graph-verification

By christine

April 13, 2015

Contents

theory *Connected-Components*
imports *../Graph-Theory/Graph-Theory*
begin

locale *connected-components-locale* =
  fin-digraph +
  fixes *num* :: $'a \Rightarrow nat$
  fixes *parent-edge* :: $'a \Rightarrow 'b$ *option*
  fixes *r* :: $'a$
  assumes *r-assms*: $r \in verts\ G \wedge parent\text{-}edge\ r = None \wedge num\ r = 0$
  assumes *parent-num-assms*:
    $\bigwedge v.\ v \in verts\ G \wedge v \neq r \Longrightarrow$
      $\exists\, e \in arcs\ G.$
        $parent\text{-}edge\ v = Some\ e \wedge$
        $head\ G\ e = v \wedge$
        $num\ v = num\ (tail\ G\ e) + 1$

sublocale *connected-components-locale* ⊆ *fin-digraph G*
  by *auto*

context *connected-components-locale*
begin

lemma *ccl-wellformed*: *wf-digraph G*
  by *unfold-locales*

lemma *num-r-is-min*:
  assumes $v \in verts\ G$
  assumes $v \neq r$
  shows $num\ v > 0$
  using *parent-num-assms assms*
  by *fastforce*

lemma *path-from-root*:
  fixes $v :: 'a$
  assumes $v \in verts\ G$

shows $r \rightarrow^* v$
using *assms*
proof (*induct num v arbitrary*: *v*)
case *0*
hence $v = r$ **using** *num-r-is-min* **by** *fastforce*
with ‹$v \in$ *verts G*› **show** *?case* **by** *auto*
next
case (*Suc n'*)
hence $v \neq r$ **using** *r-assms* **by** *auto*
then obtain *e* **where** *ee*:
$e \in$ *arcs G*
head G e $= v \wedge$ *num v* $=$ *num* (*tail G e*) $+ 1$
using *Suc parent-num-assms* **by** *blast*
with ‹$v \in$ *verts G*› *Suc*(*1*,*2*) *tail-in-verts*
have $r \rightarrow^*$ (*tail G e*) *tail G e* $\rightarrow v$
by (*auto intro*: *in-arcs-imp-in-arcs-ends*)
then show *?case* **by** (*rule reachable-adj-trans*)
qed

The underlying undirected, simple graph is connected

lemma *connectedG*: *connected G*
proof (*unfold connected-def*, *intro strongly-connectedI*)
show *verts* (*with-proj* (*mk-symmetric G*)) $\neq \{\}$
by (*metis equals0D r-assms reachable-in-vertsE reachable-mk-symmetricI reachable-refl*)
next
let *?SG* = *mk-symmetric G*
interpret *S*: *pair-fin-digraph ?SG* **..**
fix *u v* **assume** *uv-sG*: $u \in$ *verts ?SG* $v \in$ *verts ?SG*
from *uv-sG* **have** $u \in$ *verts G* $v \in$ *verts G* **by** *auto*
then have $u \rightarrow^*_{?SG} r$ $r \rightarrow^*_{?SG} v$
by (*auto intro*: *reachable-mk-symmetricI path-from-root symmetric-reachable symmetric-mk-symmetric simp del*: *pverts-mk-symmetric*)
then show $u \rightarrow^*_{?SG} v$
by (*rule S.reachable-trans*)
qed

theorem *connected-by-path*:
fixes *u v* :: *'a*
assumes $u \in$ *pverts* (*mk-symmetric G*)
assumes $v \in$ *pverts* (*mk-symmetric G*)
shows $u \rightarrow^*_{mk\text{-}symmetric\ G}\ v$
using *connectedG wellformed-mk-symmetric assms*
unfolding *connected-def strongly-connected-def* **by** *fastforce*
end

corollary (**in** *connected-components-locale*) *connected-graph*:
assumes $u \in$ *verts G* **and** $v \in$ *verts G*
shows $\exists p.$ *vpath p* (*mk-symmetric G*) $\wedge$ *hd p* $= u \wedge$ *last p* $= v$

```
proof -
  interpret S: pair-fin-digraph mk-symmetric G ..
  show ?thesis unfolding S.reachable-vpath-conv[symmetric]
    using assms by (auto intro: connected-by-path)
qed

end

theory Check-Connected
imports
  ../Library/Autocorres-Misc
  ../Witness-Property/Connected-Components
begin

install-C-file check-connected.c

autocorres check-connected.c

context check-connected begin



lemma validNFE-getsE[wp]:
  ⦃λs. P (f s) s⦄ getsE f ⦃P⦄, ⦃E⦄!
  by (auto simp: getsE-def) wp

lemma validNFE-guardE[wp]:
  ⦃λs. f s ∧ P () s⦄ guardE f ⦃P⦄, ⦃Q⦄!
  by (auto simp: guardE-def, wp, linarith)


lemma eq-of-nat-conv:
  assumes unat w1 = n
  shows w2 = of-nat n ⟷ w2 = w1
  using assms by auto


lemma less-unat-plus1:
  assumes a < unat (b + 1)
  shows a < unat b ∨ a = unat b
  apply (subgoal-tac  b + 1 ≠ 0 )
  using assms unat-minus-one add-diff-cancel
  by fastforce+

lemma unat-minus-plus1-less:
  fixes a b
  assumes a < b
  shows unat (b - (a + 1)) < unat (b - a)
 by (metis (no-types) ab-semigroup-add-class.add-ac(1) right-minus-eq measure-unat
```

add-diff-cancel2 assms is-num-normalize(1) zadd-diff-inverse linorder-neq-iff)

