```

Verified Compilers: Semantics Preservation

```
Theorem compiler_correct: forall P P' behavior,
   compiler P = OK P' ->
   prog_asm_exec P' behavior ->
   prog_src_exec P  behavior.
Proof. [...] Qed.
```

1. Define syntax and semantics for languages:
   Coq data-structures, Coq relations
2. Program the compiler: Coq function
3. State the correctness theorem: Coq property
4. Prove it, using a simulation diagram: Coq proof script

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

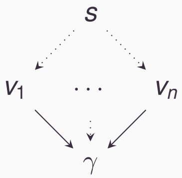
Single-source path expression problem
"Find, for each vertex v, a regular expression $P(s, v)$ which represents the set of all paths in G from s to v." — [Tarjan, 1981]

μ instructions can be translated directly from ϕ instructions.

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]

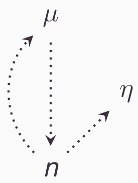
For every future γ node, get a path-expression from the dominator s to each of its predecessors v_1, v_2, ..., v_n.



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 η node, get a path-expression from the corresponding μ to this node.



Different Ways of Verifying a Compiler Pass

Ideally you want to *fully verify* the translation.

What does that mean?
No proof code should be present at *runtime*.

Why might that not be possible?
Properties might be easy to check but tedious to prove.

Different Ways of Verifying a Compiler Pass

Ideally you want to *fully verify* the translation.

What does that mean?
No proof code should be present at *runtime*.

Why might that not be possible?
Properties might be easy to check but tedious to prove.

LEMMA 1. *Let $(P_1, v_1, w_1), (P_2, v_2, w_2), \ldots, (P_l, v_l, w_l)$ be a path sequence for G and let v be any vertex. After i iterations of the loop in SOLVE, $P(s, v)$ is an unambiguous path expression representing exactly Λ (if $s = v$) and all nonempty paths p from s to v for which there is a sequence of indices $1 \leq i_1 < i_2 < \cdots < i_k \leq i$ and a partition of p into $p = p_1, p_2, \ldots, p_k$ such that $p_j \in \sigma(P_{i_j})$ for $1 \leq j \leq k$.*

PROOF. Straightforward by induction on ι. □

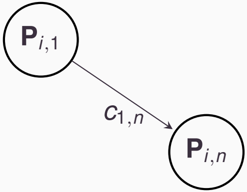
Verifying GSA Generation by Checking Properties

Instead of proving correctness of *path expressions*, check properties of *predicates*.

Verifying GSA Generation by Checking Properties

Instead of proving correctness of *path expressions*, check properties of *predicates*.

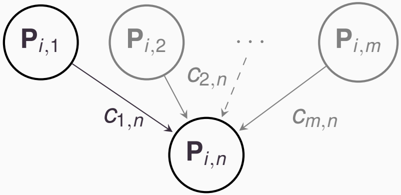
One core *invariant* we want to maintain is *predicate evaluation*:



Verifying GSA Generation by Checking Properties

Instead of proving correctness of *path expressions*, check properties of *predicates*.

One core *invariant* we want to maintain is *predicate evaluation*:

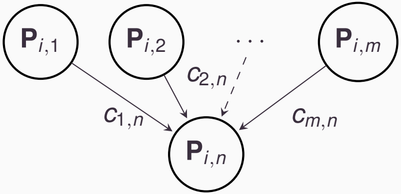


Coherence Property

(Local) Coherence property $f \models \mathbf{P}\ \mathrm{coh}\ (i, n)$

f is the SSA function

i and n are nodes in CFG of f, with i strictly dominates n



$$\bigvee_{p \in \mathrm{preds}(f,n)} (\mathbf{P}_{i,p} \wedge c_{p,n})$$

$$\Downarrow$$

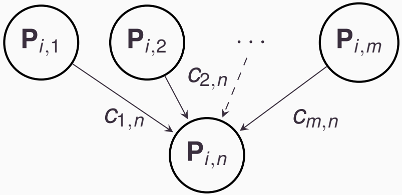
$$\mathbf{P}_{i,n}$$

Coherence Property

(Local) Coherence property $f \models \mathbf{P}\ \mathrm{coh}\ (i, n)$

f is the SSA function

i and *n* are nodes in CFG of *f*, with *i* strictly dominates *n*



$$\bigvee_{p \in \mathrm{preds}(f,n)} (\mathbf{P}_{i,p} \wedge c_{p,n})$$

$$\Downarrow$$

$$\mathbf{P}_{i,n}$$

Evaluability of predicates

Predicates: piece of syntax

Variables in conditions not always defined at runtime: use of a 3-valued logic

Well-exclusivity of γ Predicates

Intuition

In $r_d \leftarrow \gamma((p_1, r_1), (p_2, r_2))$,
p_1 and p_2 must be enough to pick one r_i

Well-exclusivity of γ Predicates

Intuition

In $r_d \leftarrow \gamma((p_1, r_1), (p_2, r_2))$,
p_1 and p_2 must be enough to pick one r_i

Definition (Mutually exclusive predicates)

p_1 and p_2 are mutually exclusive, written $p_1 \ltimes p_2$, whenever for all registers state rs they cannot both evaluate to true, *i.e.* if $rs \models_p p_1 \Downarrow 1$, then $rs \models_p p_2 \not\Downarrow 1$.

