Proving a data flow analysis algorithm for computing dominators

Nan Jiang

March 17, 2025

Abstract

This entry formalises a fast iterative algorithm for computing dominators [1]. It gives a specification of computing dominators on a control flow graph where each node refers to its reverse post order number. A semilattice of reversed-ordered list which represents dominators is built and a Kildall's algorithm on the semilattice is defined for computing dominators. Finally the soundness and completeness of the algorithm are proved w.r.t. the specification.

Contents

1 The specification of computing dominators

theory *Cfg*
imports *Main*
begin

The specification of computing dominators is defined. For fast data flow analysis presented by CHK [1], a directed graph with explicit node list and

sets of initial nodes is defined. Each node refers to its rPO (reverse PostOrder) number w.r.t a DFST, and related properties as assumptions are handled using a locale.

type-synonym *'a digraph* = (*'a* × *'a*) *set*

record *'a graph-rec* =
  g-V :: *'a list*
  g-V0 :: *'a set*
  g-E :: *'a digraph*

  tail :: *'a* × *'a* ⇒ *'a*
  head :: *'a* × *'a* ⇒ *'a*

definition *wf-cfg* :: *'a graph-rec* ⇒ *bool* **where**
  wf-cfg G ≡ *g-V0 G* ⊆ *set*(*g-V G*)

type-synonym *node* = *nat*

locale *cfg-doms* =
  — Nodes are rPO numbers
  fixes *G* :: *nat graph-rec* (**structure**)

  — General properties
  assumes *wf-cfg*: *wf-cfg G*
  assumes *tail*[*simp*]: *e* = (*u*,*v*) ⟹ *tail G e* = *u*
  assumes *head*[*simp*]: *e* = (*u*,*v*) ⟹ *head G e* = *v*
  assumes *tail-in-verts*[*simp*]: *e* ∈ *g-E G* ⟹ *tail G e* ∈ *set* (*g-V G*)
  assumes *head-in-verts*[*simp*]: *e* ∈ *g-E G* ⟹ *head G e* ∈ *set* (*g-V G*)

  — Properties of a cfg where nodes are rPO numbers
  assumes *entry0*: *g-V0 G* = {*0*}
  assumes *dfst*: ∀ *v* ∈ *set* (*g-V G*) − {*0*}. ∃ *prev*. (*prev*, *v*) ∈ *g-E G* ∧ *prev* < *v*
  assumes *reachable*: ∀ *v* ∈ *set* (*g-V G*). *v* ∈ (*g-E G*)* `` {*0*}
  assumes *verts*: *g-V G* = [*0* ..< (*length* (*g-V G*))]

 — It is required that the entry node has an immediate successor which is not itself; Otherwise, no need to compute dominators It is required for proving the lemma: "wf_dom start (unstables r step start)".
  assumes *succ-of-entry0*: ∃ *s*. (*0*,*s*) ∈ *g-E G* ∧ *s* ≠ *0*

begin

inductive *path-entry* :: *nat digraph* ⇒ *nat list* ⇒ *nat* ⇒ *bool* **for** *E* **where**
  path-entry0: *path-entry E* [] *0*
| *path-entry-prepend*: ⟦ (*u*,*v*)∈ *E*; *path-entry E l u* ⟧ ⟹ *path-entry E* (*u*#*l*) *v*

lemma *path-entry0-empty-conv*: *path-entry E* [] *v* ⟷ *v* = *0*
⟨*proof*⟩

inductive-cases *path-entry-uncons*: *path-entry E* $(u'\#l)$ *w*
inductive-simps *path-entry-cons-conv*: *path-entry E* $(u'\#l)$ *w*

lemma *single-path-entry*: *path-entry E* $[p]$ *w* $\Longrightarrow$ $p = 0$
⟨*proof*⟩

lemma *path-entry-append*:
⟦ *path-entry E l v*; $(v,w) \in E$ ⟧ $\Longrightarrow$ *path-entry E* $(v\#l)$ *w*
⟨*proof*⟩

lemma *entry-rtrancl-is-path*:
assumes $(0,v) \in E^*$
obtains *p* **where** *path-entry E p v*
⟨*proof*⟩

lemma *path-entry-is-trancl*:
assumes *path-entry E l v*
and $l \neq []$
shows $(0,v) \in E^+$
⟨*proof*⟩

lemma *tail-is-vert*: $(u,v) \in$ *g-E G* $\Longrightarrow$ $u \in$ *set* (*g-V G*)
⟨*proof*⟩

lemma *head-is-vert*: $(u,v) \in$ *g-E G* $\Longrightarrow$ $v \in$ *set* (*g-V G*)
⟨*proof*⟩

lemma *tail-is-vert2*: $(u,v) \in$ (*g-E G*)$^+$ $\Longrightarrow$ $u \in$ *set* (*g-V G*)
⟨*proof*⟩

lemma *head-is-vert2*: $(u,v) \in$ (*g-E G*)$^+$ $\Longrightarrow$ $v \in$ *set* (*g-V G*)
⟨*proof*⟩

lemma *verts-set*: *set* (*g-V G*) = $\{0$ *..<* *length* (*g-V G*)$\}$
⟨*proof*⟩