lemma *unat-image-upto*:
  fixes n :: *32 word*
  shows $unat$ ‘ $\{0..<n\} = \{unat\ 0..<unat\ n\}$ (**is** $?A = ?B$)
proof
  show $?B \subseteq ?A$
  proof
    fix i **assume** a: $i \in ?B$
    then obtain i':: *32 word* **where** ii: $i = unat\ i'$
      by (*metis ex-nat-less-eq le-unat-uoi not-leE order-less-asym unat-0*)
    then have $i' \in \{0..<n\}$
      by (*metis* (*hide-lams*, *mono-tags*) *atLeast0LessThan a unat-0 word-zero-le lessThan-iff not-leE not-less-iff-gr-or-eq order-antisym word-le-nat-alt Un-iff ivl-disj-un*(*8*))
    thus $i \in ?A$ **using** *ii* **by** *fast*
  qed
next
  show $?A \subseteq ?B$
  proof
    fix i **assume** a: $i \in ?A$
    then obtain i':: *32 word* **where** ii: $i = unat\ i'$ **by** *blast*
    then have $i' \in \{0..<n\}$ **using** a **by** *force*
    thus $i \in ?B$
      by (*metis Un-iff atLeast0LessThan ii ivl-disj-un*(*8*) *lessThan-iff unat-0 unat-mono word-zero-le*)
  qed
qed

type-synonym *IVertex* = *32 word*
type-synonym *IEdge-Id* = *32 word*
type-synonym *IEdge* = *IVertex* × *IVertex*
type-synonym *IPEdge* = *IVertex* ⇒ *32 word*
type-synonym *INum* = *IVertex* ⇒ *32 word*
type-synonym *IGraph* = *32 word* × *32 word* × (*IEdge-Id* ⇒ *IEdge*)

abbreviation
  ivertex-cnt :: *IGraph* ⇒ *32 word*
where
  ivertex-cnt G ≡ *fst G*

abbreviation
  iedge-cnt :: *IGraph* ⇒ *32 word*
where
  iedge-cnt G ≡ *fst* (*snd G*)

abbreviation
  iedges :: *IGraph* ⇒ *IEdge-Id* ⇒ *IEdge*

where
  iedges G $\equiv$ *snd* (*snd G*)

fun
  bool::32 word $\Rightarrow$ *bool*
where
  bool b = (*if b=0 then False else True*)

fun
  mk-list′ :: *nat* $\Rightarrow$ (*32 word* $\Rightarrow$ *′b*) $\Rightarrow$ *′b list*
where
  mk-list′ n f = *map f* (*map of-nat* [*0..<n*])

fun
  mk-list′-temp :: *nat* $\Rightarrow$ (*32 word* $\Rightarrow$ *′b*) $\Rightarrow$ *nat* $\Rightarrow$ *′b list*
where
  mk-list′-temp 0 - - = [] |
  mk-list′-temp (*Suc x*) *f i* = (*f* (*of-nat i*)) # *mk-list′-temp x f* (*Suc i*)

fun
  mk-iedge-list :: *IGraph* $\Rightarrow$ *IEdge list*
where
  mk-iedge-list G = *mk-list′* (*unat* (*iedge-cnt G*)) (*iedges G*)

fun
  mk-inum-list :: *IGraph* $\Rightarrow$ *INum* $\Rightarrow$ *32 word list*
where
  mk-inum-list G num = *mk-list′* (*unat* (*ivertex-cnt G*)) *num*

fun
  mk-ipedge-list :: *IGraph* $\Rightarrow$ *IPEdge* $\Rightarrow$ *32 word list*
where
  mk-ipedge-list G pedge = *mk-list′* (*unat* (*ivertex-cnt G*)) *pedge*

fun
  to-edge :: *IEdge* $\Rightarrow$ *Edge-C*
where
  to-edge (*u,v*) = *Edge-C u v*

lemma *s-C-pte*[*simp*]:
  s-C (*to-edge e*) = *fst e*
  by (*cases e*) *auto*

lemma *t-C-pte*[*simp*]:
  t-C (*to-edge e*) = *snd e*
  by (*cases e*) *auto*

definition *is-graph* **where**
is-graph h iG p ≡
is-valid-Graph-C h p ∧
ivertex-cnt iG = *n-C* (*heap-Graph-C h p*) ∧
iedge-cnt iG = *m-C* (*heap-Graph-C h p*) ∧
arrlist (*heap-Edge-C h*) (*is-valid-Edge-C h*)
(*map to-edge* (*mk-iedge-list iG*)) (*es-C* (*heap-Graph-C h p*))

definition
is-numm h iG iN p ≡ *arrlist* (*heap-w32 h*) (*is-valid-w32 h*) (*mk-inum-list iG iN*) *p*

definition
is-pedge h iG iP (*p*:: *32 signed word ptr*) ≡ *arrlist* (λ*p*. *heap-w32 h* (*ptr-coerce p*))
(λ*p*. *is-valid-w32 h* (*ptr-coerce p*)) (*mk-ipedge-list iG iP*) *p*

lemma *sint-ucast*:
sint (*ucast* (*x* ::*word32*) :: *sword32*) = *sint x*
by (*clarsimp simp*: *sint-uint uint-up-ucast is-up*)

definition
is-root :: *IGraph* ⇒ *IVertex* ⇒ *IPEdge* ⇒ *INum* ⇒ *bool*
where
is-root iG r iP iN ≡ *r* < (*ivertex-cnt iG*) ∧ (*iN r* = *0*) ∧ (*sint* (*iP r*) < *0*)

