Mechanized Verification of Computing Dominators for Formalizing Compilers

Jianzhou Zhao and Steve Zdancewic

University of Pennsylvania
{jianzhou,stevez}@cis.upenn.edu

Abstract. One prerequisite to the formal verification of modern compilers is to formalize computing dominators, which enable SSA forms, advanced optimizations, and analysis. This paper provides an abstract specification of dominance analysis that is sufficient for formalizing modern compilers; it describes a certified implementation and instance of the specification that is simple to design and reason about, and also reasonably efficient. The paper also presents applications of dominance analysis: an SSA-form type checker, verifying SSA-based optimizations, and constructing dominator trees. This development is a part of the Vellvm project. All proofs and implementation have been carried out in Coq.

1 Introduction

Compilers are not always correct due to the complexity of language semantics and transformation algorithms, the trade-offs between compilation speed and verifiability, *etc.* Bugs in compilers can undermine the source-level verification efforts (such as type systems, static analysis, and formal proofs), and produce target programs with different meaning from source programs. The CompCert project [12] first implemented a realistic and mechanically verified compiler with classic intermediate representations in the Coq proof assistant. The CompCert compiler generates compact and efficient assembly code for a large fragment of the C language, and is proved to be more robust than non-verified compilers.

Recently researchers started to formalize and verify modern compilers in the Vellvm project [14] and in the CompCertSSA project [3]. One crucial component of modern compilers, such as LLVM and GCC, is computing *dominators*—on a control-follow-graph, a node l_1 dominates a node l_2 if all paths from the entry to l_2 must go through l_1 [2]. Dominance analysis allows compilers to represent programs in the SSA form [6] (which enables many advanced SSA-based optimizations), optimize loops, analyze memory dependency, and parallelize code automatically, *etc.* Therefore, one prerequisite to the formal verification of modern compilers is to formalize computing dominators.

In this paper, we present the formalization of dominance analysis used in the Vellvm project. To the best of our knowledge, this is the first mechanized verification of dominator computation for LLVM. Although the CompCertSSA project [3] also formalized dominance analysis to prove the correctness of a global value numbering optimization, our results are more general: beyond soundness, we establish completeness and related metatheory results that can be used in

other applications. Because different styles of formalization may also affect the cost of proof engineering, we also discuss some tradeoffs in the choices of formalization. In this work, we evaluate our formalism by applying it to several applications in Vellvm.

To simplify the formal development, we describe the work in the context of Vminus, which is a simpler subset of the full LLVM SSA IR formalized in Vellvm, that still captures the essence of dominance analysis. Our Coq development formalizes all the claims of the paper for the full Vellvm[1]. Following LLVM, we distinguish dominators at the block level and at the instruction level. Given the former one, we can easily compute the latter one. Therefore, we will focus on the block-level analysis, and discuss the instruction-level analysis only briefly.

We present the following contributions. Section 2 gives a specification of computing dominators at the block level. We instantiate the specification by two algorithms. Section 3 shows the standard dominance analysis [1] (AC). Section 4 presents an extension of AC [5] (CHK) that is easy to implement and verify, but still fast. We verify the correctness of both algorithms. Section 3.1 provides a verified depth first search algorithm. Then, Section 5 extends the dominance analysis to the instruction level, and present several applications used in the Vellvm project: a type checker for SSA, verifying SSA-based optimizations, and constructing dominator trees. Section 6 evaluates performance of the algorithms, and shows that in practice CHK runs nearly as fast as the LLVM's algorithm.

2 The Specification of Computing Dominators

This section first defines dominators in term of the syntax of Vminus, then gives an abstract and succinct specification of algorithms that compute dominators.

Syntax of Vminus. Figure 1 gives the syntax of Vminus, focusing on the syntax of Vminus at the block level. Section 5 will revisit the rest of the syntax. All code in Vminus resides in top-level functions, whose bodies are composed of blocks b. Here, $\overline{b}$ denotes a list of blocks; we use similar notation for other lists throughout the paper. As in classic compiler representations, a basic block consists of a label l, a series of instructions *insn* followed by a terminator *tmn* (**br** and **ret**) that branches to another block or returns from the function. In the following, we also use the label of a block to denote the block.

The set of blocks making up the top-level function f constitutes a control-flow graph (CFG) $G = (e, succs)$ where e is the entry point (the first block) of f; *succs* maps each label to a list of its successors. On a CFG, we use $G \models l_1 \rightarrow^* l_2$ to denote a path ρ from l_1 to l_2, and $l \in \rho$ to denote that l is in the path ρ. By `wf f`, we require that a well-formed function must contain an entry point that cannot be reached from other blocks, all terminators can only branch to blocks within f, and that all labels in f are unique. In this paper, we consider only well-formed functions to streamline the presentation.

[1] Available at `http://www.cis.upenn.edu/~jianzhou/Vellvm/dominance`

Types	$typ ::= \mathbf{int}$	Instructions	$insn ::= \phi \mid c \mid tmn$
Constants	$cnst ::= Int$	Phi Nodes	$\phi ::= r = \mathbf{phi}\ typ\ \overline{[val_j, l_j]}^{\,j}$
Values	$val ::= r \mid cnst$	Commands	$c ::= r := val_1\ bop\ val_2$
Blocks	$b ::= l\ \overline{\phi}\ \overline{c}\ tmn$	Terminators	$tmn ::= \mathbf{br}\ l \mid \mathbf{br}\ val\ l_1\ l_2 \mid \mathbf{ret}\ typ\ val$
Functions	$f ::= \mathbf{fun}\ \{\overline{b}\}$	Non-ϕs	$\hat{\phi} ::= c \mid tmn$

Fig. 1. Syntax of Vminus.

