Mechanized Verification of SSA-based Compilers Techniques

Delphine Demange - U. Rennes 1 / IRISA / Inria Rennes

Joint work with G. Barthe, S. Blazy, Y. Fernandez de Retana, D. Pichardie, L. Stefanesco

GDR-IM - Villetaneuse - 19/01/16

IRISA

Context: Compiler Correctness

Compiler correctness

Compilers still contain bugs!

*"We found and reported **hundreds** of previously **unknown bugs** [...]. Many of the bugs we found cause a compiler to **emit incorrect code without any warning.** 25 of the bugs we reported against GCC were classified as **release-blocking.**"*

Regher et al. [PLDI'11]

GCC
SML/NJ
MLton
OCaml
HotSpot
LLVM

Knowledge

Well-known results and techniques
Algorithms are justified
Sometimes proofs

Implementation
Compilers

We want to bridge the gap between knowledge and practical implementations

→ bring some formal guarantee

Formal guarantee

About what ?

We want to verify what actually runs
the ***code*** of the compiler

What kind?

Test ? detect the presence of bug

→ ***Proof ! prove that the compiler meets its spec***

How do we prove this?

Fully manually, pen and paper proof
like in mathematics
does not scale (complex language, gigantic programs...)

→ ***Interactive formal proof (proof assistant)***
computers are better than humans at checking large proofs
machine-assisted, machine-checked, mechanized
it does scale to real-world programs

Fully automatically (static analysis, model checking...)
undecidability : use over-approximations
not scalable (yet?) to compilers

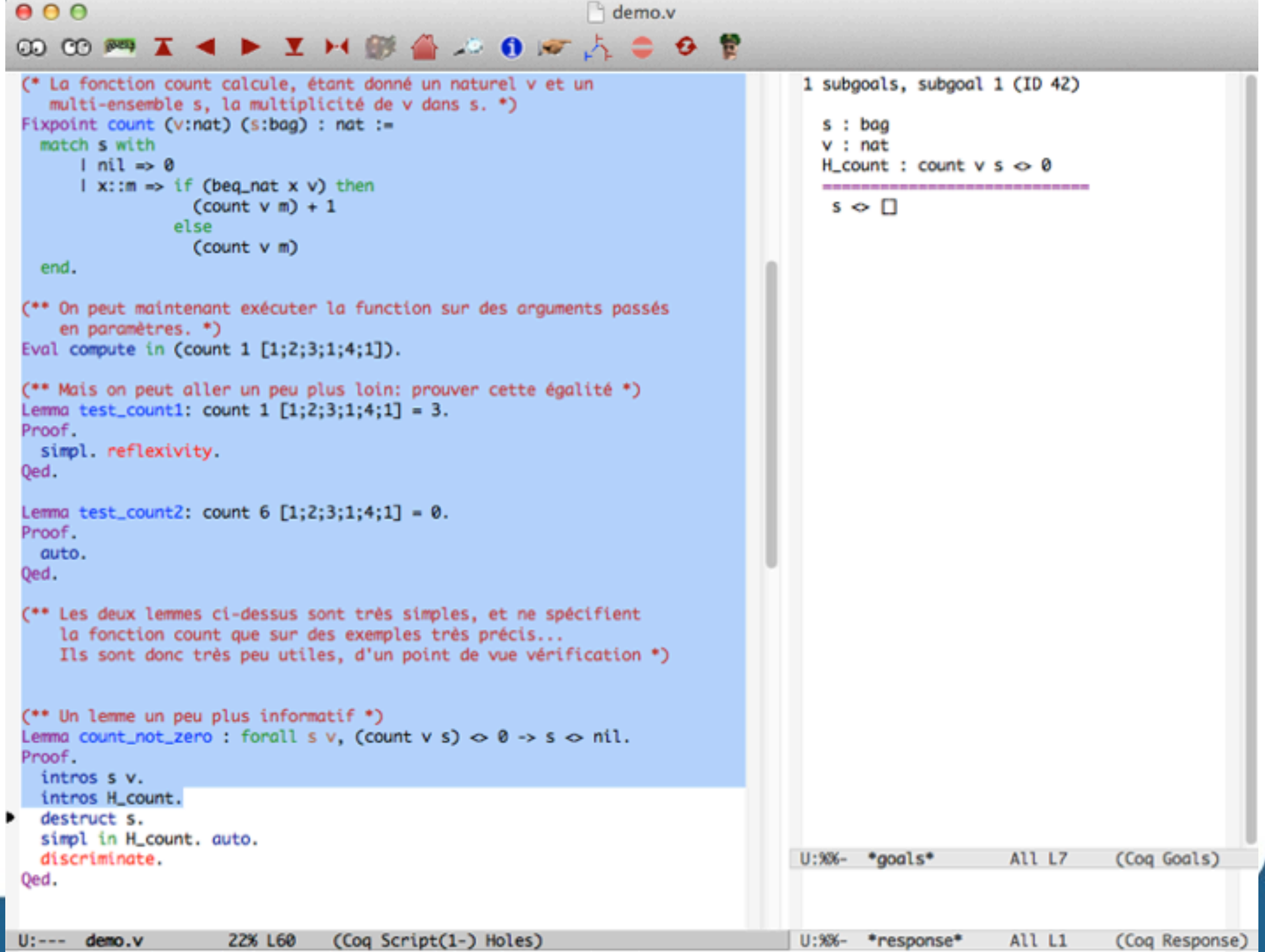
Proof assistants

Proof assistants implements well-defined mathematical logics. Coq implements the Calculus of Inductive Constructions.

A (purely functional) language
write definitions (data types, functions, predicates) and state software specifications and theorems.

Build proofs in interaction with the user

Check the validity of proofs in the logic, trusting only a type-checking kernel.


demo.v
(* La fonction count calcule, étant donné un naturel v et un
multi-ensemble s, la multiplicité de v dans s. *)
Fixpoint count (v:nat) (s:bag) : nat :=
match s with
| nil => 0
| x::m => if (beq_nat x v) then
(count v m) + 1
else
(count v m)
end.
(** On peut maintenant exécuter la function sur des arguments passés
en paramètres. *)
Eval compute in (count 1 [1;2;3;1;4;1]).
(** Mais on peut aller un peu plus loin: prouver cette égalité *)
Lemma test_count1: count 1 [1;2;3;1;4;1] = 3.
Proof.
simpl. reflexivity.
Qed.
Lemma test_count2: count 6 [1;2;3;1;4;1] = 0.
Proof.
auto.
Qed.
(** Les deux lemmes ci-dessus sont très simples, et ne spécifient
la fonction count que sur des exemples très précis...
Ils sont donc très peu utiles, d'un point de vue vérification *)
(** Un lemme un peu plus informatif *)
Lemma count_not_zero : forall s v, (count v s) <> 0 -> s <> nil.
Proof.
intros s v.
intros H_count.
destruct s.
simpl in H_count. auto.
discriminate.
Qed.
1 subgoals, subgoal 1 (ID 42)
s : bag
v : nat
H_count : count v s <> 0
s <> []
U:%%- *goals* All L7 (Coq Goals)
U:--- demo.v 22% L60 (Coq Script(1-) Holes)
U:%%- *response* All L1 (Coq Response)

Verified compilers - CompCert C compiler

Leroy et al. 2005-present

http://compcert.inria.fr

From CompCert C down to assembly:

18 passes, 10 IRs, PowerPC, ARM, x86 backends

Optimizing compiler:

tail calls, inlining, const prop, CSE, dead code elim.

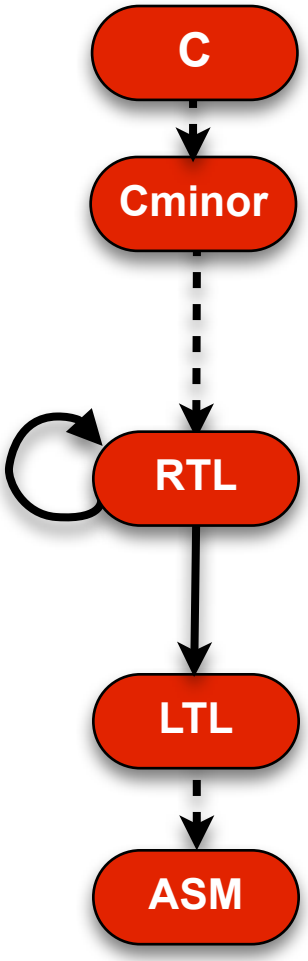
Formally verified using the Coq proof assistant

Written, specified and proved in Coq.

Extracted to efficient OCaml code

Each pass is formally proved (semantics preserving):

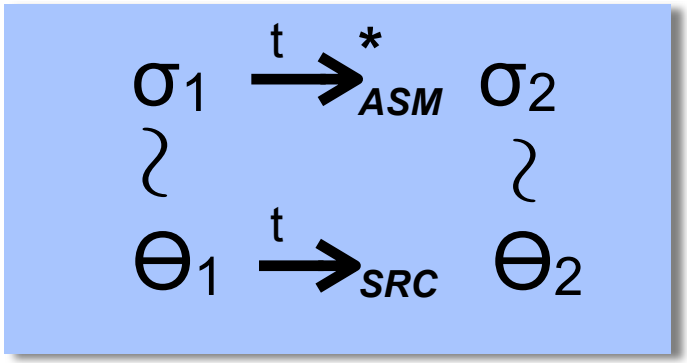
```
Theorem compiler_correct: ∀ P P' behavior,
   compiler P = OK P' →
   prog_asm_exec P' behavior →
   prog_src_exec P  behavior.
Proof. [...] Qed.
```



Formal proof of semantics preservation

```
Theorem compiler_correct: ∀ P P' behavior,
   compiler P = OK P' →
   prog_asm_exec P' behavior →
   prog_src_exec P  behavior.
Proof. [...] Qed.
```

1. Define language syntax
   Coq data-structures
2. Program the compiler
   Coq function
3. Define semantics of language
   Coq function or relation
4. State the correctness theorem
   Coq property
5. Prove it (simulation diagram)
   Coq proof script



