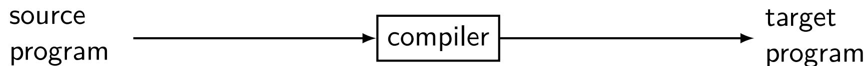
A Fast Verified Liveness Analysis in SSA form

Jean-Christophe Léchenet, Sandrine Blazy, David Pichardie

IJCAR 2020, July 4th, 2020

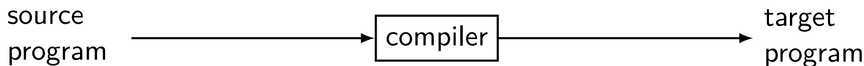
(Non-)Verified Compilation

- Compiler:

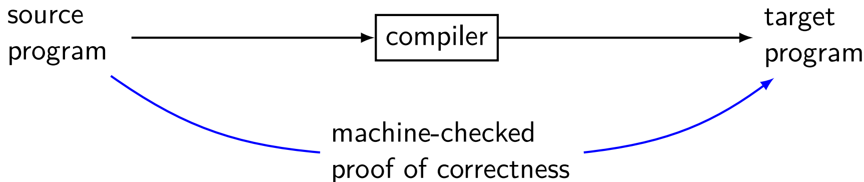


(Non-)Verified Compilation

- Compiler:



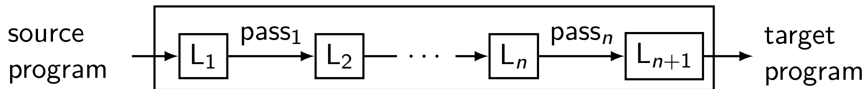
- Formally verified compiler:



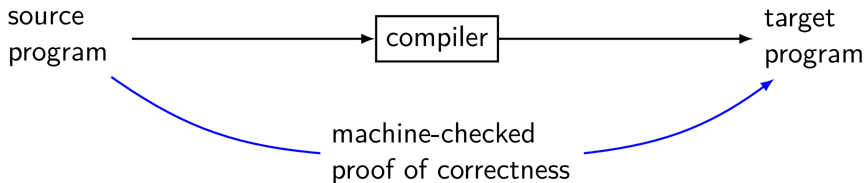
- Examples: CompCert, CakeML, Vellvm

(Non-)Verified Compilation

- Compiler: composition of passes

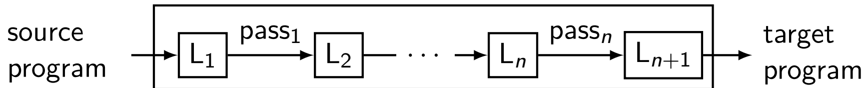


- Formally verified compiler:



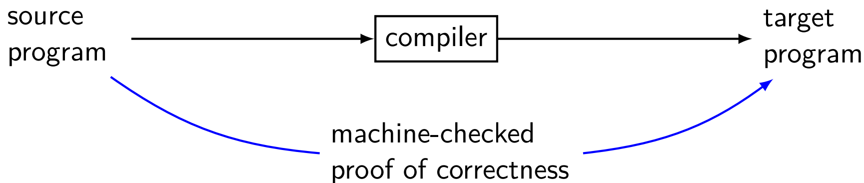
(Non-)Verified Compilation

- Compiler: composition of passes



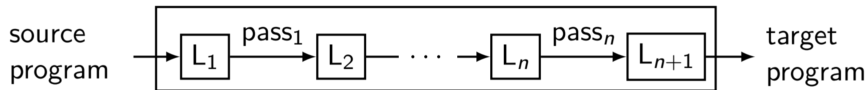
  - Passes may use static analyses

- Formally verified compiler:



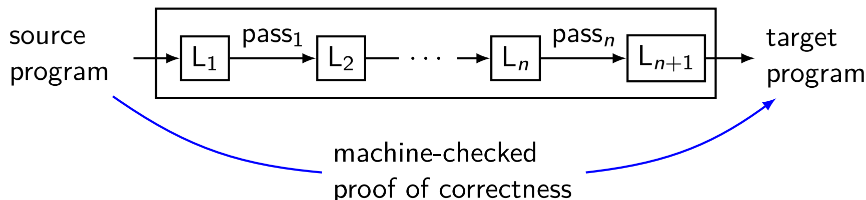
(Non-)Verified Compilation

- Compiler: composition of passes



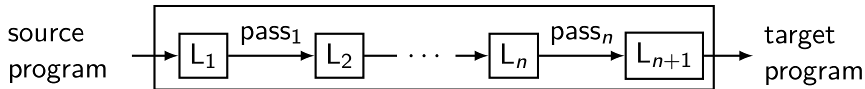
  - Passes may use static analyses

- Formally verified compiler: composition of passes



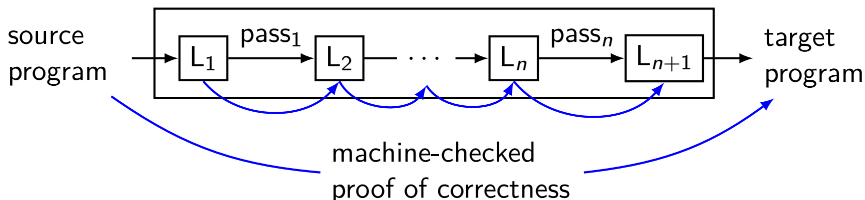
(Non-)Verified Compilation

- Compiler: composition of passes



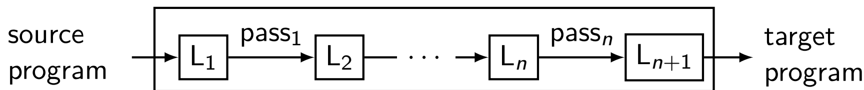
  - Passes may use static analyses

- Formally verified compiler: composition of passes and proofs



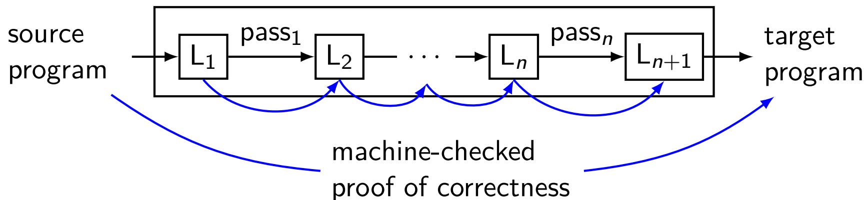
(Non-)Verified Compilation

- Compiler: composition of passes



  - Passes may use static analyses

- Formally verified compiler: composition of passes and proofs



  - Passes may use verified static analyses

Liveness Analysis

- Standard analysis (deadcode elimination, register allocation)
- Live variable: a variable whose value is used in the future

Liveness Analysis

- Standard analysis (deadcode elimination, register allocation)
- Live variable: a variable whose value is used in the future
- Example:

```
1:  x = 2
2:  y = x + 1
3:  x = 0
4:  z = x - 3
5:  y = z
```

Liveness Analysis

- Standard analysis (deadcode elimination, register allocation)
- Live variable: a variable whose value is used in the future
- Example:

```
1:  x = 2
2:  y = x + 1     <---- x live
3:  x = 0
4:  z = x - 3
5:  y = z
```

Liveness Analysis

- Standard analysis (deadcode elimination, register allocation)
- Live variable: a variable whose value is used in the future
- Example:

```
1:  x = 2
2:  y = x + 1
3:  x = 0
4:  z = x - 3
5:  y = z
```

Liveness Analysis

- Standard analysis (deadcode elimination, register allocation)
- Live variable: a variable whose value is used in the future
- Example:

```
1:   x = 2
2:   y = x + 1      <----  x live
3:   x = 0          <----  x dead
4:   z = x - 3      <----  x live
5:   y = z
```

Liveness Analysis

- Standard analysis (deadcode elimination, register allocation)
- Live variable: a variable whose value is used in the future
- Example:

```
1:  x = 2
2:  y = x + 1
3:  x = 0
4:  z = x - 3
5:  y = z
```

- between 1 and 2: x live
- between 2 and 3: x dead
- between 3 and 4: x live
- between 4 and 5: x dead

Verified Liveness Analysis

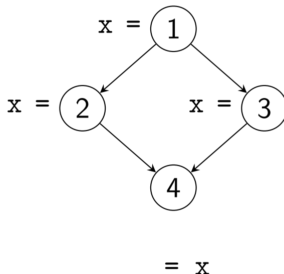
- Classic approach: backward dataflow analysis
- Computes the set of live variables at every program point.
- Examples: CompCert, CakeML

Liveness Checking

- New approach proposed by Boissinot et al. in 2008
- Not a monolithic analysis
- Liveness query : "is variable a live at node q?"
- Fast if few queries
- Applicable only to SSA-form programs
- No verified implementation yet

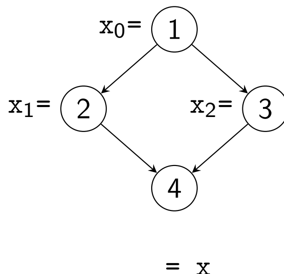
SSA

- Popular compiler intermediate representation
- Allows to write simpler and faster analyses
- Static Single Assignment
  - Each variable has a single textual definition.
  - ϕ-nodes are inserted at junction points.



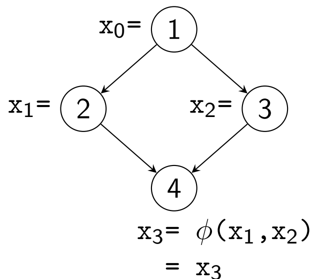
SSA

- Popular compiler intermediate representation
- Allows to write simpler and faster analyses
- Static Single Assignment
  - Each variable has a single textual definition.
  - ϕ-nodes are inserted at junction points.