Definition 1 (Domination (Block-level)). *Given G with an entry e,*

- *A block l is **reachable**, written $G \rightarrow^* l$, if there exists a path $G \models e \rightarrow^* l$.*
- *A block l_1 **dominates** a block l_2, written $G \models l_1 \gg= l_2$, if for every path ρ from e to l_2, $l_1 \in \rho$.*
- *A block l_1 **strictly dominates** a block l_2, written $G \models l_1 \gg l_2$, if for every path ρ from e to l_2, $l_1 \neq l_2 \wedge l_1 \in \rho$.*

Because the dominance relations of a function at the block level and in its CFG are equivalent, in the following we do not distinguish f and G. The following consequence of the definitions are useful to define the specification of computing dominators. For all labels in G, $\gg=$ and $\gg$ are transitive.

Lemma 1.
- If $G \models l_1 \gg= l_2$ and $G \models l_2 \gg= l_3$, then $G \models l_1 \gg= l_3$.
- If $G \models l_1 \gg l_2$ and $G \models l_2 \gg l_3$, then $G \models l_1 \gg l_3$.

However, because there is no path from the entry to unreachable labels, $\gg=$ and $\gg$ relate every label to any unreachable labels.

Lemma 2. *If $\neg(G \rightarrow^* l_2)$, then $G \models l_1 \gg= l_2$ and $G \models l_1 \gg l_2$.*

If we only consider the reachable labels in V, $\gg$ is acyclic.

Lemma 3 ($\gg$ is acyclic). *If $G \rightarrow^* l$, then $\neg G \models l \gg l$.*

Moreover, all labels that strictly dominate a reachable label are ordered.

Lemma 4 ($\gg$ is ordered). *If $G \rightarrow^* l_3$, $l_1 \neq l_2$, $G \models l_1 \gg l_3$ and $G \models l_2 \gg l_3$, then $G \models l_1 \gg l_2 \vee G \models l_2 \gg l_1$.*

2.1 Specification

Coq Notations. We use {} to denote an empty set; use {`+`}, {`<=`}, `` `In` ``, {`\/`} and {`/\`} to denote set addition, inclusion, membership, union and intersection respectively. Our developments reuse the basic tree and map data structures implemented in the CompCert project [12]: `ATree.t` and `PTree.t` are trees with keys of type l and positive respectively; `PMap.t` is a map with keys of type positive. We use `[]` to denote tree and map lookup. *succs* are defined by trees. `[]` returns an empty list when a searched-for key in *succs* does not exist. `[x]` is a list with one element `x`.

```
Module Type ALGDOM.
  Parameter sdom: f →l →set l.
  Definition dom f l1 := l1 {+} sdom f l1.
  Axiom entry_sound: forall f e, entry f = Some e →sdom f e = {}.
  Axiom successors_sound: forall f l1 l2,
    In l1 (succs f)[l2] →sdom f l1 {<=} dom f l2.
  Axiom complete: forall f l1 l2,
    wf f →f |= l1 >> l2 →l1 `In` (sdom f l2).
End ALGDOM.
Module AlgDom_Properties(AD: ALGDOM).
  Lemma sound: forall f l1 l2,
    wf f →l1 `In` (AD.sdom f l2) →f |= l1 >> l2.
  (*************************************************************************)
  (* Properties: conversion, transitivity, acyclicity, ordering and ... *)
  (*************************************************************************)
End AlgDom_Properties.
```

Fig. 2. The specification of algorithms that find dominators.

Figure 2 gives an abstract specification of algorithms that compute dominators using a Coq module interface `ALGDOM`. First of all, `sdom` defines the signature of a dominance analysis algorithm: given a function f and a label l_1, (`sdom` f l_1) returns the set of strict dominators of l_1 in f; `dom` defines the set of dominators of l_1 by adding l_1 into l_1's strict dominators.

To make the interface simple, `ALGDOM` only requires the basic properties that ensure that `sdom` is correct: it must be both `sound` and `complete` in terms of the declarative definitions (Definition 1). Given the correctness of `sdom`, the `AlgDom_Properties` module can 'lift' properties (conversion, transitivity, acyclicity, ordering, *etc.*) from the declarative definitions to the implementations of `sdom` and `dom`. Section 5 shows how clients of `ALGDOM` use the properties proven in `AlgDom_Properties` by examples.

`ALGDOM` requires completeness directly. Soundness can be proven by two more basic properties: `entry_sound` requires that the entry has no strict dominators; `successors_sound` requires that if l_1 is a successor of l_2, then l_2's dominators must include l_1's strict dominators. Given an algorithm that establishes the two properties, `AlgDom_Properties` proves that the algorithm is sound by induction over any path from the entry to l_2.

2.2 Instantiations

In the literature, there is a long history of algorithms that find dominators, each making different trade-offs between efficiency and simplicity. Most of the industry compilers, such as LLVM and GCC, use the classic Lengauer-Tarjan algorithm [11] (LT) that has a complexity of $O(E * log(N))$ where N and E are the number of nodes and edges respectively, but is complicated to implement and reason about. The Allen-Cocke algorithm [1] (AC) based on iteration is easier to design, but suffers from a large asymptotic complexity. Moreover, LT

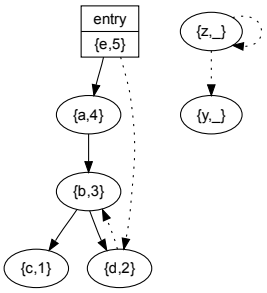

Fig. 3. Postorder.