Using an SMT Solver to Check Properties

- The properties we are trying to check are arbitrary logic properties.
- The solver needs to use three-valued logic.
- SMT solvers can do all this.

$$\bigvee_{p \in \mathrm{preds}(f,n)} (\mathbf{P}_{i,p} \wedge c_{p,n})$$

$$\Downarrow$$

$$\mathbf{P}_{i,n}$$

Verifying an SMT Solver

We are not done... The SMT solver would have to be trusted, which does not integrate with our proof.

[1] https://smtcoq.github.io/

Verifying an SMT Solver

We are not done... The SMT solver would have to be trusted, which does not integrate with our proof.

SMTCoq[1] is a formalisation of SMT unsatisfiability proofs.

However, their main use case is as:

1. a standalone tool, or
2. as a Coq tactic to solve Theorems in Coq.

[1] `https://smtcoq.github.io/`

Verifying an SMT Solver

We are not done... The SMT solver would have to be trusted, which does not integrate with our proof.

SMTCoq[1] is a formalisation of SMT unsatisfiability proofs.

However, their main use case is as:

1. a standalone tool, or
2. as a Coq tactic to solve Theorems in Coq.

We need a checker that can be integrated into the compiler, which will give us the same correctness guarantees.

[1] `https://smtcoq.github.io/`

Integrating the Verified Unsat Checker

The main workflow to prove the SMT solver:

1. Convert recursive predicates into efficient flat list structure using linear arithmetic to implement three-valued logic:
   $P_1 \wedge (P_2 \vee P_3)$ into

$$
\begin{aligned}
&-1 \leq P_1 \leq 1 \wedge -1 \leq P_2 \leq 1 \wedge -1 \leq P_3 \leq 1 \wedge -1 \leq P_4 \leq 1 \wedge -1 \leq P_5 \leq 1 \\
&\wedge (P_2 < P_3 ? P_4 == P_3 : P_4 == P_2) \\
&\wedge (P_1 < P_4 ? P_5 == P_1 : P_5 == P_4) \\
&\wedge \neg(P_5 == 1)
\end{aligned}
$$

Integrating the Verified Unsat Checker

The main workflow to prove the SMT solver:

1. Convert recursive predicates into efficient flat list structure using linear arithmetic to implement three-valued logic:
   $P_1 \wedge (P_2 \vee P_3)$ into

$$
\begin{aligned}
& -1 \leq P_1 \leq 1 \wedge -1 \leq P_2 \leq 1 \wedge -1 \leq P_3 \leq 1 \wedge -1 \leq P_4 \leq 1 \wedge -1 \leq P_5 \leq 1 \\
& \wedge (P_2 < P_3 ? P_4 == P_3 : P_4 == P_2) \\
& \wedge (P_1 < P_4 ? P_5 == P_1 : P_5 == P_4) \\
& \wedge \neg(P_5 == 1)
\end{aligned}
$$

2. Reverse engineer any optimisations that SMTCoq would do on Coq goals.

Integrating the Verified Unsat Checker

The main workflow to prove the SMT solver:

1. Convert recursive predicates into efficient flat list structure using linear arithmetic to implement three-valued logic:
   $P_1 \land (P_2 \lor P_3)$ into

$$
\begin{aligned}
& -1 \leq P_1 \leq 1 \land -1 \leq P_2 \leq 1 \land -1 \leq P_3 \leq 1 \land -1 \leq P_4 \leq 1 \land -1 \leq P_5 \leq 1 \\
& \land (P_2 < P_3 ? P_4 == P_3 : P_4 == P_2) \\
& \land (P_1 < P_4 ? P_5 == P_1 : P_5 == P_4) \\
& \land \neg(P_5 == 1)
\end{aligned}
$$

2. Reverse engineer any optimisations that SMTCoq would do on Coq goals.
3. Prove semantic preservation between initial predicates and SMTCoq formulas.

Finalising the Proof and Caveats

We can finally prove the SSA to GSA translation *without assumptions*.