lemma *fin-verts*: *finite* (*set* (*g-V G*))
⟨*proof*⟩

lemma *zero-in-verts*: $0 \in$ *set* (*g-V G*)
⟨*proof*⟩

lemma *verts-not-empty*: *g-V G* $\neq []$
⟨*proof*⟩

lemma *len-verts-gt0*: *length* (*g-V G*) > 0
⟨*proof*⟩

lemma *len-verts-gt1*: *length* (*g-V G*) > *1*
⟨*proof*⟩

lemma *verts-ge-Suc0* : ¬ [*0*..<*length* (*g-V G*)] = [*0*]
⟨*proof*⟩

lemma *distinct-verts1*: *distinct* [*0*..<*length* (*g-V G*)]
⟨*proof*⟩

lemma *distinct-verts2*: *distinct* (*g-V G*)
⟨*proof*⟩

lemma *single-entry*: *is-singleton* (*g-V0 G*)
⟨*proof*⟩

lemma *entry-is-0*: *the-elem* (*g-V0 G*) = *0*
⟨*proof*⟩

lemma *wf-digraph*: *cfg-doms G* ⟨*proof*⟩

lemma *path-entry-prepend-conv*: *path-entry* (*g-E G*) *p n* ⟹ *p* ≠ [] ⟹ ∃ *v*. *path-entry* (*g-E G*) (*tl p*) *v* ∧ (*v*, *n*) ∈ (*g-E G*)
⟨*proof*⟩

lemma *path-verts*: *path-entry* (*g-E G*) *p n* ⟹ *n* ∈ *set* (*g-V G*)
⟨*proof*⟩

lemma *path-in-verts*:
assumes *path-entry* (*g-E G*) *l v*
shows *set l* ⊆ *set* (*g-V G*)
⟨*proof*⟩

lemma *any-node-exits-path*:
assumes *v* ∈ *set* (*g-V G*)
shows ∃ *p*. *path-entry* (*g-E G*) *p v*
⟨*proof*⟩

lemma *entry0-path*: *path-entry* (*g-E G*) [] *0*
⟨*proof*⟩

definition *reachable* :: *node* ⇒ *bool*
where *reachable v* ≡ *v* ∈ (*g-E G*)* `` {*0*}

lemma *path-entry-reachable*:
assumes *path-entry* (*g-E G*) *p n*
shows *reachable n*
⟨*proof*⟩

lemma *nin-nodes-reachable*: *n* ∉ *set* (*g-V G*) ⟹ ¬ *reachable n*

⟨*proof*⟩

lemma *reachable-path-entry*: *reachable n* ⟹ ∃ *p*. *path-entry* (*g-E G*) *p n*
⟨*proof*⟩

lemma *path-entry-unconc*:
assumes *path-entry* (*g-E G*) (*la*@*lb*) *w*
obtains *v* **where** *path-entry* (*g-E G*) *lb v*
⟨*proof*⟩

lemma *path-entry-append-conv*:
path-entry (*g-E G*) (*v*#*l*) *w* ⟷ (*path-entry* (*g-E G*) *l v* ∧ (*v*,*w*) ∈ (*g-E G*))
⟨*proof*⟩

lemma *takeWhileNot-path-entry*:
assumes *path-entry E p x*
and *v* ∈ *set p*
and *takeWhile* ((≠) *v*) (*rev p*) = *c*
shows *path-entry E* (*rev c*) *v*
⟨*proof*⟩

lemma *path-entry-last*: *path-entry* (*g-E G*) *p n* ⟹ *p* ≠ [] ⟹ *last p* = *0*
⟨*proof*⟩

lemma *path-entry-loop*:
assumes *n-path*: *path-entry* (*g-E G*) *p n*
and *n*: *n* ∈ *set p*
shows ∃ *p*′. *path-entry* (*g-E G*) *p*′ *n* ∧ *n* ∉ *set p*′
⟨*proof*⟩

lemma *path-entry-hd-edge*:
assumes *path-entry* (*g-E G*) *pa p*
and *pa* ≠ []
shows (*hd pa*, *p*) ∈ (*g-E G*)
⟨*proof*⟩

lemma *path-entry-edge*:
assumes *pa* ≠ []
and *path-entry* (*g-E G*) *pa p*
shows ∃ *u*∈*set pa*. (*path-entry* (*g-E G*) (*rev* (*takeWhile* ((≠) *u*) (*rev pa*))) *u*) ∧
(*u*,*p*) ∈ (*g-E G*)
⟨*proof*⟩

definition *is-tail* :: *node* ⇒ *node* × *node* ⇒ *bool*
where *is-tail v e* = (*v* = *tail G e*)

definition *is-head* :: *node* ⇒ *node* × *node* ⇒ *bool*
where *is-head v e* = (*v* = *head G e*)

definition *succs*:: *node* ⇒ *node set*
where *succs v* = (*g-E G*) `` {*v*}

lemma *succ-in-verts*: *s* ∈ *succs n* ⟹ {*s*,*n*} ⊆ *set* (*g-V G*)
⟨*proof*⟩

lemma *succ0-not-nil*: *succs 0* ≠ {}
⟨*proof*⟩

definition *prevs*:: *node* ⇒ *node set* **where**
prevs v = (*converse* (*g-E G*)) `` {*v*}