stk	visited	PO_l2p po
e[a d]	e	
e[d]; a[b]	e a	
e[d]; a[]; b[c d]	e a b	
e[d]; a[]; b[d]; c[]	e a b c	(c,1)
e[d]; a[]; b[]; d[b]	e a b c d	(c,1)
e[d]; a[]; b[]; d[]	e a b c d	(c,1); (d,2)
e[d]; a[]; b[];	e a b c d	(c,1); (d,2); (b,3)
e[d]; a[];	e a b c d	(c,1); (d,2); (b,3); (a,4)
e[]	e a b c d	(c,1); (d,2); (b,3); (a,4); (e,5)

Fig. 4. The DFS execution sequence.

explictly creates dominator trees that provide convenient data structures for compilers whereas AC needs an additional tree construction algorithm with more overhead. The Cooper-Harvey-Kennedy algorithm [5] (CHK), extended from AC with careful engineering, runs nearly as fast as LT in common cases [5,8], but is still simple to implement and reason about. Moreover, CHK generates dominator trees implicitly, and provides a faster tree construction algorithm.

Because CHK gives a relatively good trade-off between verifiability and efficency, we present CHK as an instance of `ALGDOM`. In the following sections, we first review the AC algorithm, and then study its extension CHK.

3 The Allen-Cocke Algorithm

The Allen-Cocke algorithm (AC) is an instance of the forward worklist-based Kildall's algorithm [10] that visits nodes in reverse postorder (PO) [9] (in which AC converges faster). At the high-level, our Coq implementation of AC works in three steps: 1) calculate the PO of a CFG by depth-first-search (DFS); 2) compute strict dominators for PO-numbered nodes in Kildall; 3) finally relate the analysis results to the original nodes. We omit the 3rd step's proofs here.

This section first presents a verified DFS algorithm that computes PO, then reviews Kildall's algorithm as implemented in the CompCert project [12], and finally it studies the implementation and metatheory of AC.

3.1 DFS: PO-numbering

DFS starts at the entry, visits nodes as deep as possible along each path, and backtracks when all deep nodes are visited. DFS generates PO by numbering a node after all its children are numbered. Figure 3 gives a PO-numbered CFG. In the CFG, we represent the depth-first-search (DFS) tree edges by solid arrows, and non-tree edges by dotted arrows. We draw the entry node in a box, and other nodes in circles. Each node is labeled by a pair with its original label name on the left, and its PO number on the right. Because DFS only visits reachable nodes, the PO numbers of unreachable nodes are represented by '_'.

Figure 5 shows the data structures and auxiliary functions used by a typical DFS algorithm that maintains four components to compute PO. `PostOrder`

```
Record PostOrder := mkPO { PO_cnt: positive; PO_l2p: LTree.t positive }.
Record Frame := mkFr { Fr_name: l; Fr_scs: list l }.
Definition dfs_F_type : Type := forall (succs: LTree.t (list l))
  (visited: LTree.t unit) (po:PostOrder) (stk: list Frame), PostOrder.
Definition dfs_F (f: dfs_F_type) (succs: LTree.t (list l))
  (visited: LTree.t unit) (po:PostOrder) (stk: list Frame): PostOrder :=
  match find_next succs visited po stk with
  | inr po' ⇒po'
  | inl (next, visited', po', stk') ⇒f succs visited' po' stk'
  end.
```

Fig. 5. The DFS algorithm.

takes the next available PO number and a map from nodes to their PO numbers with type `positive`. `succs` maps a node to its successors. To facilitate reasoning about DFS, we represent the recursive information of DFS explicitly by a list of `Frame` records that each contains a node `Fr_name` and its unprocessed successors `Fr_scs`. To prevent the search from revisiting nodes, the DFS algorithm uses `visited` to record visited nodes. `dfs_F` defines one recursive step of DFS.

Figure 4 gives a DFS execution sequence (by running `dfs_F` until all nodes are visited) of the CFG in Figure 3. We use $l[l_1 \cdots l_n]$ to denote a frame with the node l and its unprocessed successors l_1 to l_n; (l, p) to denote a node l and its PO p. Initially the DFS adds the entry and its successors to the stack. At each recursive step, `find_next` finds the next available node that is the unvisited node in the `Fr_scs` of the latest node l' of the stack. If the next available node exists, the DFS pushes the node with its successors to the stack, and makes the node to be visited. `find_next` pops all nodes in front of l', and gives them PO numbers. If `find_next` fails to find available nodes, the DFS stops.

We can see that the straightforward algorithm is not a structural recursion. To implement the algorithm in Coq, we must show that it terminates. Although in Coq we can implement the algorithm by well-founded recursion, such designs are hard to reason about [4]. One possible alternative is implementing DFS with a 'strong' dependent type to specify the properties that we need to reason about DFS. However, this design is not modular because when the type of DFS is not strong enough—for example, if we need a new lemma about DFS—we must extend or redesign its implementation by adding new invariants. Instead, following the ideas in Coq'Art [4], we implement DFS by iteration and prove its termination and inductive principle separately. By separating implementation and specification, the DFS design is modular, and easier to reason about.

