A Framework for Verified Depth-First Algorithms

René Neumann (rene.neumann@in.tum.de)

June 12, 2012

Contents

1 A nondeterministic DFS implementation

theory *DFS*
imports
Main
Graph-Ext
Refine-Additions
../General/Misc-Additions
begin

Based on the work on the nondeterministic while combinator we develop a depth-first-search function that works on sets rather than lists. This collapses different runs of the search into a set of results, which share a common property.

1.1 Data structures and algorithm specification

The representation of a working state.

record $('n)$ *dfs-ws* =

start :: *'n* — the starting node
stack :: *'n list* — the search stack of the current run
wl :: *'n set list* — the set of successors that still need to visited for each node
discover :: (*'n*, *nat*) *map* — mapping a node to the timestamp of their discovery
finish :: (*'n*, *nat*) *map* — mapping a node to the timestamp of the finishing, i.e. the moment the search backtracks
counter :: *nat* — the counter for discover and finish

abbreviation *discovered s* ≡ *dom* (*discover s*)
abbreviation *finished s* ≡ *dom* (*finish s*)
abbreviation *disc* (*δ*) **where** *disc s x* ≡ *the* (*discover s x*)
abbreviation *fin* (*φ*) **where** *fin s x* ≡ *the* (*finish s x*)

As the DFS algorithm is intended to be only a framework for DFS-based algorithm, we need to carry around a state, that is used by the concrete implementation.

record (*'S*, *'n*) *dfs-sws* = (*'n*) *dfs-ws* +
state :: *'S* — the state of the implementation

The representation of the parametrized algorithm.

record (*'S*, *'n*) *dfs-algorithm* =
dfs-cond :: *'S* ⇒ *bool* — the condition to be satisified by the state S to continue searching from here.
dfs-action :: *'S* ⇒ (*'S*,*'n*) *dfs-sws* ⇒ *'n* ⇒ *'S* — modifies the state for the current node BEFORE visiting the successors.
dfs-post :: *'S* ⇒ (*'S*, *'n*) *dfs-sws* ⇒ *'n* ⇒ *'S* — modifies the state for the current node AFTER having visited the successors (i.e. during backtracking).
dfs-remove :: *'S* ⇒ (*'S*, *'n*) *dfs-sws* ⇒ *'n* ⇒ *'S* — modifies the state if a node has already been visited and is removed from the stack
dfs-start :: *'n* ⇒ *'S* — the starting state
dfs-restrict :: *'n set* — a set of states that should not be visited

context *finite-digraph*
begin

The *dfs-step* returns a set of working states, that may be reached from the current node. In general, it returns one state for each successor of the current node.

fun *dfs-step'* :: (*'S*, *'n*, *'X*) *dfs-algorithm-scheme* ⇒ (*'S*,*'n*) *dfs-sws* ⇒ *'S* ⇒ *'n* ⇒ (*'n*, *nat*) *map* ⇒ (*'n*, *nat*) *map* ⇒ *nat* ⇒ *'n set list* ⇒ *'n list* ⇒ ((*'S*, *'n*) *dfs-sws*) *set*
where *dfs-step' dfs S s n d f c w* [] = {}
|*dfs-step' dfs S s n d f c w* (*x*#*xs*) = (*if hd w* = {} *then* {*dfs-sws.make n xs* (*tl w*) *d* (*f*(*x* ↦ *c*)) (*Suc c*) (*dfs-post dfs s S x*)}
else {*dfs-sws.make n st' w' d' f c' s'* | *e st' w' d' c' s'*. *e* ∈ *hd w* ∧
(*if e* ∈ *dfs-restrict dfs then st'* = *x*#*xs* ∧ *w'* = (*hd w* − {*e*})#*tl w* ∧ *d'* = *d* ∧ *c'* = *c* ∧ *s'* = *s*

else if $e \in$ *dom* d *then* $st' = x\#xs \wedge w'$ $= (hd\ w - \{e\})\#tl\ w \wedge d' = d \wedge c' = c \wedge s' =$ *dfs-remove dfs* $s\ S\ e$
else $st' = e\#x\#xs \wedge w' =$ *succs* $e\#(hd$ $w - \{e\})\#tl\ w \wedge d' = d(e \mapsto c) \wedge c' =$ *Suc* $c \wedge s' =$ *dfs-action dfs* $s\ S\ e)\})$

definition *dfs-step* :: ('*S*, '*n*, '*R*) *dfs-algorithm-scheme* ⇒ ('*S*, '*n*) *dfs-sws* ⇒ ('*S* ,'*n*) *dfs-sws set*
where *dfs-step dfs s* ≡ *dfs-step' dfs s* (*state s*) (*start s*) (*discover s*) (*finish s*) (*counter s*) (*wl s*) (*stack s*)

lemma *dfs-step-simps*[*simp*]:
stack $s = [] \Longrightarrow$ *dfs-step dfs* $s = \{\}$
stack $s = x\#xs \Longrightarrow$ *wl* $s \neq [] \Longrightarrow$ *hd* (*wl s*) = {}
$\Longrightarrow$ *dfs-step dfs* $s = \{s(\!|$*stack* := *xs*, *wl* := *tl* (*wl s*), *finish* := (*finish s*)($x \mapsto$ *counter s*), *counter* := *Suc* (*counter s*), *state* := *dfs-post dfs* (*state s*) *s x* $|\!)\}$
stack $s = x\#xs \Longrightarrow$ *wl* $s \neq [] \Longrightarrow$ *hd* (*wl s*) ≠ {}
$\Longrightarrow$ *dfs-step dfs* $s = \{$*dfs-sws.make* (*start s*) *st' w' d'* (*finish s*) *c' s'* | *e st' w' d' c' s'*. *e* ∈ *hd* (*wl s*) ∧
(*if e* ∈ *dfs-restrict dfs then st'* = *x*#*xs* ∧ *w'* = (*hd* (*wl s*) − {*e*})#*tl* (*wl s*) ∧ *d'* = *discover s* ∧ *c'* = *counter s* ∧ *s'* = (*state s*)
else if e ∈ *discovered s then st'* = *x*#*xs* ∧ *w'* = (*hd* (*wl s*) − {*e*})#*tl* (*wl s*) ∧ *d'* = *discover s* ∧ *c'* = *counter s* ∧ *s'* = *dfs-remove dfs* (*state s*) *s e*
else st' = *e*#*x*#*xs* ∧ *w'* = *succs e*#(*hd* (*wl s*) − {*e*})#*tl* (*wl s*) ∧ *d'* = (*discover s*)(*e* ↦ *counter s*) ∧ *c'* = *Suc* (*counter s*) ∧ *s'* = *dfs-action dfs* (*state s*) *s e*)}
unfolding *dfs-step-def*
by (*simp-all add*: *dfs-sws.defs*)