lemma *v* ∈ *succs u* ⟷ *u* ∈ *prevs v*
⟨*proof*⟩

lemma *succ-edge*: ∀ *v* ∈ *succs n*. (*n*,*v*) ∈ *g-E G*
⟨*proof*⟩

lemma *prev-edge*: *u* ∈ *set* (*g-V G*) ⟹ ∀ *v* ∈ *prevs u*. (*v*, *u*) ∈ *g-E G*
⟨*proof*⟩

lemma *succ-in-G*: ∀ *v* ∈ *succs n*. *v* ∈ *set* (*g-V G*)
⟨*proof*⟩

lemma *succ-is-subset-of-verts*: ∀ *v* ∈ *set* (*g-V G*). *succs v* ⊆ *set*(*g-V G*)
⟨*proof*⟩

lemma *fin-succs*: ∀ *v* ∈ *set* (*g-V G*). *finite* (*succs v*)
⟨*proof*⟩

lemma *fin-succs'*: *v* < *length* (*g-V G*) ⟹ *finite* (*succs v*)
⟨*proof*⟩

lemma *succ-range*: ∀ *v* ∈ *succs n*. *v* < *length* (*g-V G*)
⟨*proof*⟩

lemma *path-entry-gt*:
assumes ∀ *p*. *path-entry E p n* ⟶ *d* ∈ *set p*
and ∀ *p*. *path-entry E p n'* ⟶ *n* ∈ *set p*
shows ∀ *p*. *path-entry E p n'* ⟶ *d* ∈ *set p*
⟨*proof*⟩

definition *dominate* :: *nat* ⇒ *nat* ⇒ *bool*
where *dominate n1 n2* ≡
∀ *pa*. *path-entry* (*g-E G*) *pa n2* ⟶
(*n1* ∈ *set pa* ∨ *n1* = *n2*)

definition *strict-dominate*:: *nat* ⇒ *nat* ⇒ *bool* **where**
strict-dominate n1 n2 ≡

$\forall\, pa.\ path\text{-}entry\ (g\text{-}E\ G)\ pa\ n2 \longrightarrow$
$(n1 \in set\ pa \wedge n1 \neq n2)$

lemma *any-dominate-unreachable*: $\neg\ reachable\ n \Longrightarrow dominate\ d\ n$
⟨*proof*⟩

lemma *any-sdominate-unreachable*: $\neg\ reachable\ n \Longrightarrow strict\text{-}dominate\ d\ n$
⟨*proof*⟩

lemma *dom-reachable*: $reachable\ n \Longrightarrow dominate\ d\ n \Longrightarrow reachable\ d$
⟨*proof*⟩

lemma *dominate-refl*: *dominate n n*
⟨*proof*⟩

lemma *entry0-dominates-all*: $\forall\, p \in set\ (g\text{-}V\ G).\ dominate\ 0\ p$
⟨*proof*⟩

lemma *strict-dominate i j* $\Longrightarrow j \in set\ (g\text{-}V\ G) \Longrightarrow i \neq j$
⟨*proof*⟩

definition *non-strict-dominate*:: *nat* ⇒ *nat* ⇒ *bool* **where**
non-strict-dominate n1 n2 $\equiv \exists\, pa.\ path\text{-}entry\ (g\text{-}E\ G)\ pa\ n2 \wedge (n1 \notin set\ pa)$

lemma *any-sdominate-0*: $n \in set\ (g\text{-}V\ G) \Longrightarrow$ *non-strict-dominate n 0*
⟨*proof*⟩

lemma *non-sdominate-succ*: $(i,j\ ) \in g\text{-}E\ G \Longrightarrow k \neq i \Longrightarrow$ *non-strict-dominate k i* $\Longrightarrow$ *non-strict-dominate k j*
⟨*proof*⟩

lemma *any-node-non-sdom0*: *non-strict-dominate k 0*
⟨*proof*⟩

lemma *nonstrict-eq*: *non-strict-dominate i j* $\Longrightarrow \neg$ *strict-dominate i j*
⟨*proof*⟩

lemma *dominate-trans*:
assumes *dominate n1 n2*
and *dominate n2 n3*
shows *dominate n1 n3*
⟨*proof*⟩

lemma *len-takeWhile-lt*: $x \in set\ xs \Longrightarrow length\ (takeWhile\ ((\neq)\ x)\ xs) < length\ xs$
⟨*proof*⟩

lemma *len-takeWhile-comp*:
assumes *n1* $\in$ *set xs*

and $n2 \in set\ xs$
and $n1 \neq n2$
shows $length\ (takeWhile\ ((\neq)\ n1)\ xs) \neq length\ (takeWhile\ ((\neq)\ n2)\ xs)$
⟨*proof*⟩

lemma *len-takeWhile-comp1*:
assumes $n1 \in set\ xs$
and $n2 \in set\ xs$
and $n1 \neq n2$
shows $length\ (takeWhile\ ((\neq)\ n1)\ (rev\ (x\ \#\ xs))) \neq length\ (takeWhile\ ((\neq)\ n2)\ (rev\ (x\ \#\ xs)))$
⟨*proof*⟩