Verified compilers - CompCert C compiler

Leroy et al. 2005-present

http://compcert.inria.fr

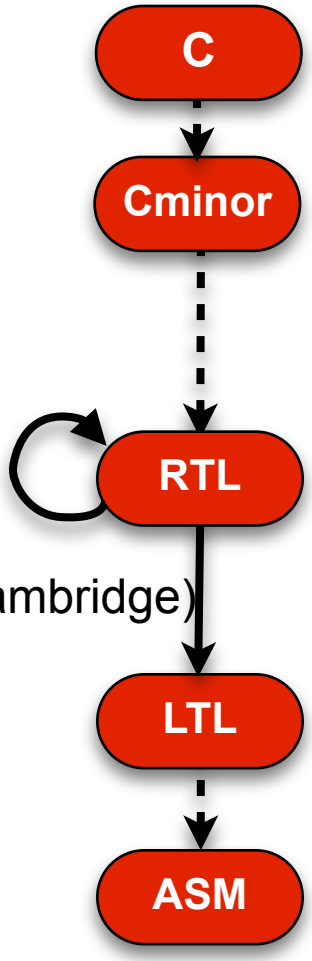
CompCert is mature (entered industrial world!):

- Airbus (fly-by-wire software)
- Performance gain in estimated WCET
- Conformance to the certification process (DO-178)
- Several awards! (MSR, La Recherche)

Various (academic) extensions:

- Other source languages, and backends
- Concurrency - VST (Princeton), CompCertTSO (Cambridge)
- Separate compilation, linking (Appel, Kang et al.)
- Optimization techniques

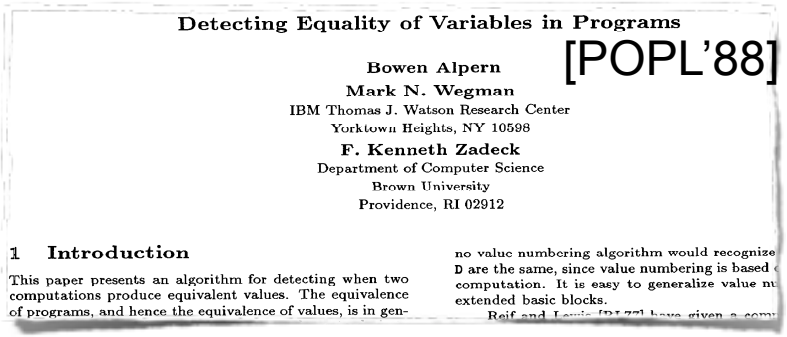
this talk



Single Static Assignment (SSA)

Static Single Assignment (SSA)

Introduced late 80's by Alpern et al.



Detecting Equality of Variables in Programs

[POPL'88]

Bowen Alpern
Mark N. Wegman
IBM Thomas J. Watson Research Center
Yorktown Heights, NY 10598
F. Kenneth Zadeck
Department of Computer Science
Brown University
Providence, RI 02912

1 Introduction

This paper presents an algorithm for detecting when two computations produce equivalent values. The equivalence of programs, and hence the equivalence of values, is in gen-

no value numbering algorithm would recognize D are the same, since value numbering is based computation. It is easy to generalize value nu extended basic blocks.

Reif and Lewis [RL77] have given a com

Widely used in modern compilers

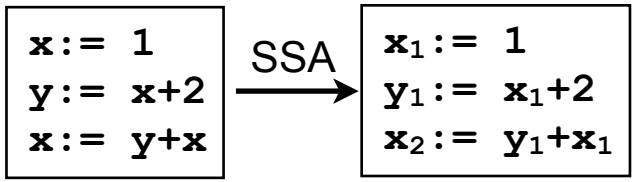
GCC, LLVM, Java Hot Spot JIT...

Variables with unique definition point

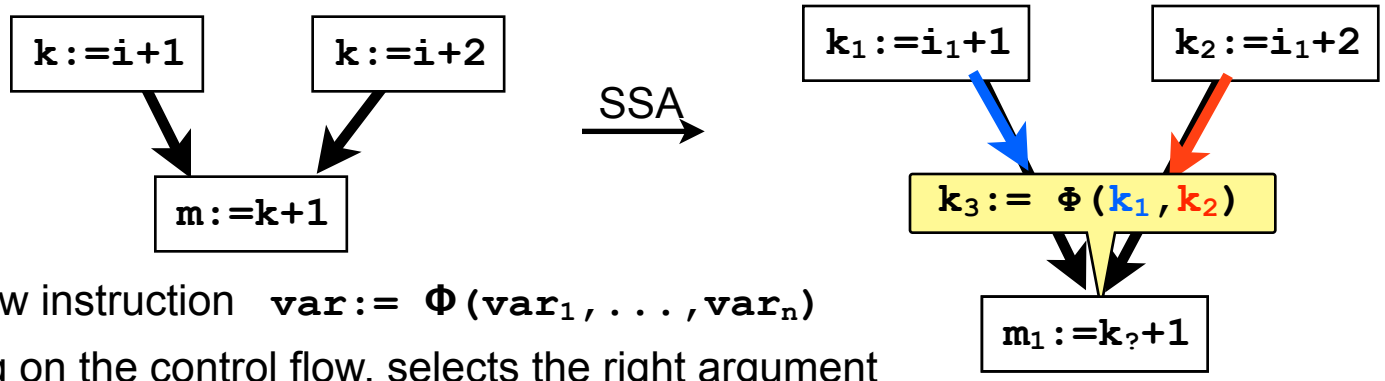
Straight-line code

Associate to each definition a fresh version number

Rename each use with by the right version



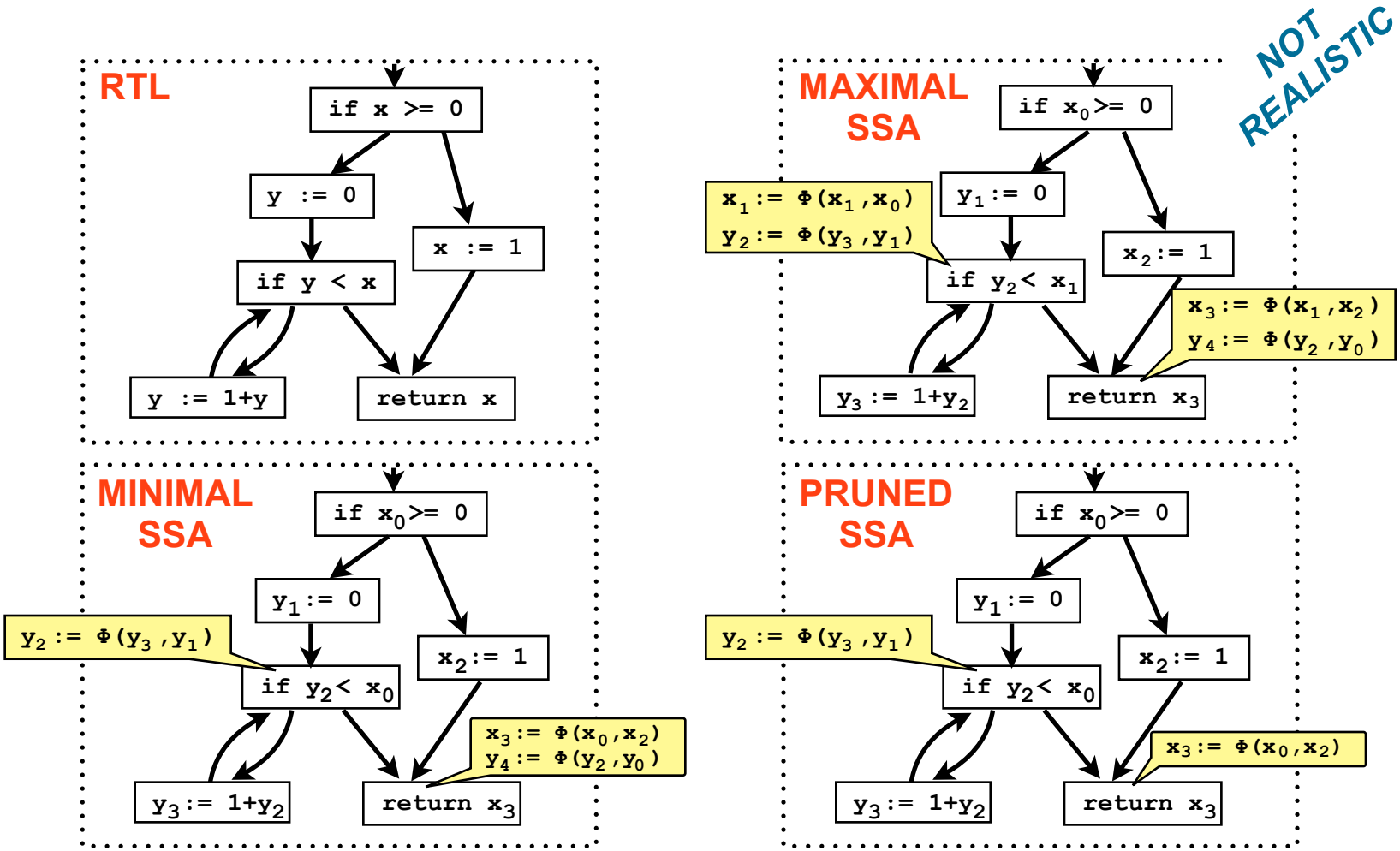
Control-flow join points



Need a new instruction $\mathtt{var} := \Phi(\mathtt{var}_1, \ldots, \mathtt{var}_n)$

Depending on the control flow, selects the right argument

Phi-instruction sites



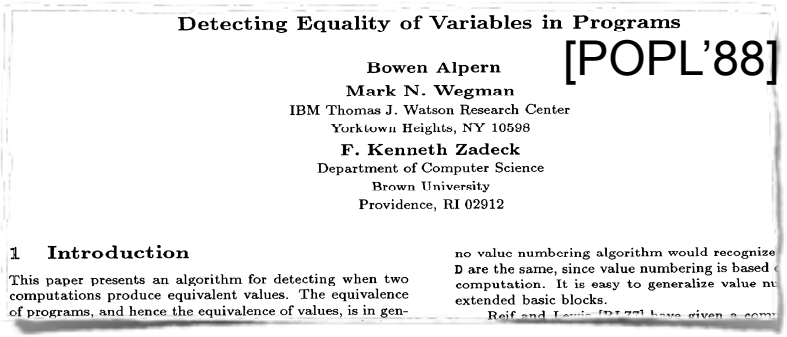
But Minimal and Pruned SSA do not come for free...