definition
parent-num-assms-inv :: *IGraph* ⇒ *IVertex* ⇒ *IPEdge* ⇒ *INum* ⇒ *nat* ⇒ *bool*
where
parent-num-assms-inv G r p n k ≡
∀ *i* < *k*. (*of-nat i*) ≠ *r* ⟶
0 ≤ *sint* (*p* (*of-nat i*)) ∧
((*p* (*of-nat i*)) < *iedge-cnt G* ∧
snd (*iedges G* (*p* (*of-nat i*))) = (*of-nat i*) ∧
n (*of-nat i*) = *n* (*fst* (*iedges G* (*p* (*of-nat i*)))) + *1*) ∧
n (*of-nat i*) < *ivertex-cnt G*

function (**in** *connected-components-locale*)
pwalk :: *'a* ⇒ *'a list*
where
pwalk v =
(*if* (*v* = *r* ∨ *v* ∉ *verts G*)
then [*v*]
else
pwalk (*tail G* (*the* (*parent-edge v*))) ⊕ [*tail G* (*the* (*parent-edge v*)), *v*])
by *simp*+
termination (**in** *connected-components-locale*)
using *parent-num-assms*

by (*relation measure num*, *auto*, *fastforce*)

lemma (**in** *connected-components-locale*) *pwalk-simps*:
$v = r \vee v \notin verts\ G \Longrightarrow pwalk\ v = [v]$
$v \neq r \Longrightarrow v \in verts\ G \Longrightarrow pwalk\ v =$
$pwalk\ (tail\ G\ (the\ (parent\text{-}edge\ v)))\ @\ [v]$
by (*simp*, *metis drop-0 pwalk.simps*
drop-Suc-Cons vwalk-join-def drop-Suc)

lemma (**in** *connected-components-locale*) *pwalk-ne*: $pwalk\ v \neq []$
by (*metis drop-0 drop-Suc drop-Suc-Cons not-Cons-self*
pwalk.simps snoc-eq-iff-butlast vwalk-join-def)

lemma (**in** *connected-components-locale*) *vwalk-length-pwalk*:
assumes $v \in verts\ G$
assumes $v \neq r$
shows $vwalk\text{-}length\ (pwalk\ v) =$
$vwalk\text{-}length\ (pwalk\ (tail\ G\ (the\ (parent\text{-}edge\ v)))) + 1$
by (*smt append-Cons assms length-append length-tl list.size*(*3*,*4*) *pwalk-ne*
pwalk.simps tl-append2 vwalk-join-Cons vwalk-join-def vwalk-length-simp)

lemma (**in** *connected-components-locale*) *pwalk-split*:
assumes $x \in set\ (pwalk\ v)$
shows $\exists p.\ pwalk\ v = pwalk\ x\ @\ p$
using *assms*
proof (*induct vwalk-length* (*pwalk v*) *arbitrary*: *v*)
case (*Suc n*)
have *vnr*: $v \neq r$
using *Suc*(*2*) **by** *fastforce*
show *?case*
proof (*cases* $v \in verts\ G$)
case *True*
thus *?thesis*
proof (*cases* $x = v$)
case *False*
let $?u = tail\ G\ (the\ (parent\text{-}edge\ v))$
have *xpu*: $x \in set\ (pwalk\ ?u)$
using *Suc*(*3*) *pwalk-simps*(*2*)[*OF vnr True*] *False* **by** *fastforce*
hence $\exists p.\ pwalk\ (tail\ G\ (the\ (parent\text{-}edge\ v))) = pwalk\ x\ @\ p$
using *vwalk-length-pwalk*[*OF True vnr*] *Suc*(*2*)
by (*metis Suc*(*1*)[*OF - xpu*] *Suc-eq-plus1*
Suc-eq-plus1-left diff-add-inverse)
thus *?thesis* **using** *pwalk-simps*(*2*)[*OF vnr True*] **by** *fastforce*
qed *fast*
qed (*metis Suc.prems append-Nil2 empty-iff empty-set pwalk-simps*(*1*) *set-ConsD*)
qed (*metis pwalk-simps*(*1*) *add-is-0 vwalk-length-pwalk*
append-Nil2 empty-iff empty-set one-neq-zero set-ConsD)