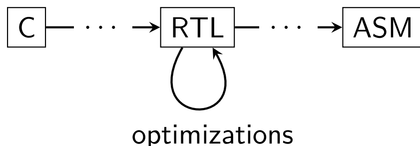
SSA

- Popular compiler intermediate representation
- Allows to write simpler and faster analyses
- Static Single Assignment
  - Each variable has a single textual definition.
  - ϕ-nodes are inserted at junction points.



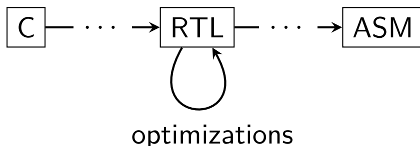
Integration into CompCert?

- Formally verified compiler initiated by Xavier Leroy
- First public release in 2008
- Programmed and proved in Coq
- 11 intermediate languages, 20 passes



Integration into CompCert?

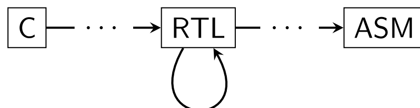
- Formally verified compiler initiated by Xavier Leroy
- First public release in 2008
- Programmed and proved in Coq
- 11 intermediate languages, 20 passes



- No SSA-form!

CompCertSSA

- Fork of CompCert with an SSA middle-end
- Initiated during Delphine Demange's PhD (2012)
- CompCert's compilation chain

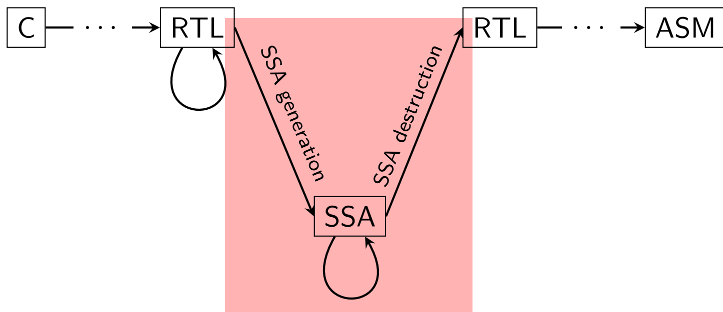


CompCertSSA

- Fork of CompCert with an SSA middle-end
- Initiated during Delphine Demange's PhD (2012)
- CompCertSSA's compilation chain

CompCertSSA

- Fork of CompCert with an SSA middle-end
- Initiated during Delphine Demange's PhD (2012)
- CompCertSSA's compilation chain



Contribution

Formalization of Boissinot et al.'s approach in CompCertSSA

- Implementation and proof of correctness

Outline

Context

Liveness Checking

Formalization in CompCertSSA

Benchmarks

Outline

Context

Liveness Checking

Formalization in CompCertSSA

Benchmarks

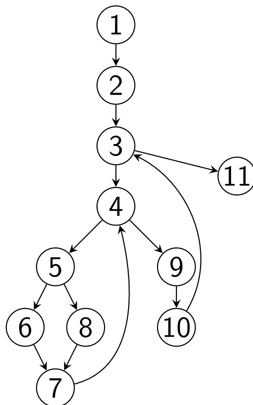
SSA: Dominance Property

- Strict SSA form
  - Each variable is defined before being used.
  - A path from the entry to a use must go through the definition.
- Dominance property: each use is strictly dominated by its definition.

Dominance

Definition (Dominance)

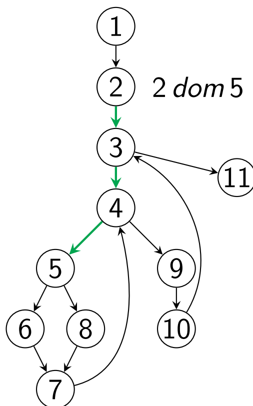
- u dominates v if every path from entry to v contains u.
- u strictly dominates v if u dominates v and $u \neq v$.



Dominance

Definition (Dominance)

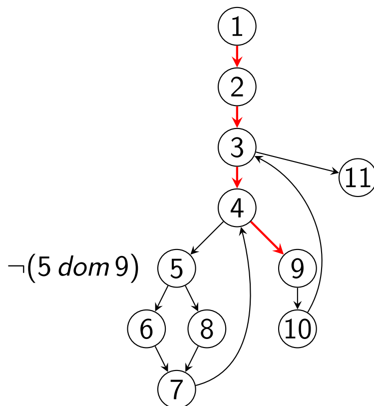
- u dominates v if every path from entry to v contains u.
- u strictly dominates v if u dominates v and $u \neq v$.



Dominance

Definition (Dominance)

- u dominates v if every path from entry to v contains u.
- u strictly dominates v if u dominates v and $u \neq v$.