Static Single Assignment (SSA)

Introduced late 80's by Alpern et al.

Widely used in modern compilers
GCC, LLVM, Java Hot Spot JIT...

Variables with unique definition point



Detecting Equality of Variables in Programs

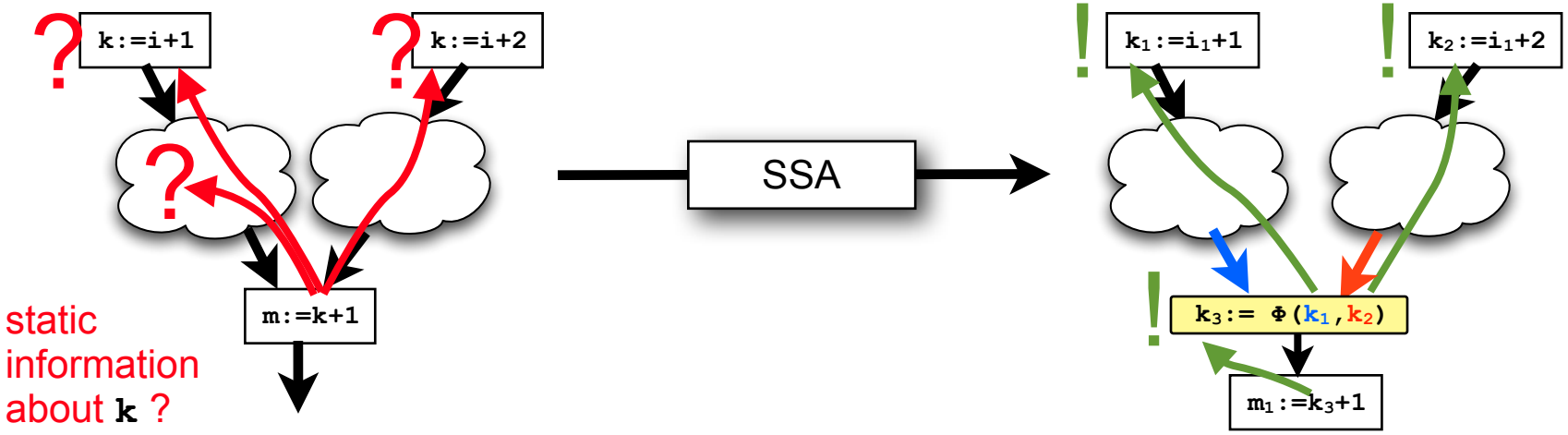
[POPL'88]

Bowen Alpern
Mark N. Wegman
IBM Thomas J. Watson Research Center
Yorktown Heights, NY 10598
F. Kenneth Zadeck
Department of Computer Science
Brown University
Providence, RI 02912

1 Introduction

This paper presents an algorithm for detecting when two computations produce equivalent values. The equivalence of programs, and hence the equivalence of values, is in gen-

no value numbering algorithm would recognize D are the same, since value numbering is based computation. It is easy to generalize value nu extended basic blocks.



Folklore added-values:

- explicit, singleton use-def chains, var uses dominated by definition point
- flow-insensitivity without precision loss, cheap fixpoint iterations

SSA in verified compilers

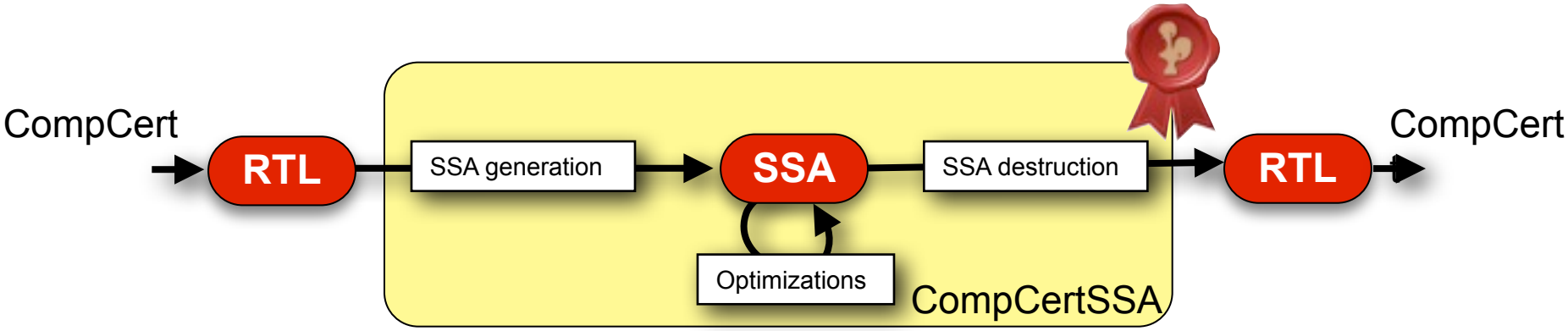
2009: *"Since the beginning of CompCert we [...] were held off by two difficulties. First, the dynamic* ***semantics for SSA*** *is not obvious to formalize. Second, the SSA property is* ***global to the code*** *of a whole function and* ***not straightforward to exploit locally*** *within proofs." X. Leroy.*

Open problem in verified compilation until 2012

2012: Spontaneous generation of realistic proof efforts in this direction

this talk → CompCertSSA

Compiler middle-end design



Realistic implementations

State-of-the-art (LLVM, GCC)

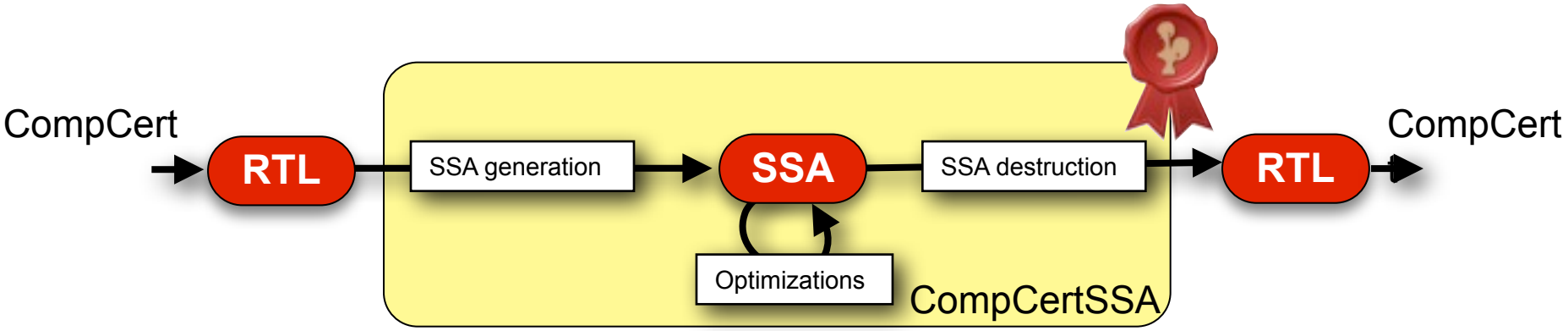
Two major SSA optimizations

SCCP: Sparse Conditional Constant Propagation
GVN: Global Value Numbering (redundant computation elimination)

Ultimate goals

Understand semantic foundations of SSA state-of-the-art techniques
Same formal guarantee as CompCert (complete all proofs)
Integrate without negative impact on performance of compiled code

Compiler middle-end design



Semantics of SSA

previous tries unsuccessful
should be intuitive and non-intrusive
absolutely required

Correctness proofs

sophisticated invariants
dominance based reasoning
proof framework and libraries

Transformations and optimizations

must be fast and powerful
heuristics, clever algorithms
verified validators

Back-end

huge influence on register allocation
lots of heuristics
error-prone

Semantics of SSA

Formal Verification of an SSA-Based Middle-End for CompCert.
G. Barthe, D. Demange, and D. Pichardie. ACM TOPLAS 2014.

Semantics of SSA : small step operational

Challenges:

- be close to RTL semantics (small-step, binary relation on states)
- be as intuitive as the informal definition given in [POPL'88].

Register states:

$$R ::= Var(P) \to Val$$

Program states:

$$\mathcal{S} ::= (\ell, R)$$

Rules when successors have no phi-blocks: same as RTL

$$\frac{code(\ell) = \texttt{nop}\ \ell'}{(\ell, R) \to (\ell', R)}$$

holds also for full SSA

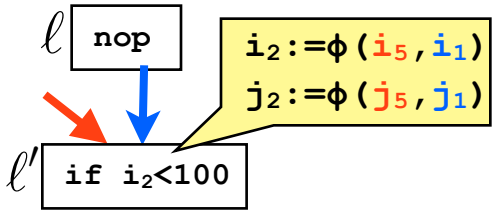
$$\frac{code(\ell) = \texttt{if e}\ \ell_1\ \ell_2 \qquad \ell' = \begin{cases} \ell_1 & \text{if } [\![\,\texttt{e}\,]\!]R \neq 0 \\ \ell_2 & \text{otherwise} \end{cases}}{(\ell, R) \to (\ell', R)}$$

$$\frac{code(\ell) = \texttt{x:= e}\ \ell' \qquad [\![\,\texttt{e}\,]\!]R = v}{(\ell, R) \to (\ell', R[\texttt{x} \mapsto v])}$$

Semantics of SSA : join points

Solution:

Execute in a single step both
the current instruction
and the phi-block at successor point



RTL normalization ensures that only a nop can lead to a junction point

$$code(\ell) = \mathbf{nop}\ \ell' \qquad \mathrm{IndexPred}(\ell, \ell') = k$$

$$phicode(\ell') = \mathbf{x_1 := \phi(args_1) \ldots ; x_n := \phi(args_n)}$$

$$\frac{R' = R[\mathbf{x_1} \mapsto R(\mathbf{args_1[k]}), \mathbf{x_2} \mapsto R(\mathbf{args_2[k]}), \ldots \mathbf{x_n} \mapsto R(\mathbf{args_n[k]})]}{(\ell, R) \rightarrow (\ell', R')}$$

ℓ is the k-th predecessor of ℓ'

k-th element of list

parallel copies

SEMANTICS NOT INSTRUMENTED

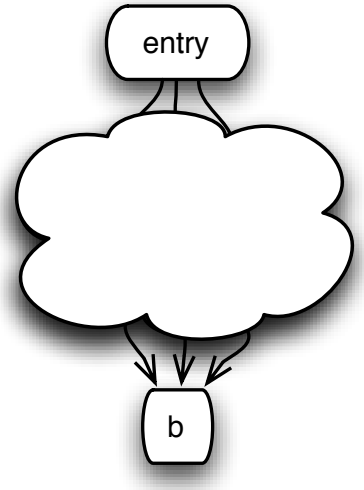
State-of-the-art implementations and verified validators

Formal Verification of an SSA-based Middle-end for CompCert.
G. Barthe, D. Demange, and D. Pichardie. ACM TOPLAS 2014.