Figure 6 presents our design. The top-level entry is `iter`, which needs a bounding step `n`, a fixpoint `F` and a default value `g`. `iter` only calls `g` when `n` reaches zero, and otherwise recursively calls one more iteration of `F`. If `F` is terminating, we can prove that there must exist a final value and a bound `n`, such that for any bound `k` that is greater than or equal to `n`, `iter` always stops and generates the same final value. In other words, `F` reaches a fixpoint in fewer

```
Fixpoint iter (A:Type) (n:nat) (F:A→A) (g:A) : A :=
  match n with
  | O ⇒g
  | S p ⇒F (iter A p F g)
  end.
Definition wf_stk succs visited stk :=
  stk_in_succs succs stk ∧incl visited succs
Program Fixpoint dfs_tmn succs visited po stk
  (Hp: wf_stk succs visited stk) {measure (size succs - size visited)}:
  { po':PostOrder | exists p:nat,
                  forall k (Hlt: p < k) (g:dfs_F_type),
                  iter _ k dfs_F g succs visited po stk = po' } :=
  match find_next succs visited po stk with
  | inr po' ⇒po'
  | inl (next, visited', po', stk') ⇒
     let _ := dfs_tmn succs visited' po' stk' _ in _
  end.
Program Definition dfs succs entry : PostOrder :=
fst (dfs_tmn succs empty (mkPO 1 empty) (mkFr entry [succs[entry]]) _).
```

Fig. 6. Termination of the DFS algorithm.

than `n` steps. The proof of the existence of `n` is erasable; the computation part provides a terminating algorithm, not requiring the bound step at runtime.

In Figure 6, `dfs_tmn` proves DFS termination, which is established by well-founded recursion over the number of unvisited nodes. This holds because each iteration the DFS visits more nodes. The invariant that the number of unvisited nodes decreases holds only for well-formed recursion states (`wf_stk`), which requires that all visited nodes and unprocessed nodes in frames are in the CFG.

To reason about `dfs`, we defined a well-founded inductive principle for `dfs` (See our code). With the inductive principle, we proved the following properties of DFS that are useful to establish the correctness of AC and CHK.

```
Variable (succs: ATree.t (list l)) (entry:l) (po:PostOrder).
Hypothesis Hdfs: dfs succs entry = po.
```

First of all, a non-entry node must have at least one predecessor that has a greater PO number than the node's. This is because 1) DFS must visit at least one predecessor of a node before visiting the node; 2) PO gives greater numbers to the nodes visited earlier:

```
Lemma dfs_order: forall l1 p1, l1 <> entry →(PO_l2p po)[l1] = Some p1,
  exists l2, exists p2,
  In l2 ((make_preds succs)[l1]) ∧(PO_l2p po)[l2] = Some p2 ∧p2 > p1.
(* Given succs, (make_preds succs) computes predecessors of each node. *)
```

Second, a node is PO-numbered iff the node is reachable:

```
Lemma dfs_reachable:forall l,(PO_l2p po)[l] <> None <→ (entry,succs)→* l.
```

Moreover, different nodes do not have the same PO number.

```
Lemma dfs_inj: forall l1 l2 p,
  (PO_l2p po)[l2] = Some p →(PO_l2p po)[l1] = Some p →l1 = l2.
```

```
Module Kildall (NS: PNODE_SET) (L: LATTICE). Section Kildall.
Variable succs: PTree.t (list positive).
Variable transf : positive →L.t →L.t.
Variable inits: list (positive * L.t).
Record state : Type := mkst { sin: PMap.t L.t; swrk: NS.t }.
Definition start_st := mkst (start_state_in inits) (NS.init succs).
Definition propagate_succ (out: L.t) (s: state) (n: positive) :=
  let oldl := s.(sin)[n] in
  let newl := L.lub oldl out in
  if L.eq newl oldl
  then mkst (PMap.set n newl s.(sin)) (NS.add n s.(swrk)) else s.
Definition step (s: state): PMap.t L.t + state :=
  match NS.pick s.(swrk) with
  | None ⇒inl s.(sin)
  | Some(n, rem) ⇒inr (fold_left
                        (propagate_succ (transf n s.(sin)[n]))
                        succs[n] (mkst s.(sin) rem))
  end.
Variable num : positive.
Definition fixpoint : option (PMap.t L.t):= Iter.iter step num start_st.
End Kildall. End Kildall.
```

Fig. 7. Kildall's algorithm.

3.2 Kildall's algorithm

Figure 7 summarizes the Kildall module used in the CompCert project. The module is parameterized by the following components: `NS` that provides the order to process nodes, and a lattice `L` that defines `top`, `bot`, equality (`eq`), least upper bound (`lub`) and order (`ge`) of the abstract domain of an analysis; `succs` that is a tree that maps each node to its successors; `transf` that is the transfer function of Kildall analysis; `inits` that initializes the analysis. Given the inputs, `state` records the iteration states that include `sin`, which records analysis states for each node, and a work list `swrk` containing nodes to process.

The `fixpoint` implements iterations by `Iter.iter`—bounded recursion with a maximal step number (`num`) [4]. `Iter.iter` is partial if an analysis does not stop after the maximal number of steps. A monotone analysis must reach its fixpoint after a finite number of steps. Therefore, we can alway pick a large enough number of steps for a monotone analysis.

Initially Kildall's algorithm calls `start_st` to initialize iteration states. Nodes not in `inits` are initialized to be the bottom of `L`. Then `start_st` adds all nodes into the worklist and starts loops. `step` defines the loop body. At `step`, Kildall's algorithm checks if there are still unprocessed nodes in the worklist. If the worklist is empty, the algorithm stops. Otherwise, `step` picks a node from the worklist in term of the order provided by `NS`, and then propagates its information (computed by `transf`) to all the node's successors by `propagate_succ`. In `propagate_succ`, the new value of a successor is `L.lub` of its old value and the

propagated value from its predecessor. The algorithm only adds a successor into the worklist when its value is changed.