lemma *dfs-step-intros*:
stack $s = x\#xs \Longrightarrow$ *wl* $s \neq [] \Longrightarrow$ *hd* (*wl s*) = {} $\Longrightarrow$ $s' = s(\!|$*stack* := *xs*, *wl* := *tl* (*wl s*), *finish* := (*finish s*)($x \mapsto$ *counter s*), *counter* := *Suc* (*counter s*), *state* := *dfs-post dfs* (*state s*) *s x* $|\!) \Longrightarrow$ $s' \in$ *dfs-step dfs s*
stack $s = x\#xs \Longrightarrow$ *wl* $s \neq [] \Longrightarrow$ *hd* (*wl s*) ≠ {} $\Longrightarrow$ *e* ∈ *hd* (*wl s*) $\Longrightarrow$ *e* ∈ *dfs-restrict dfs* $\Longrightarrow$ $s' = s(\!|$*wl* := (*hd* (*wl s*) − {*e*})#*tl* (*wl s*)$|\!) \Longrightarrow$ $s' \in$ *dfs-step dfs s*
stack $s = x\#xs \Longrightarrow$ *wl* $s \neq [] \Longrightarrow$ *hd* (*wl s*) ≠ {} $\Longrightarrow$ *e* ∈ *hd* (*wl s*) $\Longrightarrow$ *e* ∉ *dfs-restrict dfs* $\Longrightarrow$ *e* ∈ *discovered s* $\Longrightarrow$ $s' = s(\!|$*wl* := (*hd* (*wl s*) − {*e*})#*tl* (*wl s*), *state* := *dfs-remove dfs* (*state s*) *s e*$|\!) \Longrightarrow$ $s' \in$ *dfs-step dfs s*
stack $s = x\#xs \Longrightarrow$ *wl* $s \neq [] \Longrightarrow$ *hd* (*wl s*) ≠ {} $\Longrightarrow$ *e* ∈ *hd* (*wl s*) $\Longrightarrow$ *e* ∉ *dfs-restrict dfs* $\Longrightarrow$ *e* ∉ *discovered s* $\Longrightarrow$ $s' = s(\!|$ *state* := *dfs-action dfs* (*state s*) *s e*, *stack* := *e*#*x*#*xs*, *wl* := *succs e*#(*hd* (*wl s*) − {*e*})#(*tl* (*wl s*)), *discover* := (*discover s*)(*e* ↦ *counter s*), *counter* := *Suc* (*counter s*)$|\!) \Longrightarrow$ $s' \in$ *dfs-step dfs s*
unfolding *dfs-step-def*
apply (*simp-all add*: *dfs-sws.defs*)
apply (*rule exI*[**where** *x*=*e*], *simp*)+
done

lemma *dfs-step-cases*:
assumes *inv*: *stack s* $\neq$ [] $\Longrightarrow$ *wl s* $\neq$ []
and ⟦ *stack s* = [] ⟧ $\Longrightarrow$ *P*
and $\bigwedge$ *x xs w ws* . ⟦ *stack s* = *x*#*xs*; *wl s* = *w*#*ws*; *w* = {} ⟧ $\Longrightarrow$ *P*
and $\bigwedge$ *x xs w ws*. ⟦ *stack s* = *x*#*xs*; *wl s* = *w*#*ws*; *w* $\neq$ {} ⟧ $\Longrightarrow$ *P*
shows *P*
using *assms*
proof (*cases stack s*)
case (*Cons x xs*) **with** *inv* **obtain** *w ws* **where** *w*#*ws* = *wl s* **by** (*cases wl s*) *auto*
with *Cons assms*(2−) **show** *?thesis* **by** (*cases w* = {}) *metis*+
qed