lemma (**in** *connected-components-locale*) *path-from-root-num*:

```
  fixes v :: 'a
  assumes v ∈ verts G
  shows vpath (pwalk v) G ∧
        hd (pwalk v) = r ∧
        last (pwalk v) = v ∧
        num v = vwalk-length (pwalk v)
using assms
proof (induct vwalk-length (pwalk v) arbitrary: v rule: less-induct)
case less
  thus ?case
  proof (cases v=r)
    case True
      thus ?thesis using r-assms unfolding vpath-def by force
  next
    case False
      then obtain e where ee:
        e ∈ arcs G
        e = the (parent-edge v)
        head G e = v ∧ num v = num (tail G e) + 1
        using less.prems parent-num-assms by force
      let ?te = tail G e
      let ?p' = pwalk ?te
      let ?q = [?te, v]
      obtain p where
        pp: p = ?p' ⊕ ?q
        by presburger
      hence pv: p = pwalk v
        using less.prems False ee(2) by force
      have ew: vwalk ?q G unfolding vwalk-def
        using ee(3) in-arcs-imp-in-arcs-ends[OF ee(1)]
              less.prems  tail-in-verts[OF ee(1)]
        by auto
      have wlp: vwalk-length ?p' < vwalk-length (pwalk v)
        using vwalk-length-pwalk[OF less.prems False] ee(2)
        by presburger
      hence pp':
        vwalk ?p' G
        distinct ?p'
        hd ?p' = r
        last ?p' = ?te
        num ?te = vwalk-length ?p'
        using less.hyps[where v=?te,
              OF - tail-in-verts[OF ee(1)]]
        unfolding vpath-def by linarith+
      have jp: joinable ?p' ?q
         unfolding  joinable-def
         by (simp only: pp'(4) pp'(1)[unfolded vwalk-def], simp)
      have vwalk-length [tail G e, v] = 1 by force
      hence np: num v = vwalk-length p
```

```
      using pp vwalk-join-vwalk-length[OF jp] ee pp′(5)
      by (simp only: pp vwalk-join-vwalk-length[OF jp] ee pp′(5))
    have wp: vwalk p G
      by (metis pp ew pp′(1) jp vwalk-joinI-vwalk)
    {
      fix x assume xp: x ∈ set ?p′
      have vwalk-length (pwalk x) ≤ vwalk-length ?p′
      using pwalk-split[OF xp] by (smt length-append vwalk-length-simp)
      then have wlx: vwalk-length (pwalk x) < vwalk-length (pwalk v)
        using wlp by linarith
      hence num x = vwalk-length (pwalk x)
        using pp′(1) less.hyps[OF wlx] xp vwalk-verts-in-verts by blast
      with wlx have num x < vwalk-length (pwalk v) by presburger
    }
  then have v ∉ set ?p′ using wlp np pv  by (metis less-not-refl)
  hence dp: distinct p
    by (metis butlast-snoc distinct.simps(2) distinct1-rotate pp pp′(2)
      list.simps(2) rotate1.simps(2) rotate1-hd-tl vwalk-join-def)
  hence vpath p G ∧ hd p = r ∧ last p = v ∧
        num v = vwalk-length p
    using dp wp np pp′ pp
    by (metis hd-append2 last-snoc list.sel(3) pwalk-ne vpathI vwalk-join-def)
  thus ?thesis using pv by fast
 qed
qed


definition
  no-loops :: ('a, 'b) pre-digraph ⇒ bool
where
  no-loops G ≡ ∀ e ∈ arcs G. tail G e ≠ head G e

definition
  abs-IGraph :: IGraph ⇒ (32 word, 32 word) pre-digraph
where
  abs-IGraph G ≡ (| verts = {0..<ivertex-cnt G}, arcs = {0..<iedge-cnt G},
    tail = fst o iedges G, head = snd o iedges G |)

lemma verts-absI[simp]: verts (abs-IGraph G) = {0..<ivertex-cnt G}
  and edges-absI[simp]: arcs (abs-IGraph G) = {0..<iedge-cnt G}
  and start-absI[simp]: tail (abs-IGraph G) e = fst (iedges G e)
  and target-absI[simp]: head (abs-IGraph G) e = snd (iedges G e)
  by (auto simp: abs-IGraph-def)

definition
  abs-pedge :: (32 word ⇒ 32 word) ⇒ 32 word ⇒ 32 word option
where
  abs-pedge p ≡ (λv. if sint (p v) < 0 then None else Some (p v))
```

lemma *None-abs-pedgeI*[*simp*]:
  ((*abs-pedge p*) *v* = *None*) = (*sint* (*p v*) < *0*)
  using *abs-pedge-def* **by** *auto*

lemma *Some-abs-pedgeI*[*simp*]:
  ($\exists$ *e*. (*abs-pedge p*) *v* = *Some e*) = (*sint* (*p v*) $\geq$ *0*)
  using *None-not-eq None-abs-pedgeI*
  by (*metis abs-pedge-def linorder-not-le option.simps*(*3*))

lemma *wellformed-iGraph*:
  assumes *wf-digraph* (*abs-IGraph G*)
  shows $\bigwedge$*e*. *e* < *iedge-cnt G* $\Longrightarrow$
      fst (*iedges G e*) < *ivertex-cnt G* $\wedge$
      snd (*iedges G e*) < *ivertex-cnt G*
using *assms* **unfolding** *wf-digraph-def* **by** *simp*

lemma *path-length*:
  assumes *vpath p* (*abs-IGraph iG*)
  shows *vwalk-length p* < *unat* (*ivertex-cnt iG*)
proof −
  have *pne*: *p* $\neq$ [] **and** *dp*: *distinct p* **using** *assms* **by** *fast*+
  have *unat* (*ivertex-cnt iG*) = *card* (*unat* ‘ {*0*..<(*fst iG*)})
    using *unat-image-upto* **by** *simp*
  then have *unat* (*ivertex-cnt iG*) = *card* ((*verts* (*abs-IGraph iG*)))
    by (*simp add*: *inj-on-def card-image*)
  hence *length p* $\leq$ *unat* (*ivertex-cnt iG*)
     by (*metis finite-code card-mono vwalk-def*
       distinct-card[*OF dp*] *vpath-def assms*)
  hence *length p* − *1* < *unat* (*ivertex-cnt iG*)
    by (*metis pne Nat.diff-le-self le-neq-implies-less*
      less-imp-diff-less minus-eq one-neq-zero length-0-conv)
  thus *vwalk-length p* < *unat* (*fst iG*)
    using *assms*
    unfolding *vpath-def vwalk-def* **by** *simp*
qed

lemma *ptr-coerce-ptr-add-uint*[*simp*]:
  ptr-coerce ($p +_p$ *uint x*) = $p +_p$ (*uint x*)
  by *auto*