Kildall's algorithm satisfies the following properties:

```
Variable res: PMap.t L.t.
Hypothesis Hfix: fixpoint = Some res.
```

First of all, the worklist contains nodes that have unstable successors in the current state. Formally, each state `st` preserves the following invariant:

```
forall n, NS.In n st.(swrk) ∨
  (forall s, In s (succs[n]) →L.ge st.(sin)[s] (transf n st.(sin)[n])).
```

Each iteration may only remove the picked node `n` from the worklist. If none of `n`'s successors' values are changed, no matter whether `n` belongs to its successors, `n` won't be added back to the worklist. Therefore, the above invariant holds. This invariant implies that when the analysis stops, all nodes hold the in-equations:

```
Lemma fixpoint_solution: forall s,
  In s (succs[n]) →L.ge res[s] (transf n res[n]).
```

The second property of Kildall's algorithm is *monotonicity*. At each iteration, the value of a successor of the picked node can only be updated from `oldl` to `newl`. Because `newl` is the least upper bound of `oldl` and `out`, `newl` is greater than or equal to `oldl`. Therefore, iteration states are always monotonic:

```
Lemma fixpoint_mono: incr (start_state_in inits) res.
```

where `incr` is a pointwise lift of `L.ge` for corresponding nodes. With monotonicity, we proved that Kildall's algorithm must terminate (See our code).

3.3 The AC Algorithm

AC instantiates `Kildall` with `PN` that picks nodes in reverse PO (by picking the maximal nodes from the worklist), and `LDoms` that defines the lattice of AC. Dominance analysis computes a set of strict dominators for each node. We represent the domain of `LDoms` by `option (set l)`. The `top` and `bot` of `LDoms` are `Some nil` and `None` respectively. The least upper bound, order and equality of `LDoms` are lifted from set intersection, set inclusion, and set equality to `option`: `None` is smaller than `Some x` for any `x`. This design leads to better performance by providing shortcuts for operations on `None`. Note that using `None` as `bot` does not make the height of `LDoms` to be infinite, because any non-`bot` element can only contain nodes in the CFG, and the height of `LDoms` is N.

AC uses the following transfer function and initialization:

```
Definition transf l1 input := l1 {+} input.
Definition inits := [(e, LDoms.top)].
```

Initially AC sets the strict dominators of the entry to be empty, and other nodes' strict dominators to be all labels in the function. The algorithm will iteratively remove non-strict-dominators from the sets until the conditions below hold (by Lemma `fixpoint_mono` and Lemma `fixpoint_solution`):

```
(forall s, In s (succs[n]) →
          L.ge (st.(sin))[s] (n{+}(st.(sin))[n])) ∧(st.(sin))[e] = {}.
```

which proves that AC satisfies `entry_sound` and `successors_sound`.

To show that the algorithm is complete, it is sufficient to show that each iteration state `st` preserves the following invariant:

```
forall n1 n2, ∼n1 ‘In‘ st.(sin)[n2] →∼(e, succs) |= n1 >> n2.
```

In other words, AC only removes non-strict dominators. Initially, AC sets the entry's strict dominators to be empty. Because in a well-formed CFG, the entry has no predecessors, the invariant holds at the very beginning. At each iteration, suppose that we pick a node `n`, and updates one of its successors `s`. Consider a node `n'` not in `LDoms.lub st.(sin)[s] (n {+} st.(sin)[n])`. If `n'` is not in `LDoms.lub st.(sin)[s]`, then `n'` does not strictly dominate `s` because `st` holds the invariant. If `n'` is not in `(n {+} st.(sin)[n])`, then `n'` does not strictly dominate `n` because `st` holds the invariant. Appending the path from the entry to `n` that bypasses `n'` with the edge from `n` to `s` leads to a path from the entry to `s` that bypasses `n'`. Therefore, `n'` does not strictly dominate `s`, either.

4 Extension: the Cooper-Harvey-Kennedy Algorithm

The CHK algorithm is based on the following observation: when AC processes nodes in a reversed post-order (PO), if we represent the set of strict dominators in a list, and always add a newly discovered strict dominator at the head of the list (on the left in Figure 9), the list must be sorted by PO. Figure 9 shows the execution of the algorithm for the CFG in Figure 3.

Because lists of strict dominators are always sorted, we can implement the set intersection (`lub`) and the set comparison (`eq`) of two sorted lists by traversing the two lists only once. Moreover, the algorithm only calls `eq` after `lub`. Therefore, we can group `lub` and `eq` into `LDoms.lub` together. The following defines a `merge` function used by `LDoms.lub` that intersects two sorted lists and returns whether the final result is equal to the left one:

```
Program Fixpoint merge (l1 l2: list positive) (acc:list positive * bool)
  {measure (length l1 + length l2)}: (list positive * bool) :=
  let '(rl, changed) := acc in
  match l1, l2 with
  | p1::l1', p2::l2' ⇒
      match (Pcompare p1 p2 Eq) with
      | Eq ⇒merge l1' l2' (p1::rl, changed)
      | Lt ⇒merge l1' l2 (rl, true)
      | Gt ⇒merge l1 l2' (rl, changed)
      end
  | nil, _ ⇒acc
  | _::_, nil ⇒(rl, true)
  end.
(* (Pcompare p1 p2 Eq) returns whether p1 = p2, p1 < p2 or p1 > p2. *)
```

4.1 Correctness

To show that CHK is still correct, it is sufficient to show that all lists are well-sorted at each iteration, which ensures that the above `merge` correctly implements intersection and comparison. First, if a node with number `n` still maps to `bot`, the worklist must contain one of its predecessors that has a greater number.

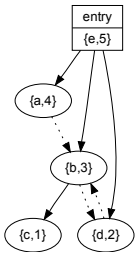

Fig. 8. Dominator Trees.