lemma *dfs-step-cases-elem* [*case-names empty restrict remove visit*, *cases pred*: *dfs-step*, *consumes 2*]:
assumes *inv*: *s'* ∈ *dfs-step dfs s wl s* $\neq$ []
and *empty*: $\bigwedge$*x xs w ws*. ⟦ *stack s* = *x*#*xs*; *wl s* = *w*#*ws*; *w* = {} ⟧ $\Longrightarrow$
s' = *s*(|*stack* := *xs*, *wl* := *ws*, *finish* := (*finish s*)(*x* ↦ *counter s*), *counter* := *Suc* (*counter s*), *state* := *dfs-post dfs* (*state s*) *s x* |) $\Longrightarrow$ *P*
and *restrict*: $\bigwedge$ *e x xs w ws*. ⟦ *stack s* = *x*#*xs*; *wl s* = *w*#*ws*; *w* $\neq$ {}; *e* ∈ *w*; *e* ∈ *dfs-restrict dfs* ⟧ $\Longrightarrow$
s' = *s*(| *wl* := (*w* − {*e*})#*ws* |) $\Longrightarrow$ *P*
and *remove*: $\bigwedge$ *e x xs w ws*. ⟦ *stack s* = *x*#*xs*; *wl s* = *w*#*ws*; *w* $\neq$ {}; *e* ∈ *w*; *e* ∉ *dfs-restrict dfs*; *e* ∈ *discovered s* ⟧ $\Longrightarrow$
s' = *s*(| *wl* := (*w* − {*e*})#*ws*, *state* := *dfs-remove dfs* (*state s*) *s e*|) $\Longrightarrow$ *P*
and *visit*: $\bigwedge$ *e x xs w ws*. ⟦ *stack s* = *x*#*xs*; *wl s* = *w*#*ws*; *w* $\neq$ {}; *e* ∈ *w*; *e* ∉ *dfs-restrict dfs*; *e* ∉ *discovered s* ⟧ $\Longrightarrow$
s' = *s*(| *state* := *dfs-action dfs* (*state s*) *s e*, *stack* := *e*#*x*#*xs*, *wl* := *succs e*#(*w* − {*e*})#*ws*, *discover* := (*discover s*)(*e* ↦ *counter s*), *counter* := *Suc* (*counter s*)|) $\Longrightarrow$ *P*
shows *P*
using *inv*
proof (*cases s rule*: *dfs-step-cases*)
case *3* **then show** *?thesis* **using** *empty*[*OF 3*] *inv*
using *dfs-step-simps*(*2*)
by *fastforce*
next
case (*4 x xs w ws*) **with** *inv dfs-step-simps*(*3*)[*of s x xs dfs*]
obtain *e* **where** *e*:
e ∈ *w*
if e ∈ *dfs-restrict dfs then stack s'* = *x*#*xs* ∧ *wl s'* = (*hd* (*wl s*) − {*e*})#*tl* (*wl s*) ∧ *discover s'* = *discover s* ∧ *counter s'* = *counter s* ∧ *state s'* = *state s* ∧ *start s'* = *start s* ∧ *finish s'* = *finish s*
else if e ∈ *discovered s*
then stack s' = *x*#*xs* ∧ *wl s'* = (*hd* (*wl s*) − {*e*})#*tl* (*wl s*) ∧ *discover s'* = *discover s* ∧ *counter s'* = *counter s* ∧ *state s'* = *dfs-remove dfs* (*state s*) *s e* ∧ *start s'* = *start s* ∧ *finish s'* = *finish s*
else stack s' = *e*#*x*#*xs* ∧ *wl s'* = *succs e*#(*hd* (*wl s*) − {*e*})#*tl* (*wl s*) ∧

discover s' = (*discover s*)(*e* ↦ *counter s*) ∧ *counter s'* = *Suc* (*counter s*) ∧ *state s'* = *dfs-action dfs* (*state s*) *s e* ∧ *start s'* = *start s* ∧ *finish s'* = *finish s*
by (*simp add*: *dfs-sws.defs*) *force*

show *?thesis*
proof (*cases e* ∈ *dfs-restrict dfs*)
case *True* **with** *4 e restrict*[*OF 4 e*(*1*)] **show** *?thesis* **by** *simp*
next
case *False* **with** *4 e remove*[*OF 4 e*(*1*)] *visit*[*OF 4 e*(*1*)] **show** *?thesis* **by** (*cases e* ∈ *discovered s*) *simp-all*
qed
qed *simp-all*

lemma *stack-wl-remove-induct* [*case-names 0 greater*, *consumes 5*]:
assumes *stack s* = *x*#*xs* **and** *wl s* = *w*#*ws*
and *stack s'* = *x*#*xs* **and** *wl s'* = (*w* − {*e*})#*ws*
and *inv*: ∀ *n* < *length* (*stack s*). *P* (*stack s* ! *n*) (*wl s* ! *n*) (*discover s*) (*finish s*) (*start s*)
and *case-0*: *P x w* (*discover s*) (*finish s*) (*start s*) ⟹ *P x* (*w* − {*e*}) (*discover s'*) (*finish s'*) (*start s'*)
and *step*: ⋀ *n*. ⟦ *n* < *length* (*stack s*); *n* > *0*; *P* (*stack s* ! *n*) (*wl s* ! *n*) (*discover s*) (*finish s*) (*start s*) ⟧ ⟹ *P* (*stack s* ! *n*) (*wl s* ! *n*) (*discover s'*) (*finish s'*) (*start s'*)
shows ∀ *n* < *length* (*stack s'*). *P* (*stack s'* ! *n*) (*wl s'* ! *n*) (*discover s'*) (*finish s'*) (*start s'*)
proof (*rule allI*, *rule impI*)
fix *n*
assume *lt'*: *n* < *length* (*stack s'*) **with** *assms* **have** *lt*: *n* < *length* (*stack s*) **by** *simp*
show *P* (*stack s'* ! *n*) (*wl s'* ! *n*)(*discover s'*) (*finish s'*) (*start s'*)
proof (*cases n*)
case *0* **with** *assms lt'* **show** *?thesis* **by** *auto*
next
case (*Suc m*) **with** *lt step*[*of n*] *inv* **have** *P* (*stack s* ! *n*) (*wl s* ! *n*) (*discover s'*) (*finish s'*) (*start s'*) **by** *simp*
with *Suc assms*(*1*−*4*) **show** *?thesis* **by** *auto*
qed
qed

lemma *stack-wl-visit-induct* [*case-names case-0 case-1 greater*, *consumes 5*]:
assumes *stack s* = *x*#*xs* **and** *wl s* = *w*#*ws*
and *stack s'* = *e*#*x*#*xs* **and** *wl s'* = *succs e*#(*w* − {*e*})#*ws*
and *inv*: ∀ *n* < *length* (*stack s*). *P* (*stack s* ! *n*) (*wl s* ! *n*) (*discover s*) (*finish s*) (*start s*)
and *case-0*: *P e* (*succs e*) (*discover s'*) (*finish s'*) (*start s'*)
and *case-1*: *P x w* (*discover s*) (*finish s*) (*start s*) ⟹ *P x* (*w* − {*e*}) (*discover s'*) (*finish s'*) (*start s'*)
and *step*: ⋀ *n*. ⟦ *n* < *length* (*stack s*); *n* > *0*; *P* (*stack s* ! *n*) (*wl s* ! *n*) (*discover*