lemma *check-r′-spc*:
  is-graph s iG p $\Longrightarrow$
   is-numm s iG iN p′ $\Longrightarrow$
   is-pedge s iG iP p″ $\Longrightarrow$
   check-r′ p r p″ p′ s =
   Some (*if is-root iG r iP iN then 1 else 0*)

unfolding *check-r′-def* **unfolding** *is-graph-def is-numm-def is-pedge-def*
apply (*simp add*: *ocondition-def oguard-def ogets-def oreturn-def obind-def*)
apply (*simp add*: *is-root-def uint-nat word-less-def sint-ucast*)
apply (*safe*, *simp-all add*: *arrlist-nth*)
   apply (*fastforce simp*: *dest*:*arrlist-nth-value*[**where** *i*=*int* (*unat r*)])
  apply (*fastforce dest*:*arrlist-nth-valid*[**where** *i*=*int* (*unat r*)])
 apply (*fastforce dest*:*arrlist-nth-value*[**where** *i*=*int* (*unat r*)])
apply (*fastforce dest*:*arrlist-nth-valid*[**where** *i*=*int* (*unat r*)])
done

lemma *pedge-num-heap*:
⟦*arrlist* (*λp. heap-w32 h* (*ptr-coerce p*)) (*λp. is-valid-w32 h* (*ptr-coerce p*))
(*map* (*iL* ∘ *of-nat*) [*0*..<*unat n*]) *l*; *i* < *n*⟧ ⟹
 iL i = *heap-w32 h* (*l* $+_p$ *int* (*unat i*))
apply (*subgoal-tac*
heap-w32 h (*l* $+_p$ *int* (*unat i*)) = *map* (*iL* ∘ *of-nat*) [*0*..<*unat n*] ! *unat i*)
 apply (*subgoal-tac map* (*iL* ∘ *of-nat*) [*0*..<*unat n*] ! *unat i* = *iL i*)
  apply *fastforce*
 apply (*metis* (*hide-lams*, *mono-tags*) *unat-mono word-unat.Rep-inverse*
  minus-nat.diff-0 nth-map-upt o-apply plus-nat.add-0)
apply (*simp add*: *arrlist-nth-value unat-mono*)
done

lemma *pedge-num-heap-ptr-coerce*:
⟦*arrlist* (*λp. heap-w32 h* (*ptr-coerce p*)) (*λp. is-valid-w32 h* (*ptr-coerce p*))
(*map* (*iL* ∘ *of-nat*) [*0*..<*unat n*]) *l*; *i* < *n*; *0* ≤ *i*⟧ ⟹
 iL i = *heap-w32 h* (*ptr-coerce* (*l* $+_p$ *int* (*unat i*)))
apply (*subgoal-tac*
heap-w32 h (*ptr-coerce* (*l* $+_p$ *int* (*unat i*))) = *map* (*iL* ∘ *of-nat*) [*0*..<*unat n*] !
unat i)
 apply (*subgoal-tac map* (*iL* ∘ *of-nat*) [*0*..<*unat n*] ! *unat i* = *iL i*)
  apply *fastforce*
 apply (*metis* (*hide-lams*, *mono-tags*) *unat-mono word-unat.Rep-inverse*
  minus-nat.diff-0 nth-map-upt o-apply plus-nat.add-0)
apply (*drule arrlist-nth-value*[**where** *i*=*int* (*unat i*)], (*simp add*:*unat-mono*)+)
done

lemma *edge-heap*:
⟦ *arrlist h v* (*map* (*to-edge* ∘ (*iedges iG* ∘ *of-nat*)) [*0*..<*unat m*]) *ep*;
e < *m*⟧ ⟹ *to-edge* ((*iedges iG*) *e*) = *h* (*ep* $+_p$ (*int* (*unat e*)))
apply (*subgoal-tac h* (*ep* $+_p$ (*int* (*unat e*))) =
(*map* (*to-edge* ∘ (*iedges iG* ∘ *of-nat*)) [*0*..<*unat m*]) ! *unat e*)
 apply (*subgoal-tac to-edge* ((*iedges iG*) *e*) =
 (*map* (*to-edge* ∘ (*iedges iG* ∘ *of-nat*)) [*0*..<*unat m*]) ! *unat e*)
  apply *presburger*
 apply (*metis* (*hide-lams*, *mono-tags*) *length-map length-upt o-apply*
   map-upt-eq-vals-D minus-nat.diff-0 unat-mono word-unat.Rep-inverse)

```
  apply (fastforce simp: unat-mono arrlist-nth-value)
  done

lemma head-heap:
  ⟦arrlist h v (map (to-edge ∘ (iedges iG ∘ of-nat)) [0..<unat m]) ep; e < m⟧ ⟹
  snd ((iedges iG) e) = t-C (h (ep +p (uint e)))
  using edge-heap to-edge.simps t-C-pte by (metis uint-nat)

lemma tail-heap:
  ⟦arrlist h v (map (to-edge ∘ (iedges iG ∘ of-nat)) [0..<unat m]) ep; e < m⟧ ⟹
  fst ((iedges iG) e) =  s-C (h (ep +p  (uint e)))
  using edge-heap to-edge.simps s-C-pte uint-nat by metis


lemma check-parent-num-spc':
  ⦃ P and
    (λs. wf-digraph (abs-IGraph iG) ∧
         is-graph s iG g ∧
         is-numm s iG iN n ∧
         is-pedge s iG iP p ∧
         r < ivertex-cnt iG)⦄
  check-parent-num' g r p n
  ⦃ (λ- s. P s) And
    (λrr s. rr ≠ 0 ⟷ parent-num-assms-inv iG r iP iN (unat (ivertex-cnt iG)))
⦄!
  apply (clarsimp simp: check-parent-num'-def)
  apply (subst whileLoopE-add-inv[where
        M=λ(vv, s). unat (ivertex-cnt iG − vv) and
        I=λvv s. P s ∧ parent-num-assms-inv iG r iP iN (unat vv) ∧
          vv ≤ ivertex-cnt iG ∧
          wf-digraph (abs-IGraph iG) ∧
          is-graph s iG g ∧ is-numm s iG iN n ∧
          is-pedge s iG iP p ∧
          r < ivertex-cnt iG])
  apply (simp add: skipE-def)
  apply wp
    unfolding is-graph-def is-numm-def is-pedge-def parent-num-assms-inv-def
    apply (subst if-bool-eq-conj)+
    apply (simp split: split-if-asm, safe, simp-all add: arrlist-nth)
                         apply (rule-tac i= (uint vv) in arrlist-nth-valid, simp+)
                         apply (metis uint-nat word-less-def)
                        apply (rule-tac x=unat vv in exI)
                        apply (subgoal-tac n-C (heap-Graph-C s g) ≤ iN vv)
                         apply (metis (hide-lams) word-less-nat-alt
                         word-not-le word-unat.Rep-inverse)
                        apply (subst pedge-num-heap[where l=n and iL=iN])
                          apply simp
                         apply simp
                        apply (metis uint-nat)
```