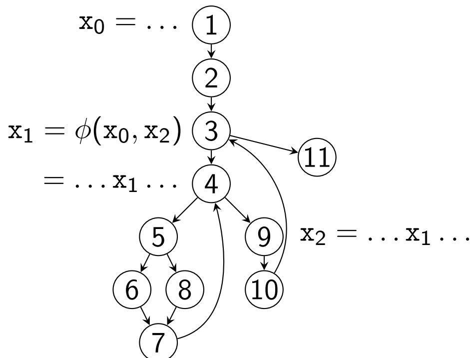


Liveness in SSA

- Formally defined on the Control Flow Graph (CFG)

Definition (Liveness)

a is live at node q if there exists a path from q to a node u where a is used and that path does not contain a definition of a.

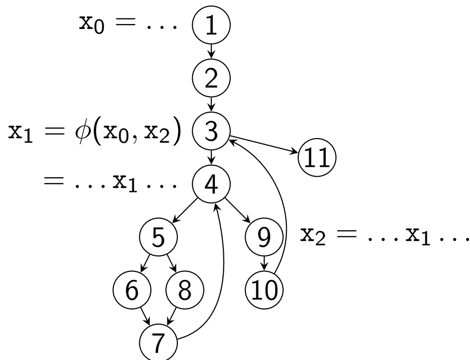


Liveness in SSA

- Formally defined on the Control Flow Graph (CFG)

Definition (Liveness in SSA)

a is live at node *q* if there exists a path from *q* to a node *u* where a is used and that path does not contain the definition *d* of a.

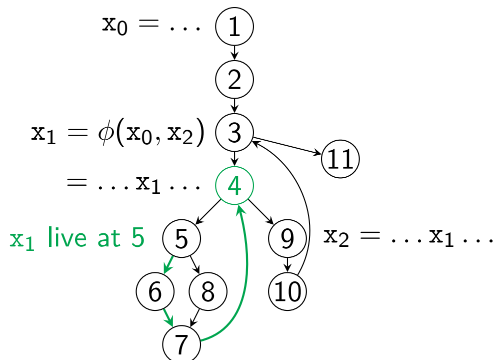


Liveness in SSA

- Formally defined on the Control Flow Graph (CFG)

Definition (Liveness in SSA)

a is live at node *q* if there exists a path from *q* to a node *u* where a is used and that path does not contain the definition *d* of a.

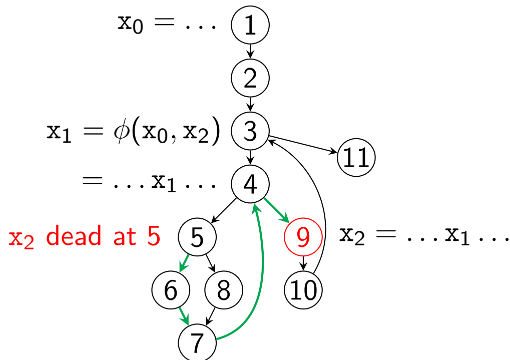


Liveness in SSA

- Formally defined on the Control Flow Graph (CFG)

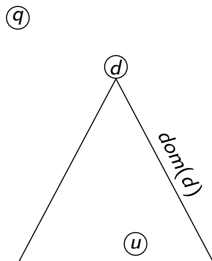
Definition (Liveness in SSA)

a is live at node q if there exists a path from q to a node u where a is used and that path does not contain the definition d of a.



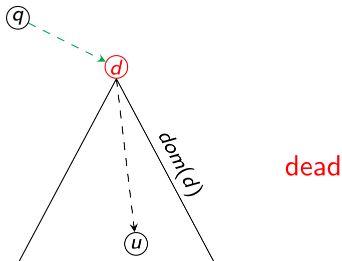
Liveness Checking: Intuition

- Assume that the CFG is acyclic.
- If q is not strictly dominated by d:



Liveness Checking: Intuition

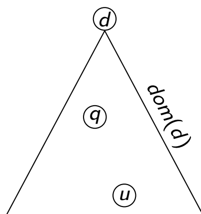
- Assume that the CFG is acyclic.
- If q is not strictly dominated by d:



dead

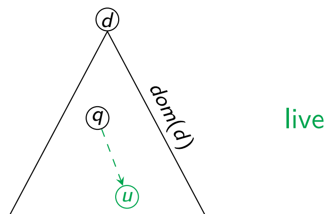
Liveness Checking: Intuition

- Assume that the CFG is acyclic.
- If q is strictly dominated by d:



Liveness Checking: Intuition

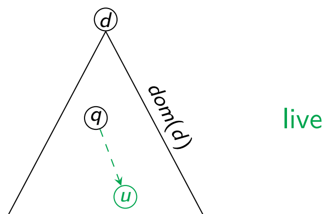
- Assume that the CFG is acyclic.
- If q is strictly dominated by d:



live

Liveness Checking: Intuition

- Assume that the CFG is acyclic.
- If q is strictly dominated by d:



live

- The problem is simple in the absence of back edges.