Verifying Fast and Sparse SSA-based Optimizations in Coq.
D. Demange, D. Pichardie and L. Stefanesco. CC 2015.

Validating Dominator Trees for a Fast, Verified Dominance Test.
S. Blazy, D. Demange and D. Pichardie. ITP 2015.

Generating Minimal SSA



Φ-instruction sites :

- at dominance frontiers of variables definitions
- **a dominates b**: all paths from entry to **b** must go through **a**

Implementation following Cytron et al. 91

- Compute dominance relation for the CFG

  dominator tree [Lengauer Tarjan 79]

  O(E*α(N))) (optimized version in O(E*α

  data structures

  - complex properties of graphs
- Add Φ-instructions at (iterated) d
- Rename variables

Using the dominator tree

Coq is purely functional
Hardly competitive

*"The algorithm is rather technical; those readers who feel content **just knowing** the dominator tree can be calculated quickly can skip this section"*
A. Appel - Modern Compiler Implementation in ML

Why a direct proof would be challenging?

- The dominator tree

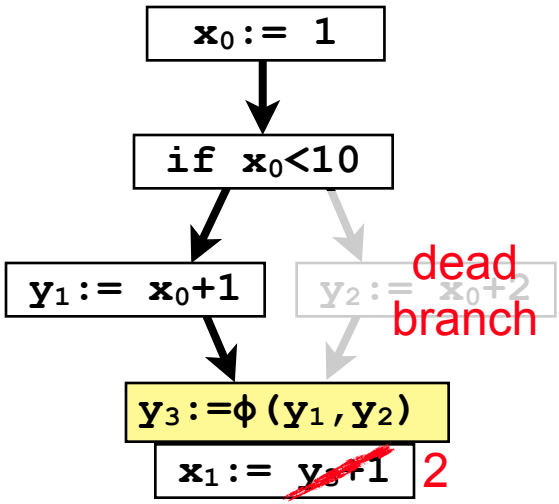
Sparse Conditional Constant Propagation (SCCP)

Two simultaneous optimizations

- detect constant values
- detect infeasible branches

Implementation following Wegman & Zadeck

- Ad hoc iterative workset algorithm with three work sets, over **def-use chains**
- Manage at the same time feasible edges + constant variables



Why a direct proof would be challenging?

- Termination proof

Global Value Numbering (GVN)

GVN discovers equalities between variables

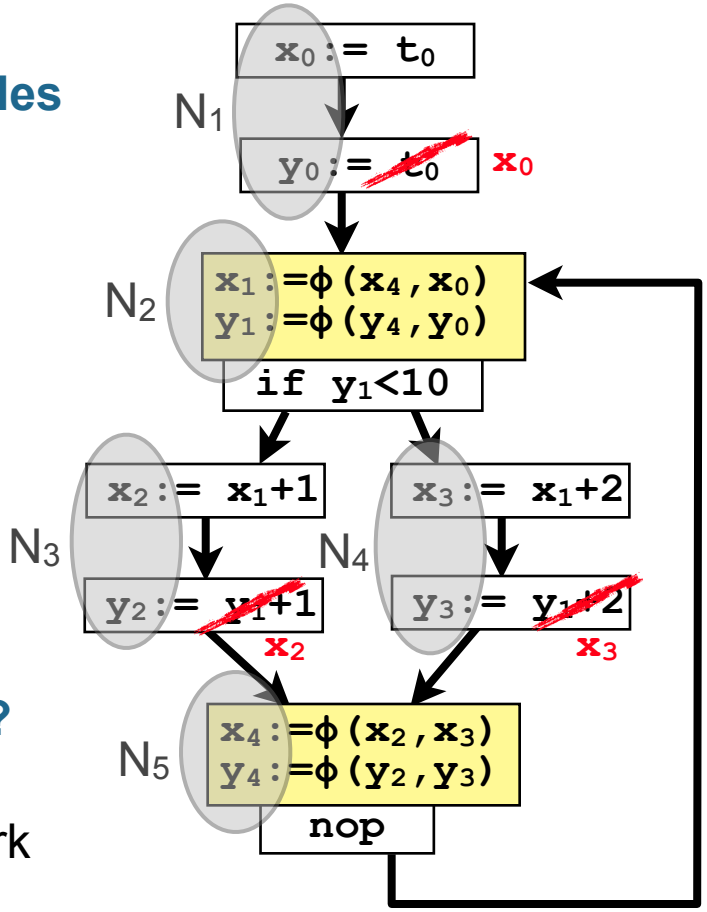
- Build congruence classes of variables
- Works across loop boundaries
- Common Subexpression Elimination

LLVM Implementation

- Scan program in a reverse-post-order
- Hash program expressions to build variable numbering à la BriggsCooper

Why a direct proof would be challenging?

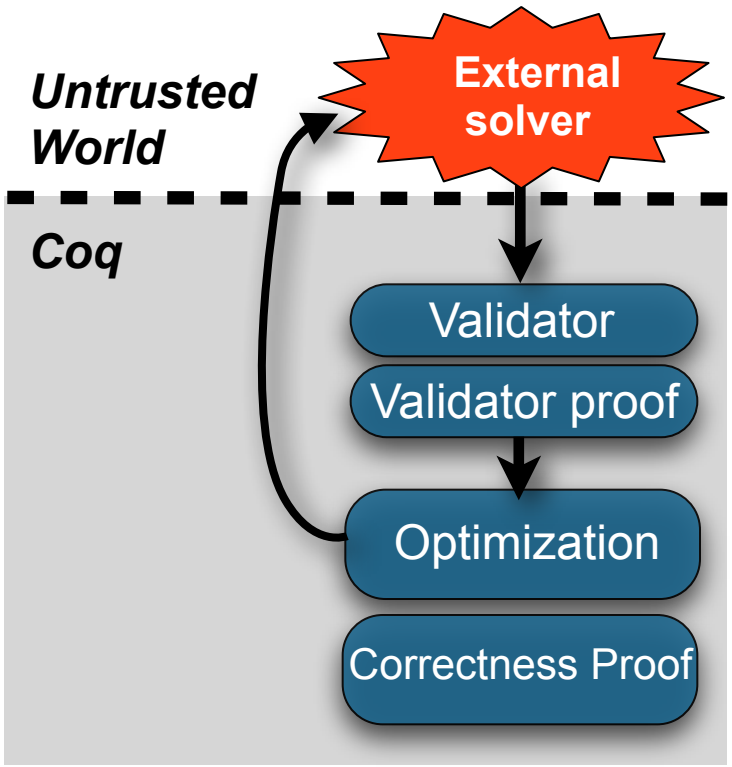
- Efficient mutable data structures
- Not in the classical monotone framework
- Dominator tree for CSE (also in LLVM)



A posteriori validation

Rather than a direct proof of analysis, check its result a posteriori

ex: register allocation and valid graph coloring



mutable data structures
fancy iteration strategy

cheap computation
trivial termination proof
simpler correctness proof

same formal guarantee
(validator may say NO...)

SCCP / GVN:
post-fixpoint validation
(dataflow / congruence)

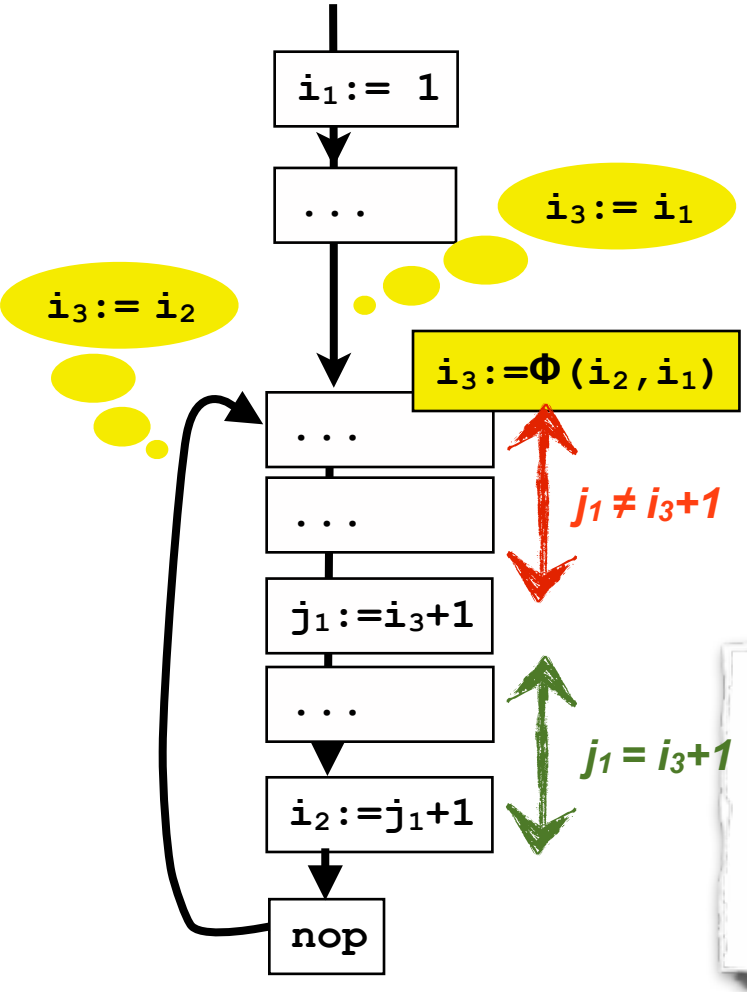
Generation / Dominance:
more specific techniques
(see papers)

Reasoning about SSA

Formal Verification of an SSA-based Middle-end for CompCert.
G. Barthe, D. Demange, and D. Pichardie. ACM TOPLAS 2014.