lemma *len-takeWhile-comp2*:
assumes $n1 \in set\ xs$
and $n2 \notin set\ xs$
shows $length\ (takeWhile\ ((\neq)\ n1)\ (rev\ (x\ \#\ xs))) \neq length\ (takeWhile\ ((\neq)\ n2)\ (rev\ (x\ \#\ xs)))$
⟨*proof*⟩

lemma *len-compare1*:
assumes $n1 = x$ **and** $n2 \neq x$
shows $length\ (takeWhile\ ((\neq)\ n1)\ (rev\ (x\ \#\ xs))) \neq length\ (takeWhile\ ((\neq)\ n2)\ (rev\ (x\ \#\ xs)))$
⟨*proof*⟩

lemma *len-compare2*:
assumes $n1 \in set\ xs$
and $n1 \neq n2$
shows $length\ (takeWhile\ ((\neq)\ n1)\ (rev\ (x\ \#\ xs))) \neq length\ (takeWhile\ ((\neq)\ n2)\ (rev\ (x\ \#\ xs)))$
⟨*proof*⟩

lemma *len-takeWhile-set*:
assumes $length\ (takeWhile\ ((\neq)\ n1)\ xs) > length\ (takeWhile\ ((\neq)\ n2)\ xs)$
and $n1 \neq n2$
and $n1 \in set\ xs$
and $n2 \in set\ xs$
shows $set\ (takeWhile\ ((\neq)\ n2)\ xs) \subseteq set\ (takeWhile\ ((\neq)\ n1)\ xs)$
⟨*proof*⟩

lemma *reachable-dom-acyclic*:
assumes *reachable n2*
and *dominate n1 n2*
and *dominate n2 n1*
shows $n1 = n2$
⟨*proof*⟩

lemma *sdom-dom*: *strict-dominate n1 n2* $\Longrightarrow$ *dominate n1 n2*

⟨*proof*⟩

lemma *dominate-sdominate*: *dominate n1 n2* ⟹ *n1* ≠ *n2* ⟹ *strict-dominate n1 n2*
⟨*proof*⟩

lemma *sdom-neq*:
assumes *reachable n2*
and *strict-dominate n1 n2*
shows *n1* ≠ *n2*
⟨*proof*⟩

lemma *reachable-dom-acyclic2*:
assumes *reachable n2*
and *strict-dominate n1 n2*
shows ¬ *dominate n2 n1*
⟨*proof*⟩

lemma *not-dom-eq-not-sdom*: ¬ *dominate n1 n2* ⟹ ¬ *strict-dominate n1 n2*
⟨*proof*⟩

lemma *reachable-sdom-acyclic*:
assumes *reachable n2*
and *strict-dominate n1 n2*
shows ¬ *strict-dominate n2 n1*
⟨*proof*⟩

lemma *strict-dominate-trans1*:
assumes *strict-dominate n1 n2*
and *dominate n2 n3*
shows *strict-dominate n1 n3*
⟨*proof*⟩

lemma *strict-dominate-trans2*:
assumes *dominate n1 n2*
and *strict-dominate n2 n3*
shows *strict-dominate n1 n3*
⟨*proof*⟩

lemma *strict-dominate-trans*:
assumes *strict-dominate n1 n2*
and *strict-dominate n2 n3*
shows *strict-dominate n1 n3*
⟨*proof*⟩

lemma *sdominate-dominate-succs*:
assumes *i-sdom-j*: *strict-dominate i j*
and *j-in-succ-k*: *j* ∈ *succs k*
shows *dominate i k*

⟨*proof*⟩

end

end

2 More auxiliary lemmas for Lists Sorted wrt <

theory *Sorted-Less2*
imports *Main HOL−Data-Structures.Cmp HOL−Data-Structures.Sorted-Less*
begin

lemma *Cons-sorted-less*: *sorted* (*rev xs*) ⟹ ∀ *x*∈*set xs*. *x* < *p* ⟹ *sorted* (*rev* (*p* # *xs*))
⟨*proof*⟩

lemma *Cons-sorted-less-nth*: ∀ *x*<*length xs*. *xs* ! *x* < *p* ⟹ *sorted* (*rev xs*) ⟹ *sorted* (*rev* (*p* # *xs*))
⟨*proof*⟩

lemma *distinct-sorted-rev*: *sorted* (*rev xs*) ⟹ *distinct xs*
⟨*proof*⟩

lemma *sorted-le2lt*:
assumes *List.sorted xs*
and *distinct xs*
shows *sorted xs*
⟨*proof*⟩

lemma *sorted-less-sorted-list-of-set*: *sorted* (*sorted-list-of-set S*)
⟨*proof*⟩

lemma *distinct-sorted*: *sorted xs* ⟹ *distinct xs*
⟨*proof*⟩

lemma *sorted-less-set-unique*:
assumes *sorted xs*
and *sorted ys*
and *set xs* = *set ys*
shows *xs* = *ys*
⟨*proof*⟩