Nodes	sin								
5	[]	[]	[]	[]	[]	[]	[]	[]	[]
4	·	[5]	[5]	[5]	[5]	[5]	[5]	[5]	[5]
3	·	·	[45]	[45]	[45]	[5]	[5]	[5]	[5]
2	·	·	·	[345]	[345]	[345]	[35]	[35]	[35]
1	·	[5]	[5]	[5]	[5]	[5]	[5]	[5]	[5]
swrk	[54321]	[4321]	[321]	[21]	[1]	[3]	[21]	[1]	[]

Fig. 9. The execution of CHK.

```
forall n, in_cfg n succs →(st.(sin))[n] = None →
  exists p, In p ((make_preds succs)[n]) ∧p > n ∧PN.In p st.(st_wrk).
(* in_cfg checks if a node is in CFG. *)
```

This invariant holds in the beginning because all nodes are in the worklist. At each iteration, the invariant implies that the picked node `n` with the maximal number in `st.(st_wrk)` is not `bot`. Suppose it is `bot`, there cannot be any node with greater number in the worklist. This property ensures that after each iteration, the successors of `n` cannot be `bot`, and that the new nodes added into the worklist cannot be `bot`, because they must be those successors. Therefore, the predecessors of the remaining `bot` nodes still in the worklist cannot be `n`. Since only `n` is removed, the rest of the `bot` nodes still hold the above invariant.

In the algorithm, a node's value is changed from `bot` to non-`bot` when one of its non-`bot` predecessors is processed. With the above invariant, we know that the predecessor must be of larger number. Once a node turns to be non-`bot`, no new elements will be added in its set. Therefore, this implies that, at each iteration, if the value of a node is not `bot`, then all its candidate strict dominators must be larger than the node:

```
forall n sdms, (st.(sin))[n] = Some sdms →Forall (Plt n) sdms.
(* Plt is the less-than of positive. *)
```

Moreover, a node `n` is considered as a candidate of strict dominators originally by `tranf` that always cons `n` at the head of `(st.(sin))[n]`. Therefore, we proved that the non-`bot` value of a node is always sorted:

```
forall n sdms, (st.(sin))[n] = Some sdms →Sorted Plt (n::sdms).
```

5 Applications

5.1 Type Checker

The first application is the type checker of Vminus. The Vminus language in Figure 1 is in SSA form [6] in which each variable may be defined only once, statically, and each use of the variable must be dominated by its definition with respect to the control-flow graph of the containing function. To maintain these invariants in the presence of branches and loops, SSA form uses ϕ-instructions to merge definitions from different incoming paths. As usual in SSA representation,

the ϕ nodes join together values from a list of predecessor blocks of the control-flow graph—each ϕ node takes a list of (value, label) pairs that indicates the value chosen when control transfers from a predecessor block with the associated label.

To check that a program is in SSA form, we need to extend domination relations from the block-level to the instruction-level. Instruction positions are denoted by program counters pc. We write $f[pc] = \lfloor insn \rfloor$ if $insn$ is at pc of f.

Definition 2 (Domination (Instruction-level)).

- val **uses** $r \triangleq val = r$.
- $insn$ **uses** $r \triangleq \exists val.\, val$ **uses** $r \wedge val$ *is an operand of* $insn$.
- *A variable* r *is defined at a program counter* pc *of* f*, written* f **defines** r @ pc*, if* $f[pc] = \lfloor insn \rfloor$ *and* r *is the left-hand side of* $insn$.
- *In function* f*,* pc_1 ***strictly dominates*** pc_2*, written* $f \models pc_1 \gg pc_2$*, if* pc_1 *and* pc_2 *are at distinct blocks* l_1 *and* l_2 *respectively and* $f \models l_1 \gg l_2$*; or if* pc_1 *and* pc_2 *are in the same block, and* pc_1 *appears earlier than* pc_2.
- $\mathbf{sdom}_f(pc)$ *is the set of variables whose definitions strictly dominate* pc*:* $\{r | f$ **defines** r @ pc' *and* $f \models pc' \gg pc\}$

Then we check if a program is of SSA form with the following rules:

$$\frac{\forall r.(\hat{\phi}\ \mathbf{uses}\ r \Longrightarrow r\ \in\ \mathbf{sdom}_f(pc))}{f \vdash_{\succ} \hat{\phi}\ @\ pc} \qquad \frac{\overline{\forall r_j.(val_j\ \mathbf{uses}\ r_j \Longrightarrow r_j\ \in\ \mathbf{sdom}_f(l_j.\,\mathbf{t}))}^{\,j}}{f \vdash_{\succ} r = \mathbf{phi}\ typ\ \overline{[val_j, l_j]}^{\,j}}$$

The left rule ensures that a non-ϕ-instruction (c or tmn) can only use the definitions in the scope of $\mathbf{sdom}_f(pc)$; the right rule ensures that in ϕ, an incoming value must use the definition that strictly dominates the end of the corresponding incoming block where $l.\,\mathbf{t}$ is the program counter at the end of l. Please refer to [14] for the type safety proofs of Vminus.

5.2 SSA-based Optimizations

The SSA form is good for implementing optimizations because the SSA invariants make def/use information of variables explicit, enforcing fewer mutable states [2]. An SSA-based transformation is correct if it preserves the semantics of the original program and its transformed program is still in SSA. Here, we briefly show how to reason about well-formedness-preservation by examples.

First, we proved that the strict domination relation at the instruction level still satisfies transitivity and acyclicity.

Lemma 5.
- If $f \models pc_1 \gg pc_2$ *and* $f \models pc_2 \gg pc_3$*, then* $f \models pc_1 \gg pc_3$.
- If pc *is in a reachable block, then* $\neg f \models pc \gg pc$.