Verifying Fast and Sparse SSA-based Optimizations in Coq.
D. Demange, D. Pichardie and L. Stefanesco. CC 2015.

Equational lemma



Intuition:

Variable definitions are equations

Use these equations to optimize program

Question:

Are these equations allowed every where in the code ?

Answer:

No. They're valid only on the dominated region of the variable definition !

$$\forall x\, pc, \quad \left.\begin{array}{l} def(x) = pc \\ rhs(x) = e \\ (\ell, R) \in [\![P]\!] \\ pc \ stricly \ dominates \ \ell \end{array}\right\} \Rightarrow R(x) = [\![e]\!]R$$

G. Barthe, D. Demange, and D. Pichardie. *Formal Verification of an SSA-Based Middle-End for CompCert.* ACM Trans. Program. Lang. Syst. 36, 1, Article 4, March 2014.

Fundamental notion: dominance

Dominance relation

Inductive dom: node -> node -> **Prop** := ...

Inductive sdom (pc pc' : node): **Prop** :=
dom pc pc' /\ pc <> pc'.

Lemma dom_order : order node dom.

Lemma sdom_not_dom : forall pc pc',
sdom pc pc' -> ~ dom pc' pc.

10+ other basic utility lemma

Strictly dominated region

Inductive dsd : function -> reg -> node -> **Prop** := ...

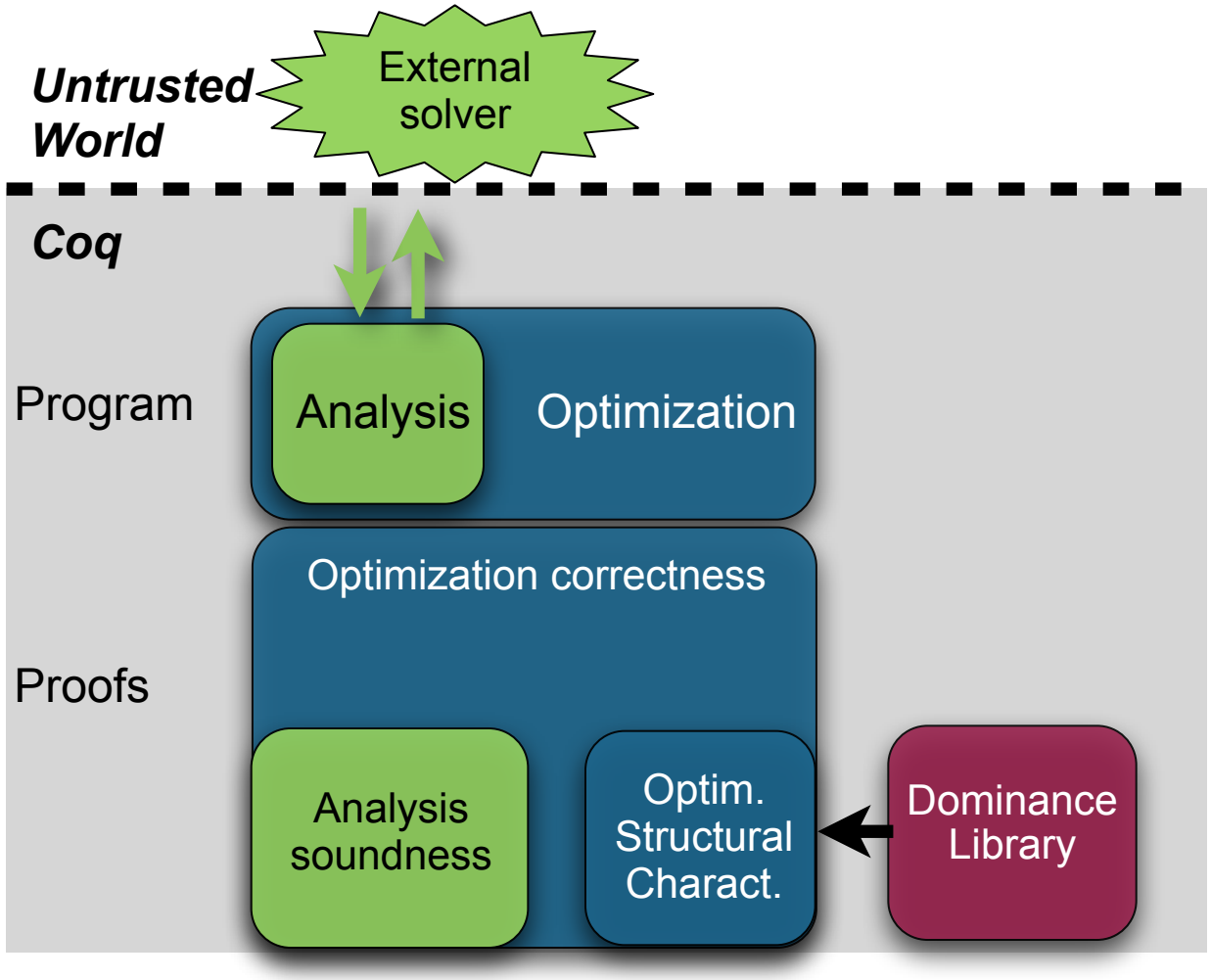
Case analysis scheme (reasoning by induction on the semantics)

Lemma ssa_dsd_entry : forall f x, ~ dsd f x (entry f).

Lemma dsd_not_joinpoint : forall f n1 n2 x,
(edge f n1 n2) /\ (~join_point n2 f) /\ (dsd f x n2) ->
(assigned_code f n1 x)
\/ (params f x /\ n1 = fn_entrypoint f)
\/ (dsd f x n1 /\ ~ assigned_code f n1 x)

Lemma dsd_joinpoint : forall f n1 n2 x,
(edge f n1 n2) /\ (join_point f n2) /\ (dsd f x n2) ->
(assigned_phi f n2 x)
\/ (params f x /\ n1 = entry f)
\/ (dsd f x n1 /\ ~ assigned_phi f n2 x).

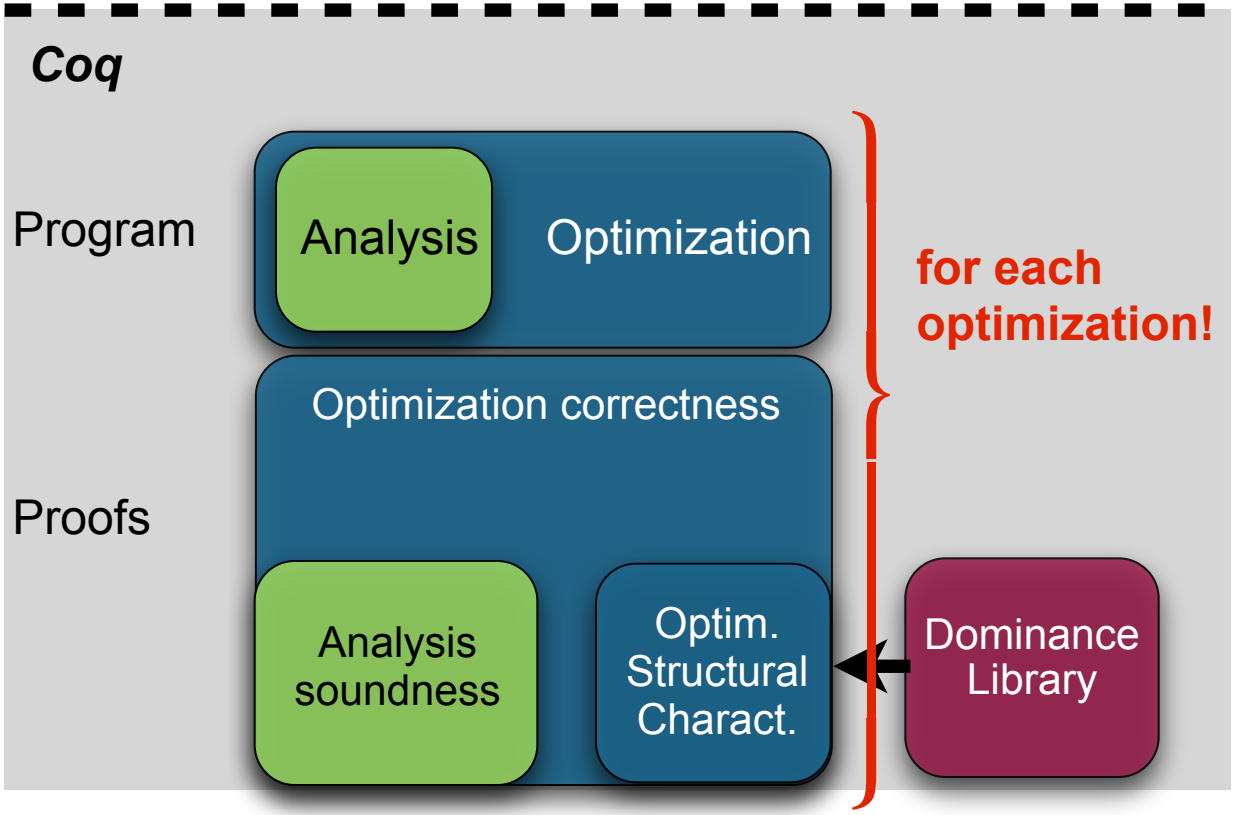
Proving an optimization



Correctness proof

1. Static analysis should
   - ➡ infer sound program invariants
2. Transformation should
   - ➡ preserve SSA structural constraints
   - ➡ preserve program semantics