Liveness checking

2 parts:

- Precomputation (monolithic)
  - Depends only on the graph structure of the CFG.
  - More precisely, on the loop structure
- Online
  - Uses the precomputation to answer the queries efficiently.

Precomputation

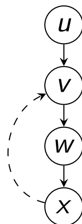
Definition (Reduced graph)

The reduced graph designates the CFG without the back edges.

Two relations:

- R_v: the set of nodes reduced-reachable from v
- T_v: the reflexive and transitive closure of $T_v^{\uparrow}$
  - $T_v^{\uparrow}$: the set of back-edge targets not reduced-reachable from v but whose sources are reduced-reachable from v.

$$T_v^{\uparrow} = \{t \in V \setminus R_v \mid \exists s \in R_v, (s, t) \text{ is a back edge}\}$$

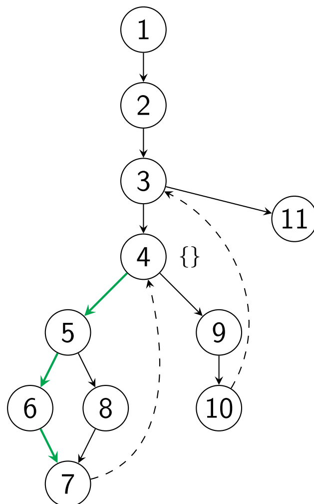


$$T_u^{\uparrow} = T_v^{\uparrow} = \emptyset$$

$$T_w^{\uparrow} = T_x^{\uparrow} = \{v\}$$

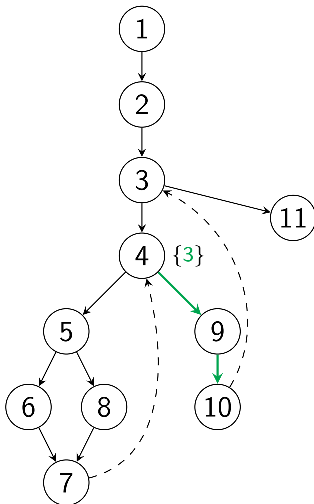
Precomputation ($T^{\uparrow}$)

$$T_v^{\uparrow} = \{t \in V \setminus R_v \mid \exists s \in R_v, (s, t) \text{ is a back edge}\}$$



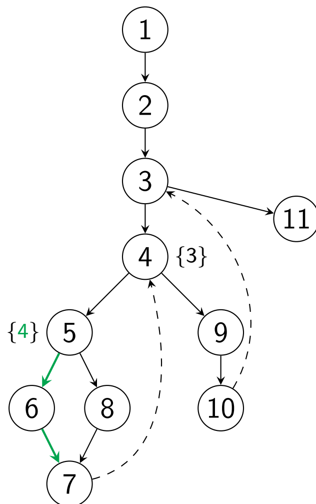
Precomputation ($T^{\uparrow}$)

$T_v^{\uparrow} = \{t \in V \setminus R_v \mid \exists s \in R_v, (s,t) \text{ is a back edge}\}$



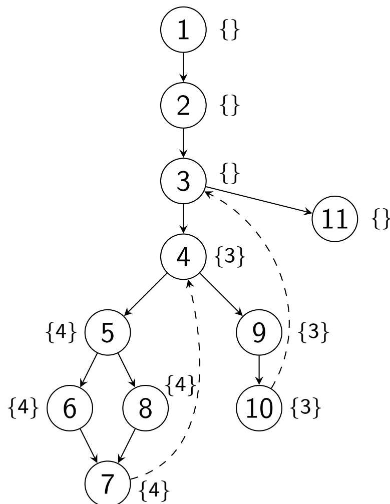
Precomputation ($T^{\uparrow}$)

$T^{\uparrow}_v = \{t \in V \setminus R_v \mid \exists s \in R_v, (s,t) \text{ is a back edge}\}$



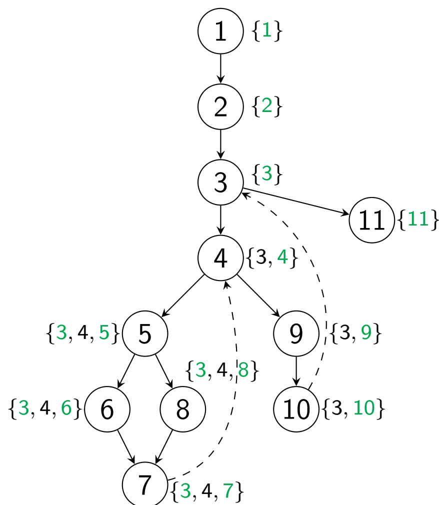
Precomputation ($T^{\uparrow}$)

$T^{\uparrow}_v = \{t \in V \setminus R_v \mid \exists s \in R_v, (s, t) \text{ is a back edge}\}$



Precomputation (T)