apply (*rule-tac i=* (*uint vv*) **in** *arrlist-nth-valid*)
apply *simp+*
apply (*metis uint-nat word-less-def*)
apply (*rule-tac x=unat vv* **in** *exI*)
apply (*rule conjI*, *metis unat-mono*, *simp*)
apply (*metis sint-ucast not-le uint-nat pedge-num-heap-ptr-coerce word-zero-le*)
apply (*rule-tac x=unat vv* **in** *exI*)
apply (*rule conjI*, *metis unat-mono*, *simp*)
apply (*metis not-le uint-nat pedge-num-heap-ptr-coerce word-zero-le*)
apply (*rule-tac x=unat vv* **in** *exI*)
apply (*rule conjI*, *metis unat-mono*, *simp*)
apply (*subgoal-tac snd* (*snd* (*snd iG*) (*iP vv*)) = *t-C* (*heap-Edge-C s* (*es-C* (*heap-Graph-C s g*) $+_p$ *uint* (*iP vv*))))
apply (*metis uint-nat pedge-num-heap-ptr-coerce word-zero-le*)
apply (*subst head-heap*[**where** *iG=iG*], *simp*)
apply (*metis not-le uint-nat pedge-num-heap-ptr-coerce word-zero-le*)
apply *simp*
apply (*rule-tac x=unat vv* **in** *exI*)
apply (*rule conjI*, *metis unat-mono*, *simp*,*clarsimp*)
apply (*subgoal-tac iN vv* $\neq$ *iN* (*fst* (*snd* (*snd iG*) (*iP vv*))) + *1*)
apply *fast*
apply (*subst pedge-num-heap*[**where** *l=n* **and** *iL=iN*])
apply *simp+*
apply (*subst pedge-num-heap*[**where** *l=n* **and** *iL=iN*])
apply *simp*
apply (*drule wellformed-iGraph*[**where** *G=iG*])
apply *simp+*
apply (*subst tail-heap*[**where** *iG=iG*], *simp+*)
apply (*subst pedge-num-heap-ptr-coerce*[**where** *l=p* **and** *iL=iP*])
apply *simp+*
apply (*metis uint-nat*)
apply (*drule less-unat-plus1*, *safe*, *blast*)
apply (*subst pedge-num-heap-ptr-coerce*[**where** *l=p* **and** *iL=iP*])
apply *simp+*
apply (*metis sint-ucast not-less uint-nat*)
apply (*drule less-unat-plus1*, *safe*, *blast*)
apply (*subst pedge-num-heap-ptr-coerce*[**where** *l=p* **and** *iL=iP*])
apply *simp+*
apply (*metis not-less uint-nat*)
apply (*drule less-unat-plus1*, *safe*, *blast*)
apply (*subst pedge-num-heap-ptr-coerce*[**where** *l=p* **and** *iL=iP*])
apply *simp+*
apply (*subst head-heap*[**where** *iG=iG*], (*simp add*: *uint-nat*)+)
apply (*drule less-unat-plus1*, *safe*, *blast*)

```
                    apply (subst pedge-num-heap[where l=n and iL=iN], simp+)
                    apply (subst pedge-num-heap[where l=n and iL=iN], simp)
                     apply (drule-tac e=iP vv in  wellformed-iGraph[where G=iG])
                      apply (metis not-le pedge-num-heap-ptr-coerce word-zero-le)
                     apply simp
                    apply (subst tail-heap[where iG=iG], simp+)
                     apply (metis not-le pedge-num-heap-ptr-coerce word-zero-le)
                    apply (subst pedge-num-heap-ptr-coerce[where l=p and iL=iP])
                       apply simp+
                    apply (metis uint-nat)
                   apply (drule less-unat-plus1, safe, blast)
                   apply (subst pedge-num-heap[where l=n and iL=iN])
                     apply (simp add: uint-nat)+
                  apply (metis le-def word-le-nat-alt word-not-le
                  less-unat-plus1 eq-of-nat-conv)
                 apply (metis unat-minus-plus1-less)
                apply (rule arrlist-nth, blast, blast)
                apply (simp add: uint-nat unat-mono)
               apply (rule arrlist-nth, blast, blast)
               apply (simp add: uint-nat)
               apply (drule-tac i=vv in pedge-num-heap-ptr-coerce[where l=p and
iL=iP])
                 apply simp+
               apply (drule-tac e=iP vv in wellformed-iGraph[where G=iG])
                apply simp+
               apply (drule-tac e=iP vv in tail-heap[where iG=iG])
                apply (simp add: uint-nat unat-mono)+
              apply (rule arrlist-nth, (simp add: uint-nat unat-mono)+)+
          apply (metis less-unat-plus1 word-unat.Rep-inverse)
         apply (metis eq-of-nat-conv less-unat-plus1)
        apply (metis (hide-lams, no-types) eq-of-nat-conv less-unat-plus1)
       apply (metis (no-types) less-unat-plus1 word-unat.Rep-inverse)
      apply (metis (no-types) less-unat-plus1 word-unat.Rep-inverse)
     apply (metis inc-le)
    apply (metis unat-minus-plus1-less)
   apply metis
  apply wp
  apply fast
  done
lemma num-less-n:
  fixes v
  assumes is-root G r p n
  assumes parent-num-assms-inv G r p n (unat (ivertex-cnt G))
  assumes v < ivertex-cnt G
  shows n v < ivertex-cnt G
proof -
  have ivertex-cnt G > 0
    using assms by (metis word-neq-0-conv word-not-simps(1))
  thus ?thesis
```