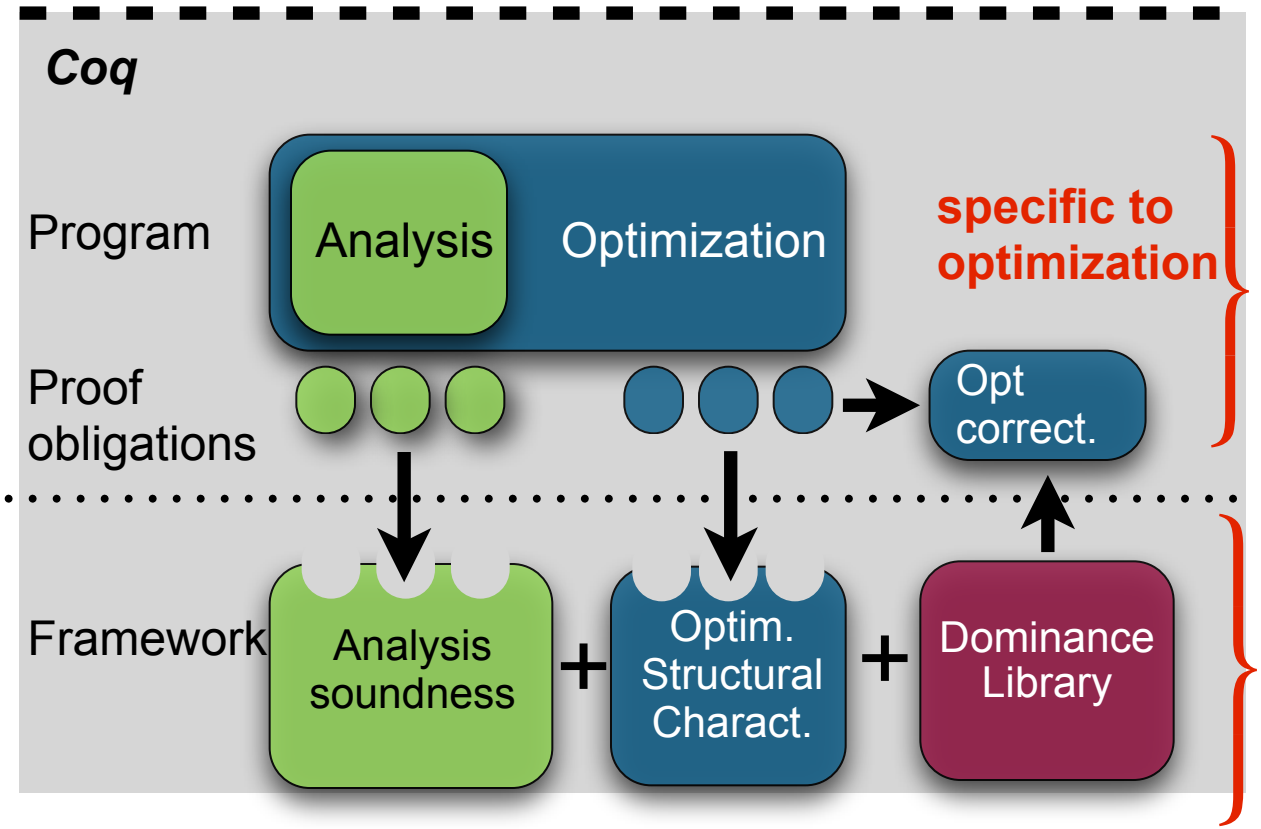
Proving an optimization



Correctness proof

1. Static analysis should
   - ➡ infer sound program invariants
2. Transformation should
   - ➡ preserve SSA structural constraints
   - ➡ preserve program semantics

Proof framework: big picture



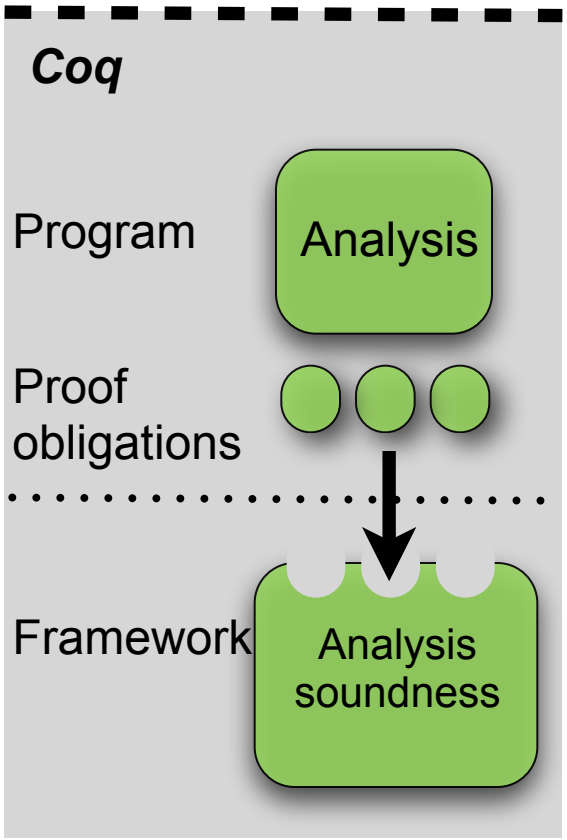
Correctness proof

1. Static analysis should
   ➡ infer sound program invariants
2. Transformation should
   ➡ preserve SSA structural constraints
   ➡ preserve program semantics

Factor out

1. dominance reasoning
2. analysis soundness inv.

Proof framework: focus



Generic analysis:

flow-insensitive

conditional

```
Variable approx: Type.
Variable analysis:
  function → (reg → approx) * exec_flags.
Record AProp := {
     exec:          function → node → Prop;
     G:             regset → approx → val → Prop;
}
```

correct approx

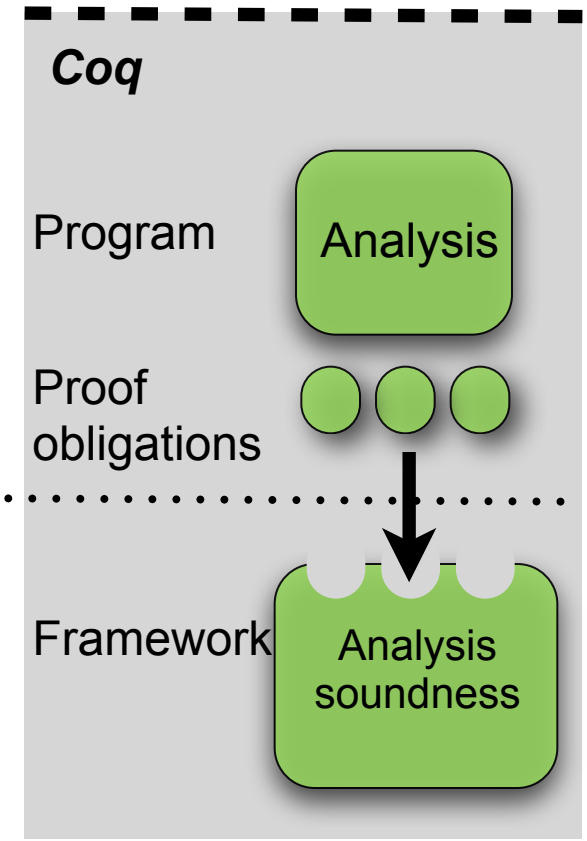
SCCP

```
approx:= ConstLatt
         // traditional constant lattice

forall c, G
forall v, G
```

GVN

```
approx:= reg
         // canon. representative register

forall regs, G regs r (regs r)
```

Proof framework: focus



Generic analysis:

flow-insensitive

conditional

```
Variable approx: Type.
Variable analysis:
  function → (reg → approx) * exec_flags.
Record AProp := {
     exec:        function → node → Prop;
     G:           regset → approx → val → Prop;
}
```

correct approx

Proof obligations:

using Equational Lemma !

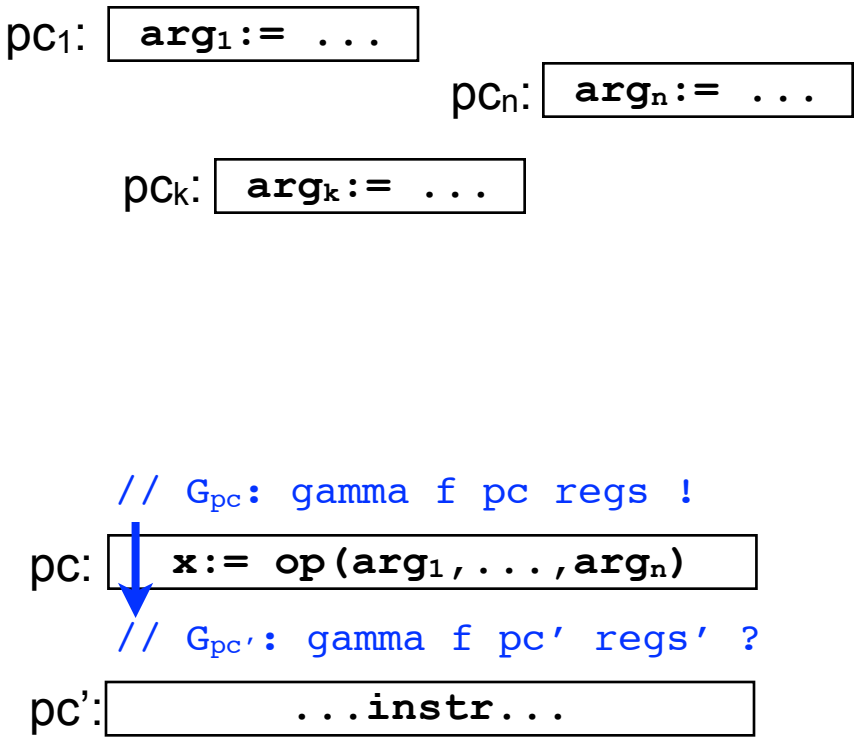
Intra procedural analysis, etc
Local correctness for assignment and phi-functions

Soundness invariant (holds in reachable states):

```
Definition gamma f pc regs : Prop :=
  forall x, dsd f x pc →
            exec f pc →
            G regs ((fst(analysis f)) x)(regs x)
```

pc in dominated region of def(x)

Proof framework: analysis correctness



Use dominated by its definition

1. **Strict SSA (+ dom transitivity):**
   dsd f arg_k pc
2. **G_{pc} hypothesis:**
   correct results for arg_k
3. **Local analysis correctness for op**
4. **At pc':**
   correct result for x
   $y \neq x$: preserved from pc (dsd f y pc')

QED

```
Definition gamma f pc regs : Prop :=
  forall x, dsd f x pc →
            exec f pc →
            G regs ((fst(analysis f)) x)(regs x)
```