Consider the following typical SSA-based optimization:

Original	Transformed
$e : \cdots$	$e : \cdots$
$@pc_1$	$r_4 := r_1 * r_2$
$\quad \mathbf{br}\, r_0\, l_1\, l_2$	$\quad \mathbf{br}\, r_0\, l_1\, l_2$
$l_1 : r_3 = \mathbf{phi}\, \mathbf{int}[0, e][r_5, l_1]$	$l_1 : r_3 = \mathbf{phi}\, \mathbf{int}[0, e][r_5, l_1]$
$@pc_2\, r_4 := r_1 * r_2$	
$\quad r_5 := r_3 + r_4$	$\quad r_5 := r_3 + r_4$
$\quad r_6 := r_5 \geq 100$	$\quad r_6 := r_5 \geq 100$
$\quad \mathbf{br}\, r_6\, l_1\, l_2$	$\quad \mathbf{br}\, r_6\, l_1\, l_2$
$l_2 : r_7 = \mathbf{phi}\, \mathbf{int}[0, e][r_5, l_2]$	$l_2 : r_7 = \mathbf{phi}\, \mathbf{int}[0, e][r_5, l_1]$
$@pc_3\, r_8 := r_1 * r_2$	
$\quad r_9 := r_8 + r_7$	$\quad r_9 := r_4 + r_7$

In the original program, $r_1 * r_2$ is a partial common expression for the definitions of r_4 and r_8, because there is no domination relation between r_4 and r_8. Therefore, eliminating the common expression directly is not correct.

We might transform this program in three steps. First, we move the instruction $r_4 := r_1 * r_2$ from l_1 to the end of e. Because e strictly dominates l_1, we have $f \models pc_1 \gg pc_2$ where pc_1 is exactly before $e.\mathbf{t}$; $f\, \mathbf{defines}\, r_4\ @\ pc_2$. By Lemma 5, the definition of r_4 at pc_1 should still strictly dominate all its uses.

We have $f \models pc_1 \gg pc_3$ where $f\, \mathbf{defines}\, r_8\ @\ pc_3$, because e strictly dominates l_2. Then, we can safely replace all the uses of r_8 by r_4, because the definition of r_4 at pc_1 dominates all the uses of r_8 (by Lemma 5).

Finally, by Lemma 5, we know that r_4 and r_8 cannot be equal. Therefore, we can remove r_8, because there are no uses of r_8 after the substitution. The final program after the transformations is shown on the right of the above example.

5.3 Constructing Dominator Trees

In practice, compilers construct dominator trees from dominators, and analyze or optimize programs by recursion on dominator trees.

Definition 3.

- *A block l_1 is an **immediate dominator** of a block l_2, written $G \models l_1 \ggg l_2$, if $G \models l_1 \gg l_2$ and $(\forall G \models l_3 \gg l_2, G \models l_3 \gg= l_1)$.*
- *A tree is called a **dominator tree** of G if the tree has an edge from l to l' iff $G \models l \ggg l'$.*

Figure 8 shows the dominator tree of a CFG, in which solid edges represent tree edges, and dotted edges represent non-tree but CFG edges. Formally, a dominator tree has the inductive **well-formed** property with which we can reason about recursion on dominator trees: given a tree node l, 1) l is reachable; 2) l is different from all labels in l's descendants; 3) labels of l's subtrees are disjointed; 4) l immediate-dominates its children; 5) l's subtrees are well-formed.

Consider the final analysis results of CHK in Figure 9, we can see that for each node, its list of strict dominators exactly presents a path from root to the node on the dominator tree. Therefore, we can construct a dominator tree by

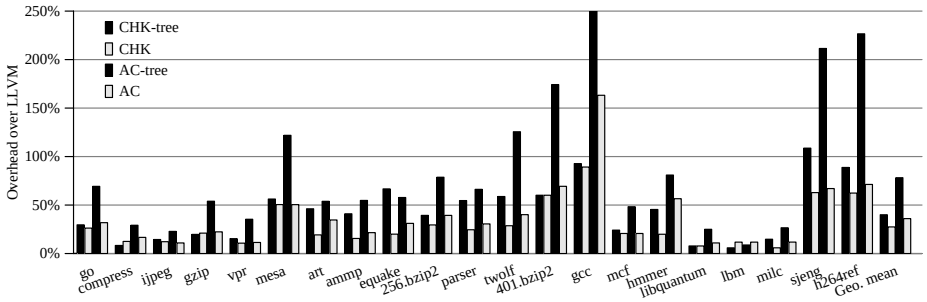

Fig. 10. Analysis overhead over LLVM's dominance analysis for our extracted analysis.

merging the paths. We proved that the algorithm correctly constructs a well-formed dominator tree (See our code). For the sake of space, we only present that each tree edge represents $\ggg$ by showing that for any node l in the final state, the list of l's dominators must be sorted by $\ggg$.

We first show that the list is sorted by $\gg$. Consider two adjacent nodes in the list, l_1 and l_2, such that $l_1 < l_2$. Because of soundness, $G \models l_1 \gg= l$ and $G \models l_2 \gg= l$. By Lemma 4, $G \models l_2 \gg l_1 \vee G \models l_1 \gg l_2$. Suppose $G \models l_1 \gg l_2$, by completeness, l_1 must be in the strict dominators computed for l_2, and therefore, be greater than l_2. This is a contradiction. Then, we prove that the list is sorted by $\ggg$. Suppose $G \models l_3 \gg l_1$. By Lemma 1, $G \models l_3 \gg l$. By completeness, l_3 must be in the list. We have two cases: 1) $l_3 \geq l_2$: because the list is sorted by $\gg$, $G \models l_3 \gg= l_2$; 2) $l_3 \leq l_1$: this is a contradiction by Lemma 3.

6 Performance Evaluation

We use Coq extraction to obtain a certified implementation of AC and CHK. We evaluate the performance of the resultant code on a 1.73 GHz Intel Core i7 processor with 8 GB memory running benchmarks selected from the SPEC CPU benchmark suite that consist of over 873k lines of C source code.