lemma *sorted-less-rev-set-unique*:
assumes *sorted* (*rev xs*)
and *sorted* (*rev ys*)
and *set xs* = *set ys*
shows *xs* = *ys*
⟨*proof*⟩

lemma *sorted-less-set-eq*:
assumes *sorted xs*
shows *xs* = *sorted-list-of-set* (*set xs*)
⟨*proof*⟩

lemma *sorted-less-rev-set-eq*:
assumes *sorted* (*rev xs*)
shows *sorted-list-of-set* (*set xs*) = *rev xs*
⟨*proof*⟩

lemma *sorted-insort-remove1*: *sorted w* ⟹ (*insort a* (*remove1 a w*)) = *sorted-list-of-set* (*insert a* (*set w*))
⟨*proof*⟩

end

3 Operations on sorted lists

theory *Sorted-List-Operations2*
imports *Sorted-Less2*
begin

The definition and the inter_sorted_correct lemma in this theory are the same as those in Collections [2]. except the former is for a descending list while the latter is for an ascending one.

fun *inter-sorted-rev* :: *'a*::{*linorder*} *list* ⇒ *'a list* ⇒ *'a list* **where**
inter-sorted-rev [] *l2* = []
| *inter-sorted-rev l1* [] = []
| *inter-sorted-rev* (*x1* # *l1*) (*x2* # *l2*) =
(*if* (*x1* > *x2*) *then* (*inter-sorted-rev l1* (*x2* # *l2*)) *else*
(*if* (*x1* = *x2*) *then x1* # (*inter-sorted-rev l1 l2*) *else inter-sorted-rev* (*x1* # *l1*) *l2*))

lemma *inter-sorted-correct* :
assumes *l1-OK*: *sorted* (*rev l1*)
assumes *l2-OK*: *sorted* (*rev l2*)
shows *sorted* (*rev* (*inter-sorted-rev l1 l2*)) ∧ *set* (*inter-sorted-rev l1 l2*) = *set l1* ∩ *set l2*
⟨*proof*⟩

lemma *inter-sorted-rev-refl*: *inter-sorted-rev xs xs* = *xs*
⟨*proof*⟩

lemma *inter-sorted-correct-col*:
assumes *sorted* (*rev xs*)
and *sorted* (*rev ys*)
shows (*inter-sorted-rev xs ys*) = *rev* (*sorted-list-of-set* (*set xs* ∩ *set ys*))
⟨*proof*⟩

lemma *cons-set-eq*: *set* (*x* # *xs*) ∩ *set xs* = *set xs*
⟨*proof*⟩

lemma *inter-sorted-cons*: *sorted* (*rev* (*x* # *xs*)) ⟹ *inter-sorted-rev* (*x* # *xs*) *xs* = *xs*
⟨*proof*⟩

end

4 A semilattice of reversed-ordered list

theory *Dom-Semi-List*
imports *Main Jinja.Semilat Sorted-List-Operations2 Sorted-Less2 Cfg*
begin

type-synonym *node* = *nat*

context *cfg-doms*
begin

definition *nodes* :: *nat list*
where *nodes* ≡ (*g-V G*)

definition *nodes-le* :: *node list* ⇒ *node list* ⇒ *bool* **where**
nodes-le xs ys ≡ (*sorted* (*rev ys*) ∧ *sorted* (*rev xs*) ∧ (*set ys*) ⊆ (*set xs*)) ∨ *xs* = *ys*

definition *nodes-sup* ::*node list* ⇒ *node list* ⇒*node list* **where**
nodes-sup = (λ*x y*. *inter-sorted-rev x y*)

definition *nodes-semi* :: *node list sl* **where**
nodes-semi ≡ ((*rev* ∘ *sorted-list-of-set*) ‘ (*Pow* (*set* (*nodes*))), *nodes-le*, *nodes-sup*)

lemma *subset-nodes-inpow*:
assumes *sorted* (*rev xs*)
and *set xs* ⊆ *set nodes*
shows *xs* ∈ (*rev* ∘ *sorted-list-of-set*) ‘ (*Pow* (*set nodes*))
⟨*proof*⟩

lemma *nil-in-A*: [] ∈ (*rev* ∘ *sorted-list-of-set*) ‘ (*Pow* (*set nodes*))
⟨*proof*⟩

lemma *single-n-in-A*: *p* < *length nodes* ⟹ [*p*] ∈ (*rev* ∘ *sorted-list-of-set*) ‘ (*Pow* (*set nodes*))
⟨*proof*⟩

lemma *inpow-subset-nodes*:
assumes *xs* ∈ (*rev* ∘ *sorted-list-of-set*) ‘ (*Pow* (*set nodes*))

shows *set xs* ⊆ *set nodes*
⟨*proof*⟩

lemma *inter-in-pow-nodes*:
assumes *xs* ∈ (*rev* ∘ *sorted-list-of-set*) ‘ (*Pow* (*set nodes*))
shows (*rev* ∘ *sorted-list-of-set*)(*set xs* ∩ *set ys*) ∈ (*rev* ∘ (*sorted-list-of-set*)) ‘ (*Pow* (*set nodes*))
⟨*proof*⟩