Finalising the Proof and Caveats

We can finally prove the SSA to GSA translation *without assumptions*.

Many Limitations

- Very slow compilation time due to many SMT checks.

Finalising the Proof and Caveats

We can finally prove the SSA to GSA translation *without assumptions*.

Many Limitations

- Very slow compilation time due to many SMT checks.
- Some comparisons are not supported (`(unsigned)x == (unsigned)y`).

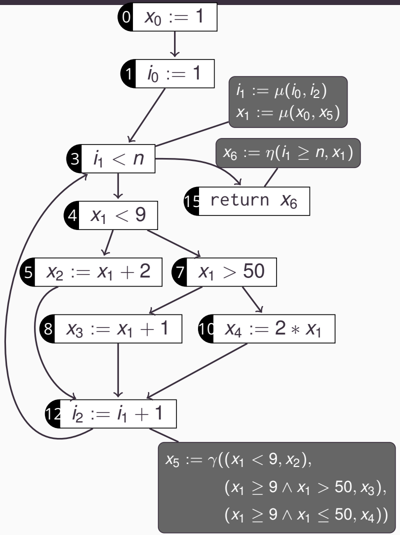
Finalising the Proof and Caveats

We can finally prove the SSA to GSA translation *without assumptions*.

Many Limitations

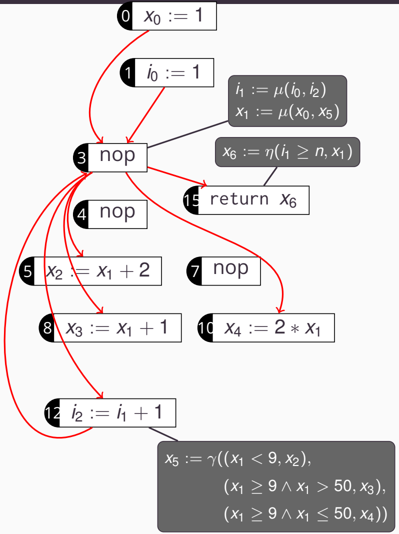
- Very slow compilation time due to many SMT checks.
- Some comparisons are not supported (`(unsigned)x == (unsigned)y`).
- Destruction of GSA is currently not proven correct.

A Note on GSA to Hardware Conversion



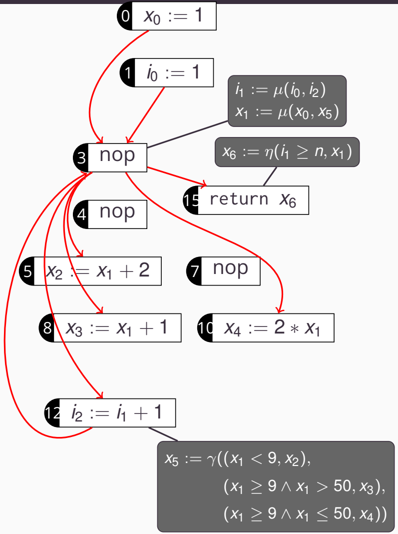
- Currently we only implemented control-flow semantics for GSA.

A Note on GSA to Hardware Conversion



- Currently we only implemented control-flow semantics for GSA.
- One can formulate Dataflow semantics.

A Note on GSA to Hardware Conversion



- Currently we only implemented control-flow semantics for GSA.
- One can formulate Dataflow semantics.
- It should map quite nicely to circuits (however efficiency becomes an issue).

Summary and On-going Work

Summary, and On-going Work

Implementation within CompCertSSA
Prior pass needed for Gated SSA: loop normalization
Gated SSA: syntax and semantics
Correct generation of Gated SSA

On-going work: destruction of Gated SSA to SSA

- Rebuild control-flow information, conforms to CFG

Future work: 3 Ph.D. projects starting in Rennes and London
Full-fledge gated SSA as a dependency graph
Integrate into verified dynamic HLS toolchain

Thank you

Any Questions?

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

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

Join

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

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

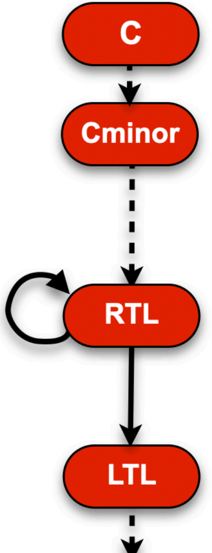
Tristan, J.-B., Govereau, P., and Morrisett, G. (2011).
Evaluating value-graph translation validation for LLVM.
In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 295–305, New York, NY, USA. Association for Computing Machinery.