Figure 10 reports the analysis time overhead (smaller is better) over the LLVM dominance analysis (which uses LT) baseline. LT only generates dominator trees. Given a dominator tree, the strict dominators of a tree node are all the node's ancestors. The second left bar of each group shows the overhead of CHK, which provides an average overhead of 27.45%. The right-most bar of each group is the overhead of AC, which provides 36.02% on average.

To study the asymptotic complexity, Figure 11 shows the result of graphs that elicit the worst-case behavior used in [8]. On average, CHK is 86.59 times slower than LT. The '_' indicates that the running time is too long to collect. For the testcases on which AC stops, AC is 226.14 times slower than LT.

The results of CHK match earlier experiments [8,5]: in common cases, CHK runs nearly as fast as LT. For programs with reducible CFGs, a forward iteration analysis in reverse PO halts in no more than 6 passes [9], and most CFGs of the benchmarks are reducible. The worst-case tests contain huge irreducible CFGs.

Instance			Analysis Times (s)				
Name	Vertices	Edges	LT	CHK	CHK-tree	AC	AC-tree
idfsquad	6002	10000	0.08	10.54	24.87	-	-
ibfsquad	4001	6001	0.14	11.38	13.16	12.43	30.00
itworst	2553	5095	0.14	8.47	11.22	19.16	69.72
sncaworst	3998	3096	0.19	17.03	32.08	205.07	740.53

Fig. 11. Worst-case behavior.

Different from these experiments, AC does not provide large overhead, because we use `None` to represent `bot`, which provides shortcuts for set operations.

As shown in Section 5.3, CHK computes dominator trees implicitly, while AC needs additional costs to create dominator trees. Figure 10 and Figure 11 also report the performance of the dominator tree construction. CHK-tree stands for the algorithm that first computes dominators by CHK, and then runs the tree construction defined in Section 5.3. AC-tree stands for the algorithm that first computes dominators by AC, sorts strict dominators for each node, and then runs the same tree construction. For common programs, on average, CHK-tree provides an overhead 40.00% over the baseline; AC-tree provides an overhead 78.20% over the baseline (gcc's overhead is 361.23%). The additional overhead of AC-tree is from its sorting algorithm. For worst-case programs, on average, CHK-tree is 104.48 times slower than LT. For the testcases on which AC-tree stops, on average, AC-tree is 738.24 times slower than LT.

These results indicate that CHK makes a good trade-off between simplicity and efficiency.

7 Related Work

Machine-checked formalizations. The Vellvm project [14] uses dominance analysis to design a type checker of LLVM bitcode in SSA form. This paper extends and generalizes the implementation and metatheory in the Vellvm project. The CompCertSSA project [3] improves the CompCert compiler by creating a verified SSA-based middle-end. They also formalize the AC algorithm to validate SSA construction and GVN passes, and prove the soundness of AC. We implement both AC and CHK—an extension of AC in a generic way, and prove they are both sound and complete. We also provide the corresponding dominator tree constructions, and evaluate performance.

Informal formalizations. Georgiadis and Tarjan [7] propose an almost linear-time algorithm that validates if a tree is a dominator tree of a CFG. Although the algorithm is fast, it is nearly as complicated as the LT algorithm, and it requires a substantial amount of graph theory. Ramalingam [13] proposes another dominator tree validation algorithm by reducing validating dominator trees to validating loop structures. However, in practice, most of modern loop identification algorithms used in LLVM and GCC are based on dominance analysis to find loop headers and bodies.

8 Conclusion

This paper provided an abstract specification of dominance analysis that is crucial for compiler design/verification and program analysis. We implemented and certified an instance of the specification that has a good trade-off between efficiency and simplicity. We also presented several applications of the analysis: a type checker for the SSA form; verifying SSA-based optimizations; and constructing dominator trees. This development is a part of the Vellvm project. However, our work might be used in other compiler verification projects [3].

Acknowledgments We thank Santosh Nagarakatte and Milo Martin whose valuable discussions and technical input helped us carry out this research. This research was sponsored in part by NSF grant CCF-1065116. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

References

1. F. E. Allen and J. Cocke. Graph theoretic constructs for program control flow analysis. Technical report, IBM T.J. Watson Research Center, 1972.
2. A. W. Appel. *Modern Compiler Implementation in C: Basic Techniques*. Cambridge University Press, 1997.
3. G. Barthe, D. Demange, and D. Pichardie. A formally verified SSA-based middle-end - Static Single Assignment meets CompCert. In *ESOP '12*, 2012.
4. Y. Bertot and P. Castéran. Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions, 2004.
5. K. D. Cooper, T. J. Harvey, and K. Kennedy. A simple, fast dominance algorithm. Available online at `www.cs.rice.edu/~keith/Embed/dom.pdf`, 2000.
6. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13:451–490, 1991.
7. L. Georgiadis and R. E. Tarjan. Dominator tree verification and vertex-disjoint paths. In *SODA '05*, pages 433–442, 2005.
8. L. Georgiadis, R. F. Werneck, R. E. Tarjan, and D. I. August. Finding dominators in practice. In *ESA '04*, pages 677–688, 2004.
9. J. B. Kam and J. D. Ullman. Global data flow analysis and iterative algorithms. *J. ACM*, 23(1):158–171, Jan. 1976.
10. G. A. Kildall. A unified approach to global program optimization. In *POPL '73*, pages 194–206, 1973.
11. T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1:121–141, 1979.
12. X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.
13. G. Ramalingam. On loops, dominators, and dominance frontiers. *ACM Trans. Program. Lang. Syst.*, 24(5):455–490, 2002.
14. J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *POPL '12*, 2012.