s) (*finish s*) (*start s*) ⟧ ⟹ *P* (*stack s* ! *n*) (*wl s* ! *n*) (*discover s'*) (*finish s'*) (*start s'*)
shows ∀ *n* < *length* (*stack s'*). *P* (*stack s'* ! *n*) (*wl s'* ! *n*) (*discover s'*) (*finish s'*) (*start s'*)
proof (*rule allI*, *rule impI*)
fix *n*
assume *lt'*: *n* < *length* (*stack s'*) **hence** *lt*: *n* < *Suc* (*length* (*stack s*)) **using** *assms*(*1*,*3*) **by** *simp*

show *P* (*stack s'* ! *n*) (*wl s'* ! *n*) (*discover s'*) (*finish s'*) (*start s'*)
proof (*cases n*)
case *0* **with** *assms lt'* **show** *?thesis* **by** *simp*
next
case (*Suc m*) **note** *suc-m* = *this*
show *?thesis*
proof (*cases m*)
case *0* **with** *suc-m assms* **show** *?thesis* **by** *auto*
next
case (*Suc m'*) **with** *suc-m lt* **have** *m* < *length* (*stack s*) **and** *m* > *0* **by** *simp-all*
with *step*[*of m*] *inv* **have** *P* (*stack s* ! *m*) (*wl s* ! *m*) (*discover s'*) (*finish s'*) (*start s'*) **by** *simp*
with *suc-m assms*(*1*−*4*) ‹*m* > *0*› **show** *?thesis* **by** *auto*
qed
qed
qed

Basic lemmata over *dfs-step* wrt to properties of the DFS

lemma *dfs-step-preserves-start*:
s' ∈ *dfs-step dfs s* ⟹ *wl s* ≠ [] ⟹ *start s'* = *start s*
by (*cases rule*: *dfs-step-cases-elem*) (*auto simp add*: *dfs-sws.defs*)

lemma *dfs-step-discovered-subset*:
s' ∈ *dfs-step dfs s* ⟹ *wl s* ≠ [] ⟹ *discovered s'* ⊇ *discovered s*
by (*cases rule*: *dfs-step-cases-elem*) *force*+

lemma *dfs-step-exists*:
assumes *inv*: *stack s* ≠ [] *wl s* ≠ []
shows *dfs-step dfs s* ≠ {}
using *inv*
proof (*cases stack s*)
case (*Cons x xs*) **then show** *?thesis*
proof (*cases hd*(*wl s*) = {})
case *True* **with** *dfs-step-simps*(*2*)[*OF Cons inv*(*2*)] **show** *?thesis* **by** *blast*
next
case *False* **then obtain** *e* **where** *e* ∈ *hd* (*wl s*) **by** *blast*
thus *?thesis*
using *dfs-step-simps*(*3*)[*OF Cons inv*(*2*) ‹*hd*(*wl s*) ≠ {}›, *of dfs*]
apply *simp*

apply (*rule ex-comm*[*THEN iffD1*]) — commute the two ex.-quant. variables in the conclusion
apply (*rule-tac x*=*e* **in** *exI*) — instantiate the first one
apply (*cases e* ∈ *dfs-restrict dfs*, *simp*)
by (*cases e* ∈ *discovered s*) *simp-all*
qed
qed *simp*

lemma *dfs-step-stack-not-discovered*:
assumes $s' \in$ *dfs-step dfs s wl s* $\neq []$
and *length* (*stack s'*) > *length* (*stack s*)
shows *hd* (*stack s'*) $\notin$ *discovered s*
using *assms*
by *cases auto*

lemma *dfs-step-stack-not-restricted*:
assumes $s' \in$ *dfs-step dfs s wl s* $\neq []$
and *length* (*stack s'*) > *length* (*stack s*)
shows *hd* (*stack s'*) $\notin$ *dfs-restrict dfs*
using *assms*
by *cases auto*

dfs-cond-compl specifies the condition that must hold to continue searching. It consists of the conditions needed by the framework and the ones given by *dfs-cond dfs*, that are dependent on the concrete implementation.

definition *dfs-cond-compl* :: ('*S*, '*n*, '*X*) *dfs-algorithm-scheme* ⇒ ('*S*, '*n*) *dfs-sws* ⇒ *bool*
where
dfs-cond-compl dfs s ≡ *stack s* $\neq []$ ∧ *wl s* $\neq []$ ∧ *dfs-cond dfs* (*state s*)

lemma *dfs-cond-compl-cond* [*dest*]:
dfs-cond-compl dfs s ⟹ *dfs-cond dfs* (*state s*)
unfolding *dfs-cond-compl-def*
by *auto*

lemma *dfs-cond-compl-step-exists* [*dest*]:
dfs-cond-compl dfs s ⟹ *dfs-step dfs s* $\neq \{\}$
unfolding *dfs-cond-compl-def*
by (*simp add*: *dfs-step-exists*)

dfs-next includes the above condition, s.t. it is only possible to make a step, if the condition is met.

definition *dfs-next* :: ('*S*, '*n*, '*X*) *dfs-algorithm-scheme* ⇒ ('*S*, '*n*) *dfs-sws* ⇒ ('*S*, '*n*) *dfs-sws* ⇒ *bool*
where
dfs-next dfs s s' ⟷
dfs-cond-compl dfs s ∧ $s' \in$ *dfs-step dfs s*

lemma *dfs-nextE*:

⟦ *dfs-next dfs s s'*; ⟦*dfs-cond-compl dfs s*; *s'* ∈ *dfs-step dfs s*⟧ ⟹ *P* ⟧ ⟹ *P*
unfolding *dfs-next-def*
by *blast*

lemma *dfs-next-dfs-step*[*dest*]:
dfs-next dfs s s' ⟹ *s'* ∈ *dfs-step dfs s*
unfolding *dfs-next-def*
by *simp*

lemma *dfs-next-stack-notempty*[*simp*, *intro*]:
dfs-next dfs s s' ⟹ *stack s* ≠ []
unfolding *dfs-next-def dfs-cond-compl-def*
by *simp-all*

lemma *dfs-next-wl-notempty*[*simp*, *intro*]:
dfs-next dfs s s' ⟹ *wl s* ≠ []
unfolding *dfs-next-def dfs-cond-compl-def*
by *simp-all*

lemma *lift-to-dfs-next*:
assumes *s'* ∈ *dfs-step dfs s* ⟹ *wl s* ≠ [] ⟹ *PROP S*
shows *dfs-next dfs s s'* ⟹ *PROP S*
using *assms*
by (*simp add*: *dfs-next-dfs-step*)

lemmas *dfs-next-cases-elem* [*case-names empty restrict remove visit*, *cases pred*: *dfs-next*] = *dfs-step-cases-elem*[*COMP lift-to-dfs-next*]

lemma *dfs-nextI*:
dfs-cond-compl dfs s ⟹ *s'* ∈ *dfs-step dfs s* ⟹ *dfs-next dfs s s'*
unfolding *dfs-next-def*
by *auto*

lemma *dfs-next-cond-compl*:
dfs-next dfs s s' ⟹ *dfs-cond-compl dfs s*
unfolding *dfs-next-def*
by *auto*

lemmas *dfs-next-intros* = *dfs-step-intros*[*THEN dfs-nextI*[*rotated*], *rotated* −*1*]

end

1.2 Properties of the DFS

Using some notion of a *good state* via the predicate *dfs-constructable*, we enable the tool of induction over DFS. This is very useful for showing that certain (basic) properties hold throughout the search.

context *finite-digraph*

begin

To be able to define good states, we have to define a basic state, i.e. the state to start with for every possible DFS. Then *dfs-constructable* is just the set of states, that can be constructed from this point with the given graph.

definition *dfs-constr-start* :: ('*S*, '*n*, '*X*) *dfs-algorithm-scheme* ⇒ '*n* ⇒ ('*S*, '*n*) *dfs-sws*
where *dfs-constr-start dfs x* ≡ (| *start* = *x*, *stack* = [*x*], *wl* = [*succs x*], *discover* = [*x* ↦ *0*], *finish* = *Map.empty*, *counter* = *1*, *state* = *dfs-start dfs x* |)

lemma *dfs-constr-start-simps*[*simp*]:
start (*dfs-constr-start dfs x*) = *x*
stack (*dfs-constr-start dfs x*) = [*x*]
wl (*dfs-constr-start dfs x*) = [*succs x*]
discovered (*dfs-constr-start dfs x*) = {*x*}
ran (*discover* (*dfs-constr-start dfs x*)) = {*0*}
finished (*dfs-constr-start dfs x*) = {}
ran (*finish* (*dfs-constr-start dfs x*)) = {}
counter (*dfs-constr-start dfs x*) = *1*
state (*dfs-constr-start dfs x*) = *dfs-start dfs x*
by (*simp-all add*: *dfs-constr-start-def*)

inductive *dfs-constructable* :: ('*S*, '*n*, '*X*) *dfs-algorithm-scheme* ⇒ ('*S*, '*n*) *dfs-sws* ⇒ *bool*
for *dfs* :: ('*S*, '*n*, '*X*) *dfs-algorithm-scheme* **where**
start: ⋀*x*. *x* ∈ *V* ⟹ *x* ∉ *dfs-restrict dfs* ⟹ *dfs-constructable dfs* (*dfs-constr-start dfs x*)
| *step*: *dfs-next dfs s s'* ⟹ *dfs-constructable dfs s* ⟹ *dfs-constructable dfs s'*

lemma *dfs-constructable-induct* [*case-names start empty restrict remove visit*, *induct pred*: *dfs-constructable*]:
assumes *inv*: *dfs-constructable dfs s*
and *start'*: ⋀*x*. *x* ∈ *V* ⟹ *x* ∉ *dfs-restrict dfs* ⟹ *P* (*dfs-constr-start dfs x*)
and *empty'*: ⋀*s s' x xs w ws*. ⟦ *dfs-next dfs s s'*; *dfs-constructable dfs s*; *dfs-constructable dfs s'*; *P s*; *stack s* = *x*#*xs*; *wl s* = *w*#*ws*; *w* = {};
s' = *s*(|*stack* := *xs*, *wl* := *ws*, *finish* := (*finish s*)(*x* ↦ *counter s*), *counter* := *Suc* (*counter s*), *state* := *dfs-post dfs* (*state s*) *s x*|)⟧
⟹ *P s'*
and *restrict'*: ⋀*s s' e x xs w ws*. ⟦ *dfs-next dfs s s'*; *dfs-constructable dfs s*; *dfs-constructable dfs s'*; *P s*; *stack s* = *x*#*xs*; *wl s* = *w*#*ws*; *w* ≠ {};
e ∈ *w*; *e* ∈ *dfs-restrict dfs*; *s'* = *s*(| *wl* := (*w* − {*e*})#*ws*|) ⟧
⟹ *P s'*
and *remove'*: ⋀*s s' e x xs w ws*. ⟦ *dfs-next dfs s s'*; *dfs-constructable dfs s*; *dfs-constructable dfs s'*; *P s*; *stack s* = *x*#*xs*; *wl s* = *w*#*ws*; *w* ≠ {};
e ∈ *w*; *e* ∉ *dfs-restrict dfs*; *e* ∈ *discovered s*; *s'* = *s*(| *wl* := (*w* − {*e*})#*ws*, *state* := *dfs-remove dfs* (*state s*) *s e*|) ⟧
⟹ *P s'*
and *visit'*: ⋀*s s' e x xs w ws*. ⟦ *dfs-next dfs s s'*; *dfs-constructable dfs s*; *dfs-constructable*