Experimental results

Optimization

SCCP/GVN vs. CompCert's CP/LVN

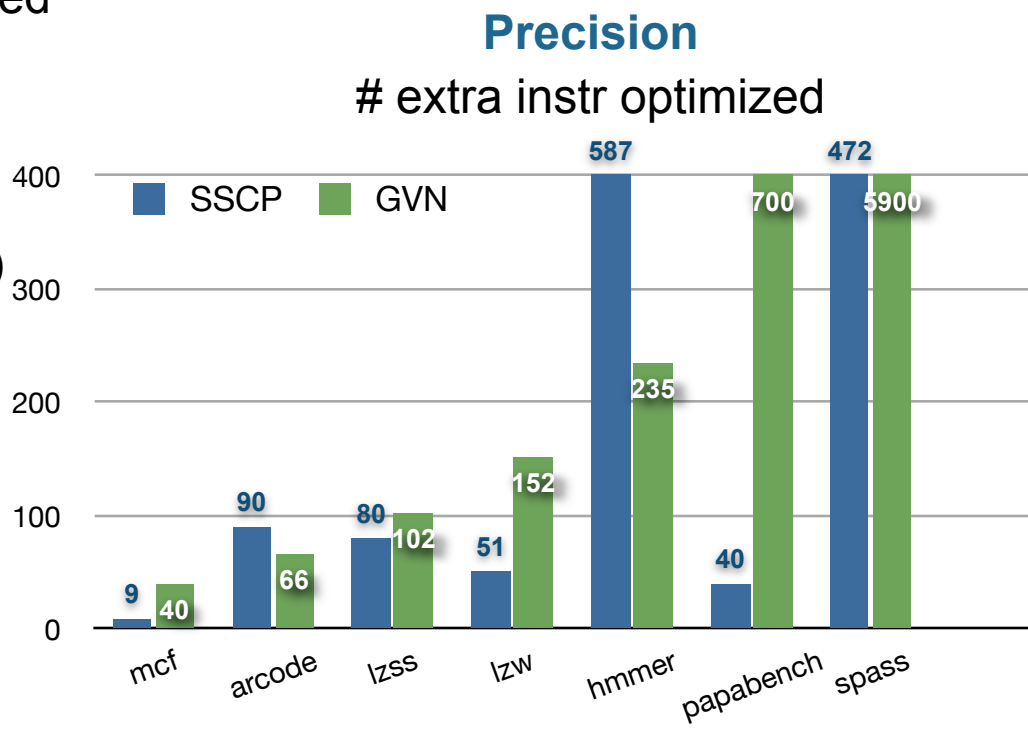
- precision: #extra instr. optimized
- checker efficiency
- transformation time

Benchmarks ~ 130 kLoC code (C)

- compression algorithms
- raytracer, Spass
- some SPEC2006 benchs
- some WCET benchs

Configuration

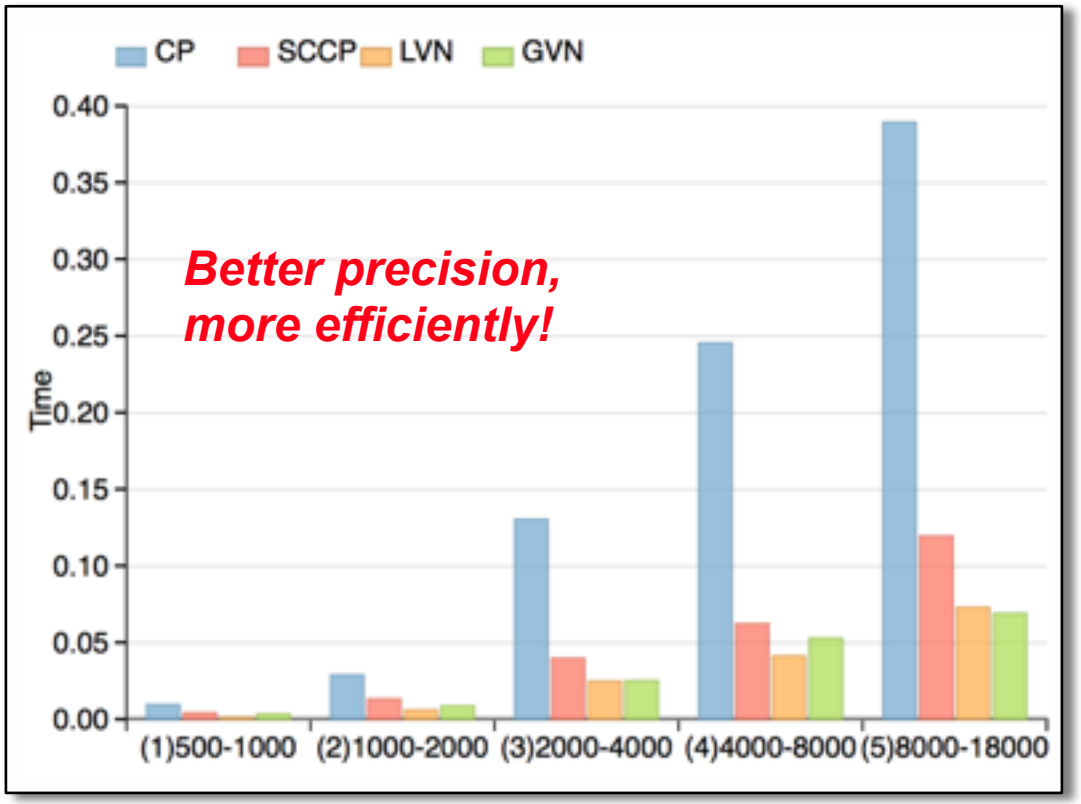
- x86 backend
- full inlining (CFG ≤ 1k nodes)



Optimization time (analysis + transfo)

Absolute time

(in sec., avg. by cat. of CFG size) : lower is better



Checker efficiency

< 20%

of whole optimization phase

SCCP: $O(E_{CFG} + E_{SSA})$
CP: $O(E \times V^2)$
LVN / GVN : linear / linearithmic

SSA destruction

SSA destruction

SSA destruction: elimination of phi-functions

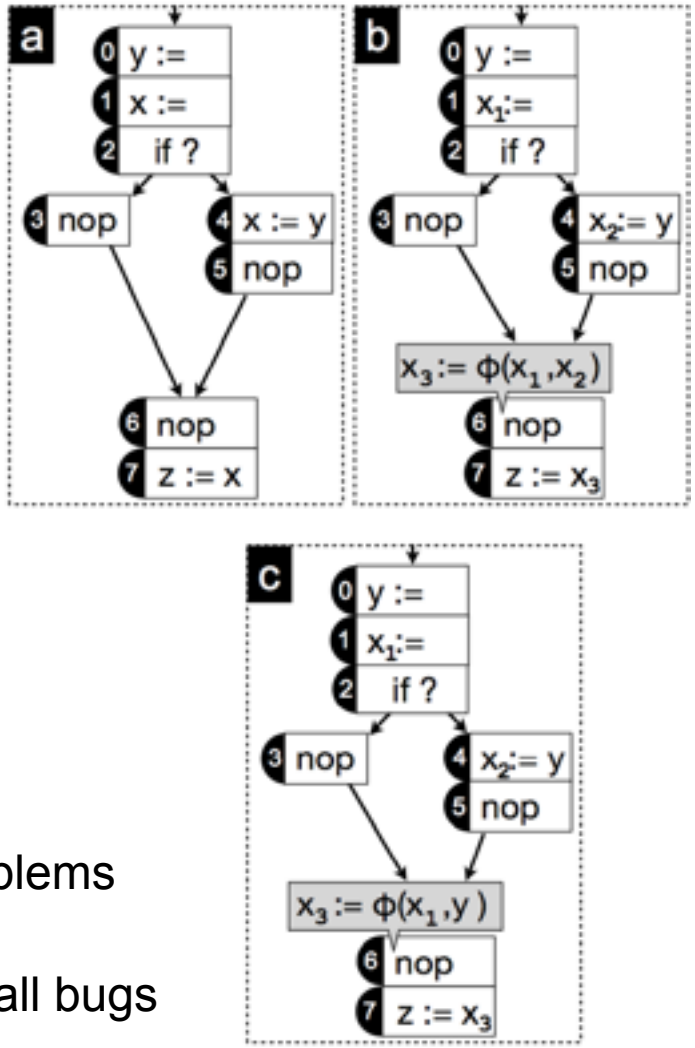
- phi-functions not hardware-supported
- need to come back to a vanilla RTL form

Naive destruction: replace phi-functions by parallel moves at predecessors

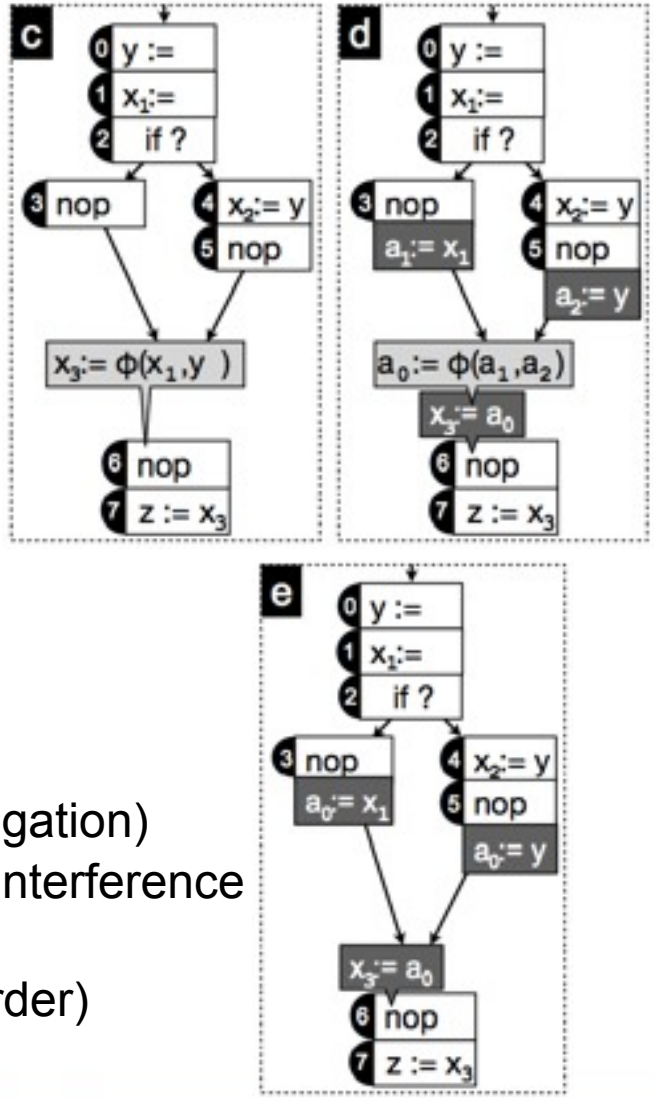
- too many copies: low quality reg. allocation
- not always possible: interferences between variables (after optimizing SSA)