using *assms* **unfolding** *parent-num-assms-inv-def is-root-def*
by (*cases v=r*, *presburger* , *metis unat-mono word-unat.Rep-inverse*)
qed

lemma *parent-num-assms-inv-num-ne-0*:
fixes *v*
assumes *wf-digraph* (*abs-IGraph G*)
assumes *is-root G r p n*
assumes *parent-num-assms-inv G r p n* (*unat* (*ivertex-cnt G*))
assumes $v \neq r$
assumes $v <$ (*ivertex-cnt G*)
shows $n\ v \neq 0$
proof−
have $p\ v \in$ *arcs* (*abs-IGraph G*)
using *assms*(*3*−*5*) *unat-mono*
unfolding *parent-num-assms-inv-def*
by *fastforce*
hence *fst* (*iedges G* (*p v*)) $\in$ *verts* (*abs-IGraph G*)
using *assms*(*1*) *wf-digraph-def* **by** *fastforce*
hence *n* (*fst* (*snd* (*snd G*) (*p v*))) $<$ *ivertex-cnt G*
using *num-less-n*[*OF assms*(*2*,*3*)] **by** *fastforce*
moreover
have *n v* = *n* (*fst* (*snd* (*snd G*) (*p v*))) + *1*
using *assms unat-mono*
unfolding *parent-num-assms-inv-def*
by *force*
ultimately
show *?thesis* **using** *assms*
by (*metis less-is-non-zero-p1*)
qed