lemma *nodes-le-order*: *order nodes-le* ((*rev* ∘ *sorted-list-of-set*) ‘ (*Pow* (*set nodes*)))
⟨*proof*⟩

lemma *nodes-semi-auxi*:
let A = (*rev* ∘ *sorted-list-of-set*) ‘ (*Pow* (*set* (*nodes*)));
r = *nodes-le*;
f = (λ*x y*. (*inter-sorted-rev x y*))
in semilat(*A*, *r*, *f*)
⟨*proof*⟩

lemma *nodes-semi-is-semilat*: *semilat* (*nodes-semi*)
⟨*proof*⟩

lemma *sorted-rev-subset-len-lt*:
assumes *sorted* (*rev a*)
and *sorted* (*rev b*)
and *set a* ⊂ *set b*
shows *length a* < *length b*
⟨*proof*⟩

lemma *wf-nodes-le-auxi*: *wf* {(*y*, *x*). (*sorted* (*rev y*) ∧ *sorted* (*rev x*) ∧ *set y* ⊂ *set x*) ∧ *x* ≠ *y*}
⟨*proof*⟩

lemma *wf-nodes-le-auxi2*:
wf {(*y*, *x*). *sorted* (*rev y*) ∧ *sorted* (*rev x*) ∧ *set y* ⊂ *set x* ∧ *rev x* ≠ *rev y*}
⟨*proof*⟩

lemma *wf-nodes-le*: *wf* {(*y*, *x*). *nodes-le x y* ∧ *x* ≠ *y*}
⟨*proof*⟩

lemma *acc-nodes-le*: *acc nodes-le*
⟨*proof*⟩

lemma *acc-nodes-le2*: *acc* (*fst* (*snd nodes-semi*))
⟨*proof*⟩

lemma *nodes-le-refl* [*iff*] : *nodes-le s s*
⟨*proof*⟩

end

end

5 A kildall's algorithm for computing dominators

theory *Dom-Kildall*
imports *Dom-Semi-List HOL−Library.While-Combinator Jinja.SemilatAlg*
begin

A kildall's algorithm for computing dominators. It uses the ideas and the framework of kildall's algorithm implemented in Jinja [3], and modifications are needed to make it work for a fast algorithm for computing dominators

type-synonym *state-dom = nat list*

primrec *propa* ::
's binop ⇒ (nat × 's) list ⇒ 's list ⇒ nat list ⇒ 's list ∗ nat list
where
propa f [] τs wl = (τs,wl)
| *propa f (q'# qs) τs wl = (let (q,τ) = q';*
u = (τ ⊔f τs!q);
wl' = (if u = τs!q then wl
else (insort q (remove1 q wl)))
in propa f qs (τs[q := u]) wl')

definition *iter* ::
's binop ⇒ 's step-type ⇒ 's list ⇒ nat list ⇒ 's list × nat list
where
iter f step τs w =
while (λ(τs,w). w ≠ [])
(λ(τs,w). let p = hd w
in propa f (step p (τs!p)) τs (tl w))
(τs,w)

definition *unstables* :: *state-dom ord ⇒ state-dom step-type ⇒ state-dom list ⇒ nat list*
where
unstables r step τs = sorted-list-of-set {p. p < size τs ∧ ¬ stable r step τs p}

definition *kildall* :: *state-dom ord ⇒state-dom binop ⇒ state-dom step-type ⇒ state-dom list ⇒ state-dom list* **where**
kildall r f step τs = fst(iter f step τs (unstables r step τs))

lemma *init-worklist-is-sorted*: *sorted (unstables r step τs)*
⟨*proof*⟩

context *cfg-doms*

begin

definition *transf* :: *node* ⇒ *state-dom* ⇒ *state-dom* **where**
transf n input ≡ (*n* # *input*)

definition *exec* :: *node* ⇒ *state-dom* ⇒ (*node* × *state-dom*) *list*
where *exec n xs* = *map* (λ*pc*. (*pc*, (*transf n xs*))) (*rev* (*sorted-list-of-set*(*succs n*)))

lemma *transf-res-is-rev*: *sorted* (*rev ns*) ⟹ *n* > *hd ns* ⟹ *sorted* (*rev* ((*transf n* (*ns*))))
⟨*proof*⟩

abbreviation *start* ≡ [] # (*replicate* (*length* (*g-V G*) − *1*) ((*rev*[*0*..<*length*(*g-V G*)])))

definition *dom-kildall* :: *state-dom list* ⇒ *state-dom list*
where *dom-kildall* = *kildall* (*fst* (*snd nodes-semi*)) (*snd* (*snd nodes-semi*)) *exec*

definition *dom*:: *nat* ⇒ *nat* ⇒ *bool* **where**
dom i j ≡(*let res* = (*dom-kildall start*) !*j in i* ∈ (*set res*) ∨ *i* = *j*)

lemma *dom-refl*: *dom i i*
⟨*proof*⟩

definition *strict-dom* :: *nat* ⇒ *nat* ⇒ *bool* **where**
strict-dom i j ≡ (*let res* = (*dom-kildall start*) !*j in* *i* ∈ *set res*)