dfs s'; *P s*; *stack s* = *x*#*xs*; *wl s* = *w*#*ws*; *w* ≠ {};
e ∈ *w*; *e* ∉ *dfs-restrict dfs*; *e* ∉ *discovered s*;
s' = *s*(| *state* := *dfs-action dfs* (*state s*) *s e*, *stack* := *e*#*x*#*xs*, *wl* := *succs e*#(*w* − {*e*})#*ws*, *discover* := (*discover s*)(*e* ↦ *counter s*), *counter* := *Suc* (*counter s*)|) ⟧
⟹ *P s'*
shows *P s*
using *inv*
proof (*induct rule*: *dfs-constructable.induct*)
case *start* **with** *start'* **show** *?case* **by** *metis*
next
case (*step s s'*) **then show** *?case* **by** (*cases rule*: *dfs-next-cases-elem*) (*metis dfs-constructable.step step empty' restrict' remove' visit'*)+
qed

Formulate the constructable predicate with an explicit starting point.

inductive-set *dfs-constr-from* :: ('*S*, '*n*, '*X*) *dfs-algorithm-scheme* ⇒ '*n* ⇒ ('*S*,'*n*) *dfs-sws set*
for *dfs* :: ('*S*, '*n*, '*X*) *dfs-algorithm-scheme*
and *x* :: '*n*
where
start[*simp*]: *x* ∈ *V* ⟹ *x* ∉ *dfs-restrict dfs* ⟹ *dfs-constr-start dfs x* ∈ *dfs-constr-from dfs x*
| *step*: *s* ∈ *dfs-constr-from dfs x* ⟹ *dfs-next dfs s s'* ⟹ *s'* ∈ *dfs-constr-from dfs x*

lemma *dfs-constr-from-rtranclp-dfs-next*:
assumes *x* ∈ *V x* ∉ *dfs-restrict dfs*
shows *s* ∈ *dfs-constr-from dfs x* ⟷ (*dfs-next dfs*)** (*dfs-constr-start dfs x*) *s*
proof
assume *s* ∈ *dfs-constr-from dfs x* **thus** (*dfs-next dfs*)** (*dfs-constr-start dfs x*) *s*
by *induction* (*simp-all add*: *assms*)
next
assume (*dfs-next dfs*)** (*dfs-constr-start dfs x*) *s* **thus** *s* ∈ *dfs-constr-from dfs x*
by *induction* (*simp-all add*: *dfs-constr-from.step assms*)
qed

lemma *dfs-constr-from-constructable*[*dest*]:
s ∈ *dfs-constr-from dfs x* ⟹ *dfs-constructable dfs s*
by (*induction rule*: *dfs-constr-from.induct*) (*simp-all add*: *dfs-constructable.start dfs-constructable.step*)

lemma *dfs-constructable-constr-from-start*:
dfs-constructable dfs s ⟹ *s* ∈ *dfs-constr-from dfs* (*start s*)
by (*induction rule*: *dfs-constructable.induct*) (*simp*, *metis dfs-constr-from.step dfs-step-preserves-start*[*COMP lift-to-dfs-next*])

lemma *constr-from-implies-start*:
s ∈ *dfs-constr-from dfs x* ⟹ *start s* = *x*

by (*induction rule*: *dfs-constr-from.induct*) (*simp-all add*: *dfs-step-preserves-start*[*COMP lift-to-dfs-next*])

lemma *counter-larger-one*:
dfs-constructable dfs s $\Longrightarrow$ *counter s* $\geq$ *1*
by (*induct rule*: *dfs-constructable-induct*) *auto*

lemma *length-wl-eq-stack*:
dfs-constructable dfs s $\Longrightarrow$ *length* (*wl s*) = *length* (*stack s*)
by (*induction rule*: *dfs-constructable-induct*) *auto*

lemma *succs-discovered-subset-wl-all*:
dfs-constructable dfs s $\Longrightarrow$ $\forall$ *n* < *length* (*stack s*). *succs* (*stack s* ! *n*) − *discovered s* − *dfs-restrict dfs* $\subseteq$ *wl s* ! *n*
proof (*induction rule*: *dfs-constructable-induct*)
case (*empty s s'*) **hence** *length* (*stack s*) = *length* (*wl s*) **by** (*auto dest*: *length-wl-eq-stack*)
with *list-take-induct-tl2*[*OF this*(*1*)] *empty.IH empty* **show** *?case* **by** *simp*
next
case (*remove s s' e x xs w ws*)

then have *w'*: *wl s'* = (*w* − {*e*}) # *ws* **and** *s'*: *stack s'* = *x*#*xs* **and** *d'*: *discovered s'* = *discovered s* **by** *auto*
from *remove*(*4*,*5*) *s' w' remove.IH* **show** *?case*
by (*rule stack-wl-remove-induct*) (*simp-all add*: *d'*, *blast intro*!: *remove*(*9*))
next
case (*restrict s s' e x xs w ws*)

then have *w'*: *wl s'* = (*w* − {*e*}) # *ws* **and** *s'*: *stack s'* = *x*#*xs* **and** *d'*: *discovered s'* = *discovered s* **by** *auto*
from *restrict*(*4*,*5*) *s' w' restrict.IH* **show** *?case*
by (*rule stack-wl-remove-induct*) (*simp-all add*: *d'*, *blast intro*!: *restrict*(*8*))
next
case (*visit s s' e x xs w ws*) **hence** *s'*: *stack s'* = *e*#*x*#*xs* **and** *w'*: *wl s'* = *succs e*#(*w* − {*e*})#*ws* **and** *d'*: *discovered s'* = *insert e* (*discovered s*) **by** *simp-all*
from *visit*(*4*,*5*) *s' w' visit.IH* **show** *?case*
by (*rule stack-wl-visit-induct*) (*auto simp add*: *d'*)
qed *auto*