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.

Verified Compilers: CompCert

X.Leroy, S.Blazy et al. 2005-present

https://compcert.org/

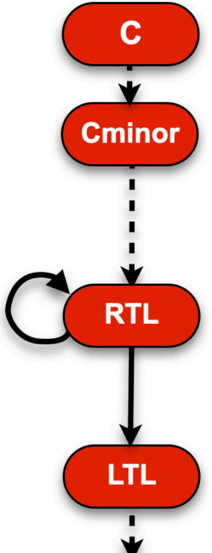


From CompCert C down to Assembly
20 passes, 11 IRs, targets PPC, ARM, x86, Risc-V
Optos: const. prop., CSE, DCE, tailcalls, inlining
Formally verified using the Coq proof assistant
Compiler programmed, specified, and proved in Coq
Extracted to efficient OCaml code

Verified Compilers: CompCert

X.Leroy, S.Blazy et al. 2005-present

https://compcert.org/



From CompCert C down to Assembly
20 passes, 11 IRs, targets PPC, ARM, x86, Risc-V
Optos: const. prop., CSE, DCE, tailcalls, inlining
Formally verified using the Coq proof assistant
Compiler programmed, specified, and proved in Coq
Extracted to efficient OCaml code
CompCert is mature, commercialized by AbsInt
Airbus (fly-by-wire soft.), MTU (control soft. for emergency power generators)
Conformance to the certification process IEC 60880
Performance gain in estimated WCET
2022: ACM Software System award, ACM SIGPLAN Programming Languages Software award

This Work

- Gated SSA, a compiler IR famous for:
  - optimizations in parallelizing compilers [Arenaz et al., 2008]
  - high-level synthesis [Derrien et al., 2020]
  - code generation for GPUs [Sampaio et al., 2012]
- Semantics and correctness of generation
- Focus on gates, in isolation of other challenges

Static Single Assignment (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

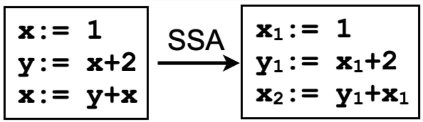
Static Single Assignment (SSA)

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



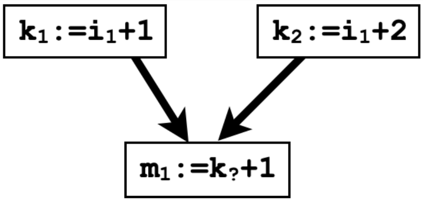
Static Single Assignment (SSA)

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!



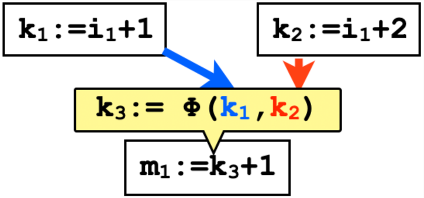
Static Single Assignment (SSA)

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

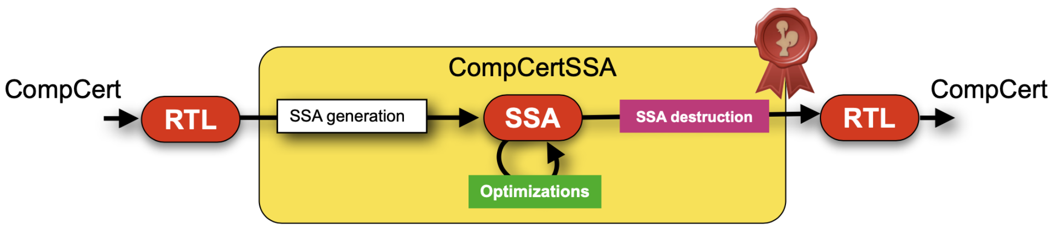
Now widely adopted in compiler community
GCC, LLVM, Java HotSpot JIT, . . .