$T = (T^{\uparrow})^*$



Online

```
Function IsLiveIn(var a, node q):
    T_(q,a) ← T_q ∩ sdom(def(a))
    foreach t ∈ T_(q,a) do
        if R_t ∩ uses(a) ≠ ∅ then
            return true
    end
    return false
end
```

Function $\textsc{IsLiveIn}$(var a, node q):

$T_{(q,\mathtt{a})} \leftarrow T_q \cap sdom(def(\mathtt{a}))$

foreach $t \in T_{(q,\mathtt{a})}$ **do**

  if $R_t \cap uses(\mathtt{a}) \neq \emptyset$ **then**

    return *true*

end

return *false*

end

Online

Function IsLiveIn(var a, node q):

  $T_{(q,\mathtt{a})} \leftarrow T_q \cap sdom(def(\mathtt{a}))$

  foreach $t \in T_{(q,\mathtt{a})}$ **do**

    if $R_t \cap uses(\mathtt{a}) \neq \emptyset$ **then**

      return *true*

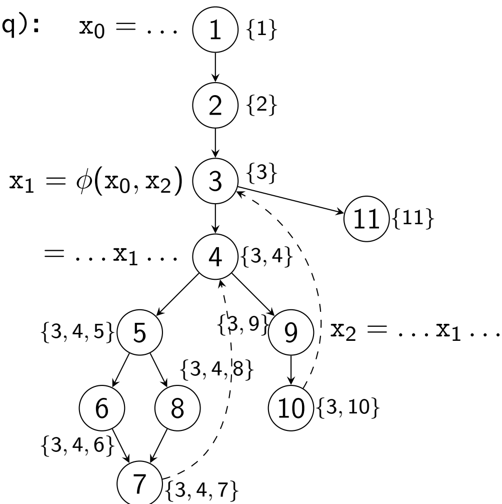
  end

  return *false*

end

Example:

- $q = 5$, $\mathtt{a} = \mathtt{x}_0$
- $T_{(q,\mathtt{a})} = \{3,4,5\} \cap sdom(1)$
  $= \{3,4,5\}$
- $uses(\mathtt{x}_0) = \{2\}$



Online

Function IsLiveIn(var a, node q):

$T_{(q,\mathtt{a})} \leftarrow T_q \cap sdom(def(\mathtt{a}))$

foreach $t \in T_{(q,\mathtt{a})}$ **do**

  if $R_t \cap uses(\mathtt{a}) \neq \emptyset$ **then**

    return *true*

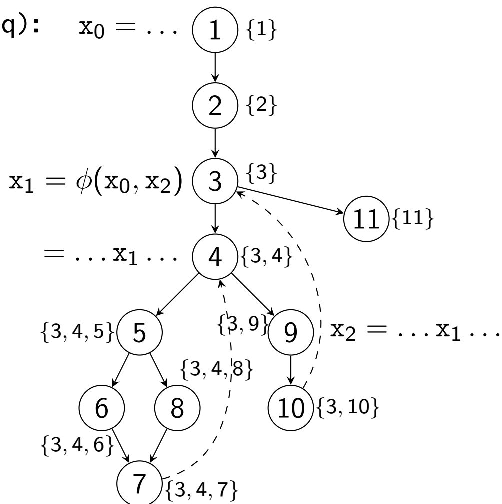
end

return *false*

end

Example:

- $q = 5$, $\mathtt{a} = \mathtt{x}_0$
- $T_{(q,\mathtt{a})} = \{3,4,5\} \cap sdom(1) = \{3,4,5\}$
- $uses(\mathtt{x}_0) = \{2\}$
- dead



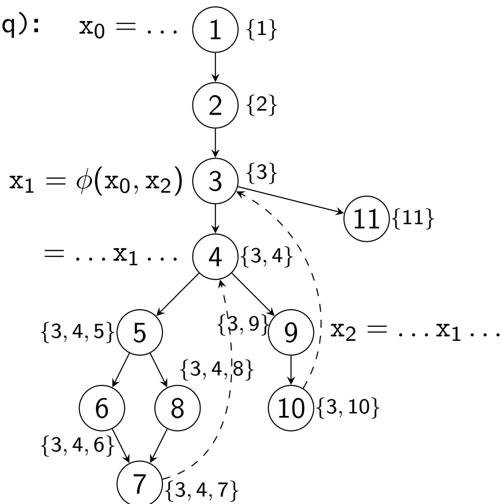
Online

Function IsLiveIn(var a, node q):
- $T_{(q,\mathtt{a})} \leftarrow T_q \cap sdom(def(\mathtt{a}))$
- **foreach** $t \in T_{(q,\mathtt{a})}$ **do**
  - **if** $R_t \cap uses(\mathtt{a}) \neq \emptyset$ **then**
    - **return** *true*
- **end**
- **return** *false*

end

Example:

- $q = 5$, $\mathtt{a} = \mathtt{x}_1$
- $T_{(q,\mathtt{a})} = \{3,4,5\} \cap sdom(3) = \{4,5\}$
- $uses(\mathtt{x}_1) = \{4,9\}$

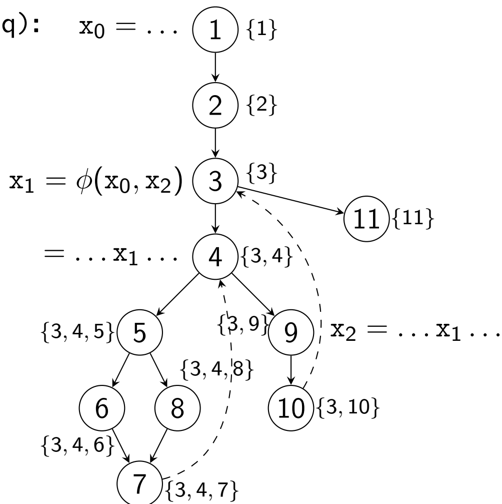


Online

Function IsLiveIn(var a, node q):
  $T_{(q,\mathtt{a})} \leftarrow T_q \cap sdom(def(\mathtt{a}))$
  foreach $t \in T_{(q,\mathtt{a})}$ **do**
    if $R_t \cap uses(\mathtt{a}) \neq \emptyset$ **then**
      return *true*
  end
  return *false*
end

Example:

- $q = 5$, $\mathtt{a} = \mathtt{x}_1$
- $T_{(q,\mathtt{a})} = \{3, 4, 5\} \cap sdom(3)$
  $= \{4, 5\}$
- $uses(\mathtt{x}_1) = \{4, 9\}$
- live



Online

Function IsLiveIn(var a, node q):

$T_{(q,\mathtt{a})} \leftarrow T_q \cap sdom(def(\mathtt{a}))$

foreach $t \in T_{(q,\mathtt{a})}$ **do**

  if $R_t \cap uses(\mathtt{a}) \neq \emptyset$ **then**

    return *true*

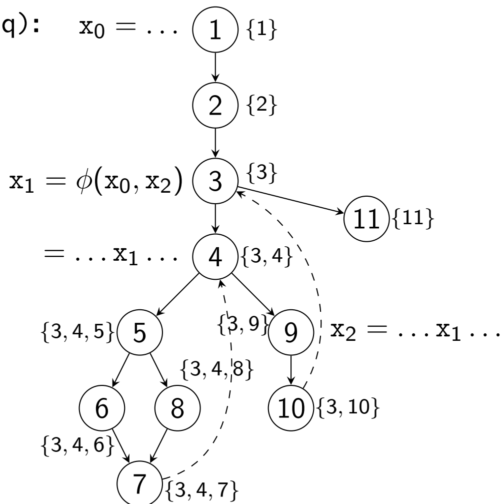
end

return *false*

end

Example:

- $q = 5$, $\mathtt{a} = \mathtt{x}_2$
- $T_{(q,\mathtt{a})} = \{3, 4, 5\} \cap sdom(9) = \emptyset$
- dead



Outline

Context

Liveness Checking

Formalization in CompCertSSA

Benchmarks

Precomputation

Two steps:

- R and $T^{\uparrow}$ in one pass
- T computed from $T^{\uparrow}$

Precomputation: first part

- R and $T^{\uparrow}$ computed in one DFS
- Difference with Boissinot et al.: storage of R
  - Boissinot et al. store R explicitly and suggest using bitsets
  - CFGs in CompCert do not have blocks, thus are bigger
- Our choice: decompose R into two relations R and C
  - R: reachability in the spanning tree (2 integers per node)
  - C: interesting cross-edge targets ($\approx T$ for cross edges)

Precomputation: first part

DFS inspired from Postorder.v

```
Record state : Type := mkstate {
  gr: graph;                      (* current graph, without already-visited nodes *)
  wrk: list (node * list node); (* worklist *)
  next: positive                  (* number to use for next numbering *)
  m: map positive;                (* mapping node → postorder number *)
}.

Definition transition (s: state) : map positive + state := ...

Definition postorder (g: graph) (root: node) :=
  WfIter.iterate _ _ transition
    lt_state lt_state_wf transition_decreases
    (init_state g root).
```

Precomputation: first part

DFS inspired from Postorder.v

```
Record state := {
  gr: graph;                          (* current graph, without already-visited nodes *)
  wrk: list (node * positive * list node * (set * list node));
                                      (* worklist *)
  next: positive;                     (* number to use for next numbering *)
  r : map itv;                        (* R *)
  c : map set;                        (* C *)
  t_up : map (list node);             (* T↑ *)
  back : list (node * node)           (* list of back edges *)
}.

(* result: type of the returned tuple (r, c, t_up, back) *)
Definition result := map itv * map set * map (list node) * list(node * node).
Definition transition (s: state) : result + state := ...

Definition precompute_r_t_up (g: graph) (root: node) : result :=
  WfIter.iterate _ _ transition
    lt_state lt_state_wf transition_decreases
    (init_state g root).
```