lemma *succs-discovered-subset-wl*:
dfs-constructable dfs s $\Longrightarrow$ *n* < *length* (*stack s*) $\Longrightarrow$ *succs* (*stack s* ! *n*) − *discovered s* − *dfs-restrict dfs* $\subseteq$ *wl s* ! *n*
by (*metis succs-discovered-subset-wl-all*)

lemma *empty-wl-succs-discovered*:
dfs-constructable dfs s $\Longrightarrow$ *n* < *length* (*stack s*) $\Longrightarrow$ *wl s* ! *n* = {} $\Longrightarrow$ *succs* (*stack s* ! *n*) − *dfs-restrict dfs* $\subseteq$ *discovered s*
using *succs-discovered-subset-wl*
by *blast*

lemma *wl-subset-succs-all*:
dfs-constructable dfs s $\Longrightarrow \forall\, n < \mathit{length}\ (\mathit{stack}\ s).\ \mathit{wl}\ s\ !\ n \subseteq \mathit{succs}\ (\mathit{stack}\ s\ !\ n)$
proof (*induction rule*: *dfs-constructable-induct*)
case (*empty s s'*) **hence** *length* (*wl s*) = *length* (*stack s*) **by** (*auto dest*: *length-wl-eq-stack*)
with *list-take-induct-tl2*[*OF this*(*1*)] *empty.IH empty* **show** *?case* **by** *simp*
next
case (*remove s s' e x xs w ws*) **hence** *w'*: *wl s'* = (*w* − {*e*}) # *ws* **and** *s'*: *stack s'* = *x*#*xs* **by** *simp-all*
from *remove*(*4*,*5*) *s' w' remove.IH* **show** *?case* **by** (*rule stack-wl-remove-induct*) *auto*
next
case (*restrict s s' e x xs w ws*) **hence** *w'*: *wl s'* = (*w* − {*e*}) # *ws* **and** *s'*: *stack s'* = *x*#*xs* **by** *simp-all*
from *restrict*(*4*,*5*) *s' w' restrict.IH* **show** *?case* **by** (*rule stack-wl-remove-induct*) *auto*
next
case (*visit s s' e x xs w ws*) **hence** *e*: *stack s'* = *e*#*x*#*xs wl s'* = *succs e*#(*w* − {*e*})#*ws* **by** *simp-all*
from *visit*(*4*,*5*) *e visit.IH* **show** *?case* **by** (*rule stack-wl-visit-induct*) *auto*
qed *simp*

lemma *wl-subset-succs*:
dfs-constructable dfs s $\Longrightarrow$ *n* < *length* (*stack s*) $\Longrightarrow$ *wl s* ! *n* ⊆ *succs* (*stack s* ! *n*)
by (*metis wl-subset-succs-all*)

lemma *wl-subset-verts*:
dfs-constructable dfs s $\Longrightarrow$ *x* ∈ *set* (*wl s*) $\Longrightarrow$ *x* ⊆ *V*
proof −
assume *c*: *dfs-constructable dfs s* **and** *x* ∈ *set* (*wl s*)
hence ∃ *n*. *x* = *wl s* ! *n* ∧ *n* < *length* (*wl s*) **using** *in-set-conv-nth*[*of x wl s*]
by *blast*
then obtain *n* **where** *x* = *wl s* ! *n n* < *length* (*wl s*) **by** *force*
with *length-wl-eq-stack*[*OF c*] *wl-subset-succs*[*OF c*] **have** *x* ⊆ *succs* (*stack s* ! *n*) **by** *simp*
with *succs-in-V* **show** *?thesis* **by** *auto*
qed

lemma *wl-finite*:
dfs-constructable dfs s $\Longrightarrow$ *x* ∈ *set* (*wl s*) $\Longrightarrow$ *finite x*
by (*metis wl-subset-verts finite-V finite-subset*)

lemma *discovered-subset-verts*:
dfs-constructable dfs s $\Longrightarrow$ *discovered s* ⊆ *V*
proof (*induct s rule*: *dfs-constructable-induct*)
case (*visit s s' e - - w*) **hence** *discovered s'* = *insert e* (*discovered s*) **by** *simp*
moreover
from *visit wl-subset-verts*[*OF visit*(*2*)] **have** *w* ⊆ *V* **by** *simp*

with *visit* **have** $e \in V$ **by** *auto*
ultimately show *?case* **using** ‹*discovered s* ⊆ *V*› **by** *auto*
qed *simp+*

lemma *discovered-finite*:
dfs-constructable dfs s ⟹ *finite* (*discovered s*)
by (*metis finite-V finite-subset discovered-subset-verts*)

lemma *ran-discover-finite*:
dfs-constructable dfs s ⟹ *finite* (*ran* (*discover s*))
by (*metis discovered-finite map-dom-ran-finite*)

lemma *stack-subset-discovered*:
dfs-constructable dfs s ⟹ *set* (*stack s*) ⊆ *discovered s*
by (*induct rule*: *dfs-constructable-induct*) *auto*

lemma *stack-subset-verts*:
dfs-constructable dfs s ⟹ *set* (*stack s*) ⊆ *V*
by (*metis discovered-subset-verts stack-subset-discovered subset-trans*)

lemma *stack-distinct*:
dfs-constructable dfs s ⟹ *distinct* (*stack s*)
proof (*induct rule*: *dfs-constructable-induct*)
case (*visit s s'* - *x xs*)
hence *tl*: *stack s* = *tl* (*stack s'*) **by** *simp*