lemma *strict-domI1*: (*dom-kildall* ([] # (*replicate* (*length* (*g-V G*) − *1*) ((*rev*[*0*..<*length*(*g-V G*)])))))!*j* = *res* ⟹ *i* ∈ *set res* ⟹ *strict-dom i j*
⟨*proof*⟩

lemma *strict-domD*:
strict-dom i j ⟹
dom-kildall (([] # (*replicate* (*length* (*g-V G*) − *1*) ((*rev*[*0*..<*length*(*g-V G*)])))))!*j* = *a* ⟹
i ∈ *set a*
⟨*proof*⟩

lemma *sdom-dom*: *strict-dom i j* ⟹ *dom i j*
⟨*proof*⟩

lemma *not-sdom-not-dom*: ¬*strict-dom i j* ⟹ *i* ≠ *j* ⟹ ¬*dom i j*
⟨*proof*⟩

lemma *dom-sdom*: *dom i j* ⟹ *i* ≠ *j* ⟹ *strict-dom i j*

⟨*proof*⟩

end

end

6 Properties of the kildall's algorithm on the semilattice

theory *Dom-Kildall-Property*
imports *Dom-Kildall Jinja.Listn Jinja.Kildall-1*
begin

lemma *sorted-list-len-lt*: $x \subset y \Longrightarrow$ *finite* $y \Longrightarrow$ *length* (*sorted-list-of-set* x) $<$ *length* (*sorted-list-of-set* y)
⟨*proof*⟩

lemma *wf-sorted-list*:
wf ((λ(*x*,*y*). (*sorted-list-of-set* *x*, *sorted-list-of-set* *y*)) ' *finite-psubset*)
⟨*proof*⟩

lemma *sorted-list-psub*:
sorted $w \longrightarrow$
$w \neq []$ $\longrightarrow$
(*sorted-list-of-set* (*set* (*tl* *w*)), *w*) ∈ (λ(*x*, *y*). (*sorted-list-of-set* *x*, *sorted-list-of-set* *y*)) ' {(*A*, *B*). *A* ⊂ *B* ∧ *finite* *B*}
⟨*proof*⟩

locale *dom-sl* = *cfg-doms* +
fixes *A* **and** *r* **and** *f* **and** *step* **and** *start* **and** *n*
defines *A* ≡ ((*rev* ∘ *sorted-list-of-set*) ' (*Pow* (*set* (*nodes*))))
defines *r* ≡ *nodes-le*
defines *f* ≡ *nodes-sup*
defines *n* ≡ (*size* *nodes*)
defines *step* ≡ *exec*
defines *start* ≡ ([] # (*replicate* (*length* (*g-V* *G*) − *1*) ((*rev*[*0*..<*n*])))):: *state-dom list*

begin

lemma *is-semi*: *semilat*(*A*,*r*,*f*)
⟨*proof*⟩
lemma *Cons-less-Conss* [*simp*]:
x#*xs* [⊏$_r$] *y*#*ys* = (*x* ⊏$_r$ *y* ∧ *xs* [⊑$_r$] *ys* ∨ *x* = *y* ∧ *xs* [⊏$_r$] *ys*)
⟨*proof*⟩

lemma *acc-le-listI* [*intro*!]:

acc r $\Longrightarrow$ *acc* (*Listn.le r*)
⟨*proof*⟩

lemma *wf-listn*: *wf* {(*y*,*x*). *x* $\sqsubset_{Listn.le\ r}$ *y*}
⟨*proof*⟩

lemma *wf-listn'*: *wf* {(*y*,*x*). *x* [$\sqsubset_r$] *y*}
⟨*proof*⟩

lemma *wf-listn-termination-rel*:
wf ({(*y*,*x*). *x* $\sqsubset_{Listn.le\ r}$ *y*} <∗*lex*∗> (λ(*x*, *y*). (*sorted-list-of-set x*, *sorted-list-of-set y*)) ‘ *finite-psubset*)
⟨*proof*⟩

lemma *inA-is-sorted*: *xs* ∈ *A* $\Longrightarrow$ *sorted* (*rev xs*)
⟨*proof*⟩

lemma *list-nA-lt-refl*: *xs* ∈ *nlists n A* $\longrightarrow$ *xs* [$\sqsubseteq_r$] *xs*
⟨*proof*⟩

lemma *nil-inA*: [] ∈ *A*
⟨*proof*⟩

lemma *upt-n-in-pow-nodes*: {*0*..<*n*} ∈ *Pow* (*set nodes*)
⟨*proof*⟩

lemma *rev-all-inA*: *rev* [*0*..<*n*] ∈ *A*
⟨*proof*⟩

lemma *len-start-is-n*: *length start* = *n*
⟨*proof*⟩

lemma *len-start-is-len-verts*: *length start* = *length* (*g-V G*)
⟨*proof*⟩

lemma *start-len-gt-0*: *length start* > *0*
⟨*proof*⟩

lemma *start-subset-A*: *set start* ⊆ *A*
⟨*proof*⟩

lemma *start-in-A* : *start* ∈ (*nlists n A*)
⟨*proof*⟩

lemma *sorted-start-nth*: *i* < *n* $\Longrightarrow$ *sorted* (*rev* (*start*!*i*))
⟨*proof*⟩