Precomputation: first part

```
Inductive invariant (s: state) : Prop := ...

Inductive postcondition (res : result) : Prop := ...

Lemma transition_spec:
  forall s, invariant s →
  match transition s with
  | inr s' ⇒ invariant s'
  | inl res ⇒ postcondition res end.
Proof. ...  Qed.
```

Precomputation: second part

- T is computed from $T^{\uparrow}$ in two steps.
  - Back-edge targets: in DFS preorder
  - Other nodes: in any order
- Proof for back-edge targets is non-trivial.
  - `fold_left` on the sorted list of back edges
  - Subtle invariant
  - We reused the structure of Postorder.v, but only for the proof.

Precomputation: theorems

```
Lemma precompute_r_c_correct : forall g root,
  let '(r, c, t, back) := precompute g root in
  forall u v,
    r_c_linked r c u v
  <->
    (reachable g root u /\ g ! u <> None
    /\ reduced_reachable g back u v).
```

Precomputation: theorems

```
Definition in_t_up g back u v :=
  ~ reduced_reachable g back u v (* v is not reachable *)
  ∧ ∃ s, In (s, v) back (* v is the target of a back-edge *)
    ∧ reduced_reachable g back u s.
      (* whose source is reduced-reachable *)

Lemma precompute_t_correct : forall g root,
  let '(r, c, t, back) := precompute g root in
  forall u v,
    t_linked t u v
    ↔ (reachable g root u
         ∧ clos_refl_trans _ (in_t_up g back) u v).
```

Online part

- Relatively straightforward
- Boissinot et al.'s algorithm in functional style
- Proofs similar to the original ones
- One exception:
  - Original proof: "let us consider a path with a minimal number of back-edges"
  - Coq proof: induction on the path

Online part: theorem

```
Theorem analyze_correct :
  forall (f : function), wf_ssa_function f →
  let live := analyze f in
  forall a q, live a q = true ↔ live_spec f a q.
```

Outline

Context

Liveness Checking

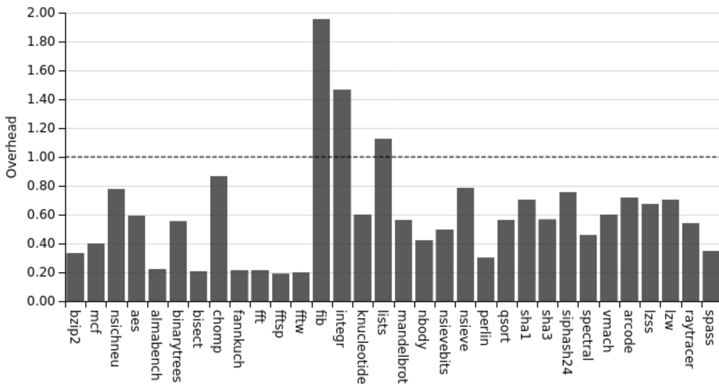
Formalization in CompCertSSA

Benchmarks

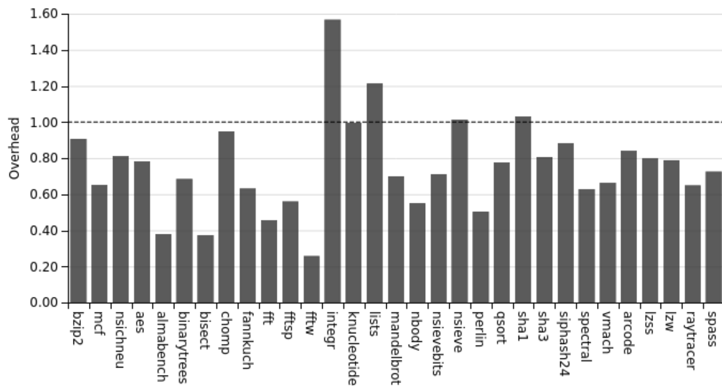
Benchmark

- Comparison of our analysis w.r.t. a dataflow-based analysis
- Dataflow-based analysis based on module Kildall of CompCert
- Artificial criterion: one query per variable and per loop header

Benchmarks: precomputation



Benchmarks: total



Conclusion

- Formalization in CompCertSSA of “liveness checking”
  - Non-trivial justification
    - Dominance property of SSA
    - Graph theoretic arguments: DFS, dominance
- A second implementation not presented
  - It can skip nodes by using dominance more.
- Proof effort
  - Precomputation: 1700 lines of specification, 4000 lines of proof
  - Online part: 230 lines of specification, 1000 lines of proof

Future work

- Use the analysis in a pass of CompCertSSA
  - e.g. SSA destruction
- Completeness of the dominance test is not proved
  - Formalization based on Georgiadis and Tarjan, Dominator tree certification and divergent spanning trees, 2015 (Unpublished)