Hard problem even in the non-verified case

- Cytron et al. 91 : first destruction
- Briggs et al. 98 : spot lost-copy, and swap problems
- Sreedhar et al. 99 : conventional SSA (CSSA)
- Boissinot et al. 09 : coalescing on CSSA + small bugs



Conventional SSA and Coalescing



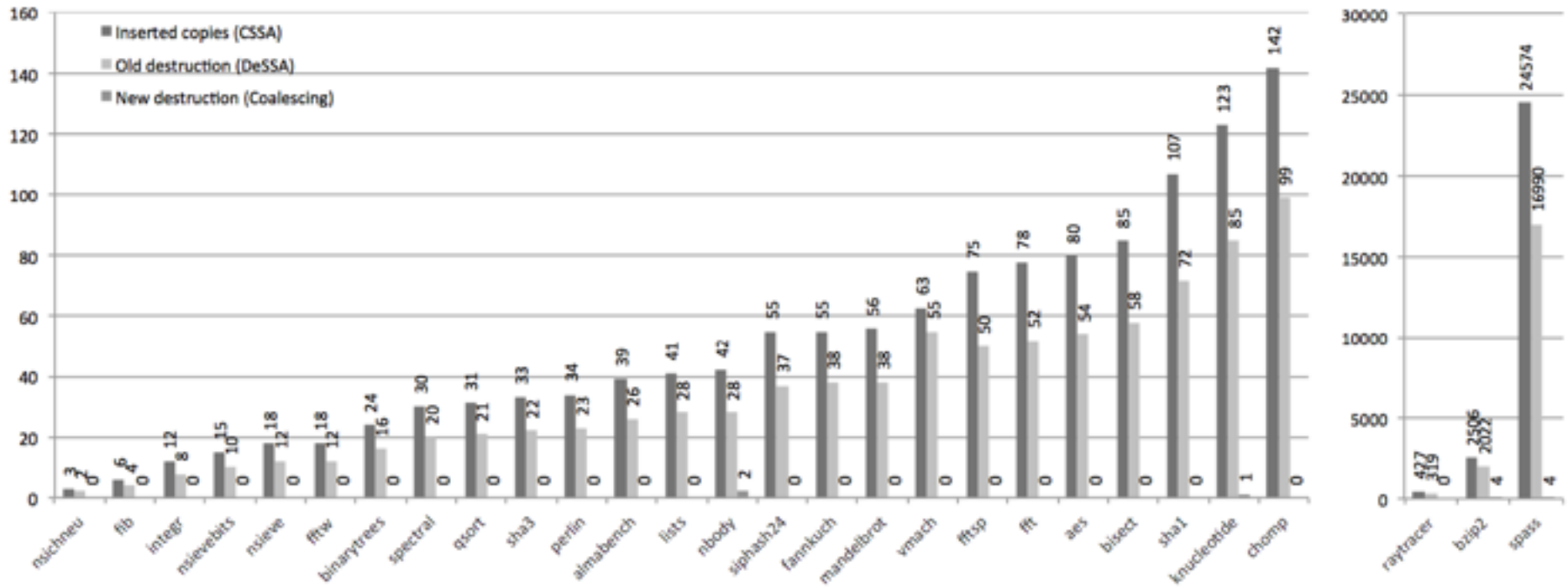
Boissinot et al. approach

- Generate Conventional SSA
- *"Destruction is obviously correct"*
- Perform coalescing (merge variables) to eliminate further copies

Not obvious to Coq...

- Conventional SSA
- Freshness of names is central
- Enriched notion of non-interference:
  - disjoint live-ranges
  - value analysis (here, simple copy propagation)
- Salient point: we must prove the absence of interference between phi-related variables!
  This is a "precision"-like property (usually harder)

Coalescing of variables



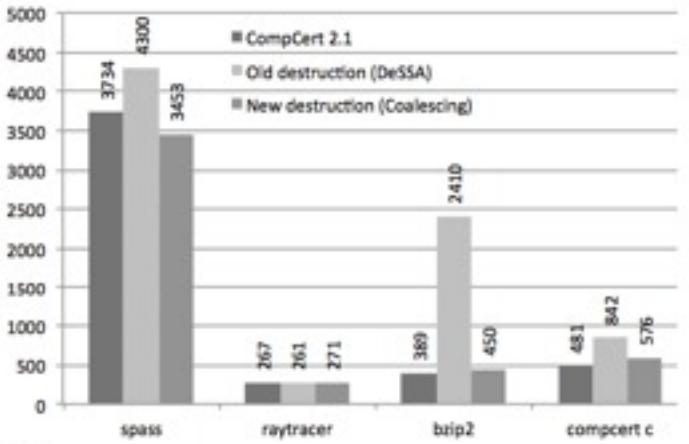
Coalescing of >99% of introduced copies

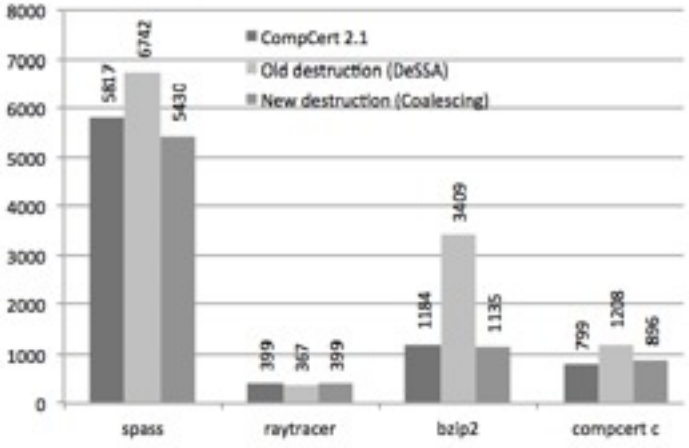
- Current CompCertSSA touch rarely phi-functions arguments
- Remaining copies: for liveness reasons

Impact on the backend

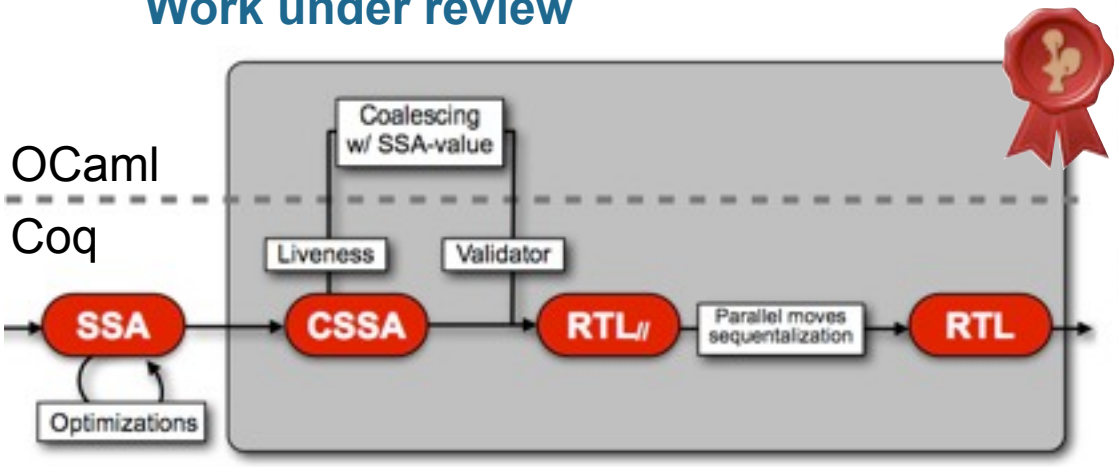
Measure spilling and reloading

- Considerably better than previous destruction
- Closer to CompCert 2.1
- Static measures



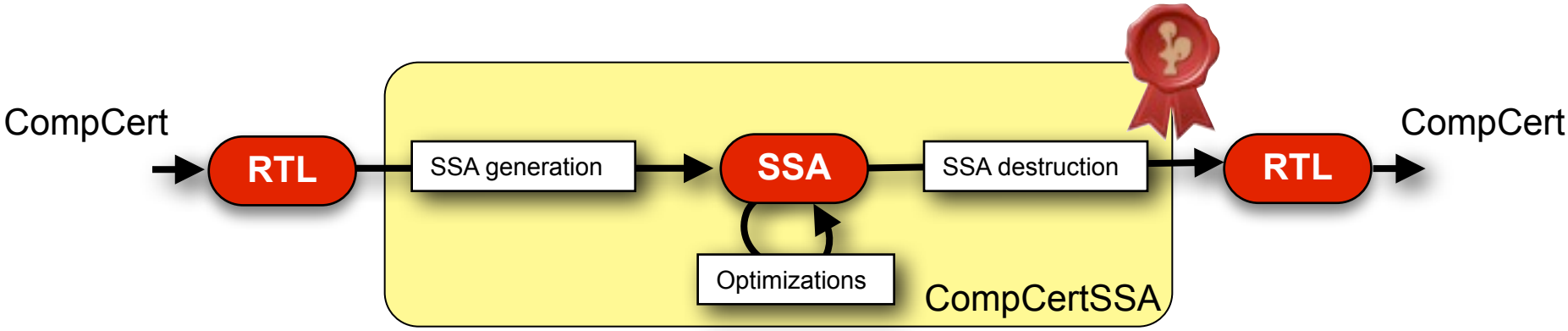


Work under review



Wrap-up

Wrap-up



Realistic use of SSA in a verified compilers

Working operational semantics
Suited, cheap verified validators
State of the art optimizers
Sparse analyses
Dominance reasoning framework
Destruction w/ coalescing

Future plans?

More optims: code motion, memory...
Abstract away from CFG in proofs?
Stay tuned!

http://compcertssa.gforge.inria.fr

Mechanized Verification of SSA-based Compilers Techniques

Delphine Demange - U. Rennes 1 / IRISA / Inria Rennes

Joint work with G. Barthe, S. Blazy, Y. Fernandez de Retana, D. Pichardie, L. Stefanesco

GDR-IM - Villetaneuse - 19/01/16

IRISA