lemma *start-nth0-empty*: *start*!*0* = []
⟨*proof*⟩

lemma *start-nth-lt0-all*: $\forall\, p{\in}\{1..<\ length\ start\}.\ start!p = (rev\ [0..<n])$
⟨*proof*⟩

lemma *in-nodes-lt-n*: $x \in set\ (g\text{-}V\ G) \Longrightarrow x < n$
⟨*proof*⟩

lemma *start-nth0-unstable-auxi*: $\neg\ [0] \sqsubseteq_r (rev\ [0..<n])$
⟨*proof*⟩

lemma *start-nth0-unstable*: $\neg$ *stable r step start 0*
⟨*proof*⟩

lemma *start-nth-unstable*:
assumes $p \in \{1\ ..<\ length\ (g\text{-}V\ G)\}$
and $succs\ p \neq \{\}$
shows $\neg$ *stable r step start p*
⟨*proof*⟩

lemma *start-unstable-cond*:
assumes $succs\ p \neq \{\}$
and $p < length\ (g\text{-}V\ G)$
shows $\neg$ *stable r step start p*
⟨*proof*⟩

lemma *unstable-start*: $unstables\ r\ step\ start = sorted\text{-}list\text{-}of\text{-}set\ (\{p.\ succs\ p \neq \{\}$
$\wedge\ p < length\ start\})$
⟨*proof*⟩

end

declare *sorted-list-of-set-insert-remove*[*simp del*]

context *dom-sl*
begin

lemma (**in** *dom-sl*) *decomp-propa*: $\bigwedge ss\ w.$
$(\forall\,(q,t){\in}set\ qs.\ q < size\ ss \wedge t \in A\ ) \Longrightarrow$
$sorted\ w \Longrightarrow$
$set\ ss \subseteq A \Longrightarrow$
$propa\ f\ qs\ ss\ w = (merges\ f\ qs\ ss, (sorted\text{-}list\text{-}of\text{-}set\ (\{q.\ \exists\, t.(q,t){\in}set\ qs \wedge t \sqcup_f$
$(ss!q) \neq ss!q\} \cup set\ w)))$⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*
lemma (**in** *Semilat*)*list-update-le-listI* [*rule-format*]:
$set\ xs \subseteq A \longrightarrow set\ ys \subseteq A \longrightarrow xs\ [\sqsubseteq_r]\ ys \longrightarrow p < size\ xs \longrightarrow$
$x \sqsubseteq_r ys!p \longrightarrow x{\in}A \longrightarrow$
$xs[p := x \sqcup_f xs!p]\ [\sqsubseteq_r]\ ys$⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*pro*

7 Soundness and completeness

theory *Dom-Kildall-Correct*
imports *Dom-Kildall-Property*
begin

context *dom-sl*
begin

lemma *entry-dominate-dom*:
  assumes $i \in set\ (g\text{-}V\ G)$
    and *dominate i 0*
  shows *dom i 0*
  ⟨*proof*⟩

lemma *path-entry-dom*:
  fixes *pa i d*
  assumes *path-entry* $(g\text{-}E\ G)$ *pa i*
    and *dom d i*
  shows $d \in set\ pa \vee d = i$
  ⟨*proof*⟩

lemma *dom-sound*: *dom i j* $\Longrightarrow$ *dominate i j*
  ⟨*proof*⟩

lemma *sdom-sound*: *strict-dom i j* $\Longrightarrow$ $j \in set\ (g\text{-}V\ G)$ $\Longrightarrow$ *strict-dominate i j*
⟨*proof*⟩

lemma *dom-complete-auxi*: $i < length\ start \Longrightarrow (dom\text{-}kildall\ start)!i = ss' \wedge k \notin set\ ss' \Longrightarrow non\text{-}strict\text{-}dominate\ k\ i$
⟨*proof*⟩

lemma *notsdom-notsdominate*: $\neg$ *strict-dom i j* $\Longrightarrow j < length\ start \Longrightarrow$ *non-strict-dominate i j*
⟨*proof*⟩

lemma *notsdom-notsdominate'*: $\neg$ *strict-dom i j* $\Longrightarrow j < length\ start \Longrightarrow \neg$ *strict-dominate i j*
  ⟨*proof*⟩

lemma *dom-complete*: *strict-dominate i j* $\Longrightarrow j < size\ start \Longrightarrow$ *strict-dom i j*
  ⟨*proof*⟩

end

end

References

[1] K. D. Cooper, T. J. Harvey, and K. Kennedy. A simple, fast dominance algorithm. Technical report, Rice University, Houston, Jan. 2006. https://scholarship.rice.edu/handle/1911/96345.

[2] P. Lammich. Operations on sorted lists. 2009. https://www.isa-afp.org/browser_info/current/AFP/Collections/Sorted_List_Operations.html.

[3] T. Nipkow and G. Klein. Operations on sorted lists. 2000. https://www.isa-afp.org/browser_info/current/AFP/Jinja/Kildall.html.