lemma *connected-components-locale-num-eq-invariants′*:
$\bigwedge$*G r p n.*
(*connected-components-locale* (*abs-IGraph G*) (*unat* $\circ$ *n*) (*abs-pedge p*) *r*
$\wedge$ ($\forall$ *v* $\in$ *verts* (*abs-IGraph G*). *v*$\neq$ *r* $\longrightarrow$ (*unat* $\circ$ *n*) *v* $<$ *unat* (*ivertex-cnt G*)))
=
(*wf-digraph* (*abs-IGraph G*) $\wedge$
is-root G r p n $\wedge$
parent-num-assms-inv G r p n (*unat* (*ivertex-cnt G*)))
proof −
fix *G* **fix** *r*::*32 word* **fix** *p n*::*32 word* $\Rightarrow$ *32 word*
let *?aG* = *abs-IGraph G*
let *?ap* = *abs-pedge p*
let *?an* = *unat* $\circ$ *n*
let *?wf* = *wf-digraph ?aG*
let *?is-root* = *r* $\in$ *verts ?aG* $\wedge$ *?ap r* = *None* $\wedge$ *?an r* = *0*
let *?pnai* = ($\forall$ *v*. *v* $\in$ *verts ?aG* $\wedge$ *v* $\neq$ *r* $\longrightarrow$
($\exists$ *e*$\in$*arcs ?aG*. *?ap v* = *Some e* $\wedge$
head ?aG e = *v* $\wedge$

```
          ?an v = ?an (tail ?aG e) + 1)) ∧
        (∀ v. v ∈verts ?aG ∧  v ≠ r ⟶
          ?an v < unat (ivertex-cnt G))
  have isr-eq: ?is-root = is-root G r p n
    unfolding is-root-def
    using None-abs-pedgeI unat-eq-0 by auto
moreover
  have (?wf ∧ ?is-root ∧ ?pnai)
     = (?wf ∧ is-root G r p n ∧
       parent-num-assms-inv G r p n (unat (ivertex-cnt G)))
  proof –
  {
    assume wf: ?wf
    assume isr: ?is-root
    assume ∗: ⋀ v. v ∈ verts ?aG ∧ v ≠ r ⟹
    (∃ e ∈ arcs ?aG. ?ap v = Some e ∧
    head ?aG e = v ∧
    ?an v =  ?an (tail ?aG e) + 1) ∧ (?an v < unat (ivertex-cnt G))
    {
      fix i
      let ?i = of-nat i
      assume i < unat (ivertex-cnt G) ∧ ?i ≠ r
      then have ii: ?i ∈ verts (abs-IGraph G) ∧ ?i ≠ r
        by (simp add: word-of-nat-less)
      then obtain e where e-assms:
        e ∈ arcs ?aG
        ?ap ?i = Some e
        head ?aG e = ?i
        ?an ?i =  ?an (tail ?aG e) + 1
        ?an ?i < unat (ivertex-cnt G) using ∗[OF ii] by auto
      have pi-e: p ?i = e
        using e-assms(2) abs-pedge-def Some-abs-pedgeI
        by (cases ?ap ?i) force+
      with e-assms pi-e  Some-abs-pedgeI have
        p ?i < iedge-cnt G ∧
         0 ≤ sint (p ?i) ∧
         snd (iedges G (p ?i)) = ?i ∧
         n ?i = n (fst (iedges G (p ?i))) + 1 ∧
         n ?i < ivertex-cnt G ∧
         n ?i ≠ 0
         by (auto,
             metis Some-abs-pedgeI,
             metis (hide-lams, mono-tags) Suc-eq-plus1 unat-1
                   word-arith-nat-add word-unat.Rep-inverse,
             metis word-less-nat-alt)
    } then have is-root G r p n ∧
              parent-num-assms-inv G r p n (unat (ivertex-cnt G))
    unfolding parent-num-assms-inv-def using isr isr-eq by blast
  }
```

```
  hence ?wf ∧ ?is-root ∧ ?pnai
      ⟹ is-root G r p n ∧
        parent-num-assms-inv G r p n (unat (ivertex-cnt G)) by presburger
  moreover
  {
    assume wf: ?wf
    assume isr: is-root G r p n
    assume pna: parent-num-assms-inv G r p n (unat (ivertex-cnt G))
    {
      fix v
      assume vG: v ∈ verts ?aG
      assume vnr: v ≠ r
      have uvG: unat v < unat (ivertex-cnt G)
        using vG by (simp add: word-less-nat-alt)
     have nv-ne0: n v ≠ 0 using pna isr wf unfolding parent-num-assms-inv-def

        by (metis parent-num-assms-inv-num-ne-0 pna uvG vnr word-less-nat-alt)
      then have ∗:
        p v < iedge-cnt G ∧
        0 ≤ sint (p v) ∧
        snd (iedges G (p v)) =  v ∧
        n v = n (fst (iedges G (p v))) + 1 ∧
        n v < ivertex-cnt G
        using vnr pna
        unfolding parent-num-assms-inv-def
        by (metis uvG word-unat.Rep-inverse)
      then have 1:
      ∃ e. e ∈ arcs ?aG ∧ ?ap v = Some e ∧
          head ?aG e = v ∧
          ?an v = ?an (tail ?aG e) + 1
        using abs-pedge-def linorder-not-less unatSuc2 nv-ne0 by auto
     have 2: ?an v < unat (ivertex-cnt G)
     using ∗ by (metis o-apply word-less-nat-alt)
     from 1 2 have
     (∃ e. e ∈ arcs ?aG ∧ ?ap v = Some e ∧
          head ?aG e = v ∧
          ?an v = ?an (tail ?aG e) + 1) ∧
      ?an v < unat (ivertex-cnt G) by blast
   } then have ?is-root ∧ ?pnai using  isr isr-eq by fast
  }
  hence ?wf ∧ is-root G r p n ∧
        parent-num-assms-inv G r p n (unat (ivertex-cnt G)) ⟹
        ?is-root ∧ ?pnai by presburger
  ultimately
    show ?thesis by blast
  qed
ultimately
show  ?thesis G r p n
  unfolding connected-components-locale-def
```

connected-components-locale-axioms-def
fin-digraph-def fin-digraph-axioms-def
by *auto*
qed

lemma *cc-num-less-n*:
assumes *connected-components-locale* (*abs-IGraph G*) (*unat* ∘ *n*) (*abs-pedge p*) *r*
assumes $v \in$ *verts* (*abs-IGraph G*)
shows (*unat* ∘ *n*) $v <$ *unat* (*ivertex-cnt G*)
using *connected-components-locale.path-from-root-num*[*OF assms*] *path-length*
by *presburger*

lemma *connected-components-locale-eq-invariants'*:
⋀*G r p n.*
(*connected-components-locale* (*abs-IGraph G*) (*unat* ∘ *n*) (*abs-pedge p*) *r*) =
(*wf-digraph* (*abs-IGraph G*) ∧
is-root G r p n ∧
parent-num-assms-inv G r p n (*unat* (*ivertex-cnt G*)))
using *connected-components-locale-num-eq-invariants' cc-num-less-n* **by** *blast*

lemma *check-connected-spc*:
⦃ *P and*
(λ*s. wf-digraph* (*abs-IGraph iG*) ∧
is-graph s iG g ∧
is-numm s iG iN n ∧
is-pedge s iG iP p)⦄
check-connected' g r p n
⦃ (λ- *s. P s*) *And*
(λ*rr s. rr* ≠ *0* ⟷
connected-components-locale (*abs-IGraph iG*) (*unat* ∘ *iN*) (*abs-pedge iP*) *r*)
⦄!
apply (*clarsimp simp*: *check-connected'-def*
connected-components-locale-eq-invariants')
apply *wp*
apply (*rule-tac P1*= *P and*
(λ*s. wf-digraph* (*abs-IGraph iG*) ∧
is-graph s iG g ∧
is-numm s iG iN n ∧
is-pedge s iG iP p ∧
r < *ivertex-cnt iG* ∧
is-root iG r iP iN)
in *validNF-post-imp*[*OF - check-parent-num-spc'*])
unfolding *fin-digraph-def fin-digraph-axioms-def*
apply *force*
apply *wp*
apply (*auto simp*: *check-r'-spc is-root-def*)[]
done

end
end