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 $x_3 \leftarrow \phi(x_1, x_2)$
Based on control-flow, select right argument



CompCertSSA: an SSA-based Middle-end for CompCert



Middle-end: optimization

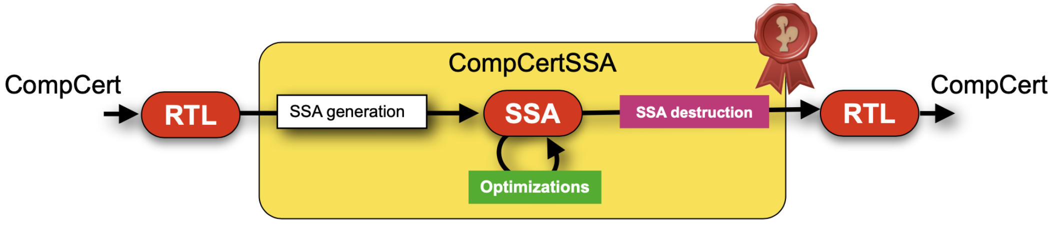
RTL: 3-address code, virtual registers, CFG representation

SSA: RTL + ϕ-instructions + invariants

Realistic implementation: GVN, sparse cond. c. pro., coalescing

State-of-the-art, similar to LLVM and GCC

CompCertSSA: an SSA-based Middle-end for CompCert



Middle-end: optimization

RTL: 3-address code, virtual registers, CFG representation

SSA: RTL + ϕ-instructions + invariants

Realistic implementation: GVN, sparse cond. c. pro., coalescing

State-of-the-art, similar to LLVM and GCC

Ultimate goals

Understand semantic foundations of SSA techniques

Same formal guarantees as CompCert

No negative impact on code performance

SSA: semantics

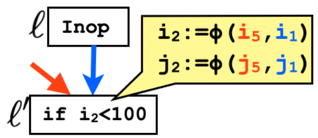
Challenges: integrate well in CompCert compiler chain
Be close to RTL semantics
Be as intuitive as informal definition given in [Alpern et al., 1988]

Execution states and transition relation, as in RTL

$$\vdash \mathcal{S}(f, l, rs) \rightarrow \mathcal{S}(f, l', rs')$$

Execute in a single small-step:

1. current instruction
2. and potential ϕ-block at successor label



Remarks:
Prior RTL normalization: only an `Inop` can lead to a join point
Parallel assignment semantics for ϕ-blocks

From SSA to Gated SSA

SSA strengths

CFG-based representation: simple operational semantics

ϕ-functions already capture def/use dependencies

SSA weaknesses

Semantics of ϕ-functions depends on control-flow

Non-local semantics of ϕ-functions: $\mathcal{S}(f, l, rs)$ not enough

Some dependencies are still implicit

Gated SSA: gates turn control-dep. into data-dep.

Building block of Program Dependence Web [Ottenstein et al., 1990]

Ignore some dependencies [Havlak, 1994]

Symbolic analysis for parallelizing compiler [Tu and Padua, 1995]

Gated SSA: State of Affairs

Key component, alas not described in papers!

Numerous variants
Each come with own notion of dependencies
No reference implementation, no specification
No formal semantics, partial and informal prose

$\Rightarrow$ We need a semantics and some expected properties for this critical component

Disclaimer
Baby steps: focus on gates and generation
No performance evaluation yet!

Invariants for Gates: Coherence and Exclusivity

Predicates: Main technical point in generation algorithm
Generation algorithm: Single-source path expression problem (regexp on path cond.) [Tarjan, 1981]

Predicate matrix P
Gates: syntactical, global information
$\mathbf{P}_{i,j}$ = set of paths from *i* to *j* in CFG of *f*

Two intrinsic properties for $r_d \leftarrow \gamma((p_1, r_1), (p_2, r_2))$

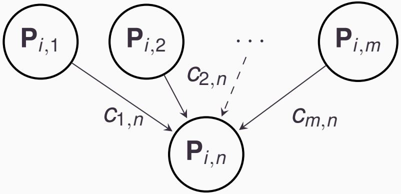
- Coherence: gates are characterizing correct paths
- Well-exclusivity: gates in γ-functions are precise enough
  Intuition: in $r_d \leftarrow \gamma((p_1, r_1), (p_2, r_2), (p_3, r_3))$,
  p_1, p_2 and p_3 must be enough to pick one r_i

Coherence Property of Predicates

(Local) Coherence property $f \models \mathbf{P}\ \mathrm{coh}\ (i, n)$

f is the SSA function

i and n are nodes in CFG of f, with i strictly dominates n



$$\bigvee_{p \in \mathrm{preds}(f,n)} (\mathbf{P}_{i,p} \wedge c_{p,n})$$

$$\Downarrow$$

$$\mathbf{P}_{i,n}$$

Evaluability of predicates

Predicates: piece of syntax

Variables in conditions not always defined at runtime: use of a 3-valued logic

Summary, and On-going Work

Implementation within CompCertSSA
Prior pass needed for Gated SSA: loop normalization
Gated SSA: syntax and semantics
Correct generation of Gated SSA

On-going work: destruction of Gated SSA to SSA

- Rebuild control-flow information, conforms to CFG

Future work: 3 Ph.D. projects starting in Rennes and London
Full-fledge gated SSA as a dependency graph
Integrate into verified dynamic HLS toolchain