{
assume *hd* (*stack s'*) ∉ *set* (*stack s*)
with *tl* **have** *hd* (*stack s'*) ∉ *set* (*tl* (*stack s'*)) **by** *simp*

moreover
from *tl* ‹*distinct* (*stack s*)› **have** *distinct* (*tl* (*stack s'*)) **by** *simp*

ultimately have *distinct* (*stack s'*) **by** (*cases stack s'*) *simp-all*
}

moreover
from *visit* **have** *hd* (*stack s'*) ∉ *discovered s* **using** *dfs-step-stack-not-discovered*[*OF dfs-next-dfs-step*] **by** *simp*

moreover
from *visit stack-subset-discovered* **have** *set* (*stack s*) ⊆ *discovered s* **by** *blast*
ultimately show *?case* **by** *auto*
qed *simp+*

lemma *last-stack-is-start*:
dfs-constructable dfs s ⟹ *stack s* ≠ [] ⟹ *last* (*stack s*) = *start s*
by (*induct rule*: *dfs-constructable-induct*) (*auto simp add*: *dfs-constr-start-def tl-last*)

lemma *start-discovered*:
dfs-constructable dfs s $\Longrightarrow$ *start s* $\in$ *discovered s*
proof (*induct rule*: *dfs-constructable.induct*)
case *step* **thus** *?case* **by** (*metis set-rev-mp dfs-step-discovered-subset dfs-next-dfs-step dfs-step-preserves-start dfs-next-wl-notempty*)
qed *simp*

lemma *start-not-restr*:
dfs-constructable dfs s $\Longrightarrow$ *start s* $\notin$ *dfs-restrict dfs*
by (*induction rule*: *dfs-constructable-induct*) *simp-all*

lemma *dfs-start-in-verts*:
dfs-constructable dfs s $\Longrightarrow$ *start s* $\in$ *V*
by (*metis start-discovered discovered-subset-verts set-rev-mp*)

lemma *finished-stack-eq-discovered*:
dfs-constructable dfs s $\Longrightarrow$ *finished s* $\cup$ *set* (*stack s*) = *discovered s*
proof (*induction rule*: *dfs-constructable-induct*)
case (*empty s s' x xs*)
hence *set* (*stack s*) = *set* (*stack s'*) $\cup$ {*hd* (*stack s*)} **by** (*fastforce intro*: *list.exhaust*)
with *empty* **have** *finished s'* $\cup$ *set* (*stack s'*) = *finished s* $\cup$ *set* (*stack s*) **by** *auto*
also note *empty.IH*
also from *empty* **have** *discovered s* = *discovered s'* **by** *simp*
finally show *?case* **.**
next
case (*visit s s' e x xs*) **then have** *fin-eq*: *finished s'* = *finished s* **by** *simp*

from *visit* **have** *dis-eq*: *discovered s'* = *insert e* (*discovered s*) **and** *stack s'* = *e* # *stack s* **by** *simp-all*
then have *set* (*stack s'*) = *insert e* (*set* (*stack s*)) **by** *auto*
hence *finished s'* $\cup$ *set* (*stack s'*) = *insert e* ((*finished s'*) $\cup$ *set* (*stack s*)) **by** *auto*
also with *fin-eq* **have** ... = *insert e* (*finished s* $\cup$ *set* (*stack s*)) **by** *simp*
also with *visit.IH* **have** ... = *insert e* (*discovered s*) **by** *simp*
also with *dis-eq* **have** ... = *discovered s'* **by** *simp*
finally show *?case* **.**
qed *simp+*

lemma *finished-subset-discovered*:
dfs-constructable dfs s $\Longrightarrow$ *finished s* $\subseteq$ *discovered s*
by (*auto dest*!: *finished-stack-eq-discovered*)

lemma *finished-subset-verts*:
dfs-constructable dfs s $\Longrightarrow$ *finished s* $\subseteq$ *V*
by (*metis finished-subset-discovered discovered-subset-verts subset-trans*)

lemma *finished-finite*:
dfs-constructable dfs s $\Longrightarrow$ *finite* (*finished s*)

by (*metis finished-subset-verts finite-V finite-subset*)

lemma *ran-finish-finite*:
dfs-constructable dfs s $\Longrightarrow$ *finite* (*ran* (*finish s*))
by (*metis map-dom-ran-finite finished-finite*)

lemma *finished-implies-discovered*:
dfs-constructable dfs s $\Longrightarrow$ *v* $\in$ *finished s* $\Longrightarrow$ *v* $\in$ *discovered s*
by (*auto dest*!: *finished-subset-discovered*)

lemma *not-discovered-implies-not-finished*:
dfs-constructable dfs s $\Longrightarrow$ *v* $\notin$ *discovered s* $\Longrightarrow$ *v* $\notin$ *finished s*
by (*metis finished-implies-discovered*)

lemma *discovered-non-stack-implies-finished*:
dfs-constructable dfs s $\Longrightarrow$ *v* $\in$ *discovered s* $\Longrightarrow$ *v* $\notin$ *set* (*stack s*) $\Longrightarrow$ *v* $\in$ *finished s*
by (*auto dest*!: *finished-stack-eq-discovered*)

lemma *discovered-not-finished-implies-stack*:
dfs-constructable dfs s $\Longrightarrow$ *v* $\in$ *discovered s* $\Longrightarrow$ *v* $\notin$ *finished s* $\Longrightarrow$ *v* $\in$ *set* (*stack s*)
by (*metis discovered-non-stack-implies-finished*)

lemma *discovered-not-restricted*:
dfs-constructable dfs s $\Longrightarrow$ *v* $\in$ *discovered s* $\Longrightarrow$ *v* $\notin$ *dfs-restrict dfs*
by (*induction rule*: *dfs-constructable-induct*) *auto*

lemma *finished-not-restricted*:
dfs-constructable dfs s $\Longrightarrow$ *v* $\in$ *finished s* $\Longrightarrow$ *v* $\notin$ *dfs-restrict dfs*
using *discovered-not-restricted finished-subset-discovered*
by *blast*

lemma *stack-not-restricted*:
dfs-constructable dfs s $\Longrightarrow$ *v* $\in$ *set* (*stack s*) $\Longrightarrow$ *v* $\notin$ *dfs-restrict dfs*
using *discovered-not-restricted stack-subset-discovered*
by *blast*

lemma *restricted-not-finished*:
dfs-constructable dfs s $\Longrightarrow$ *v* $\in$ *dfs-restrict dfs* $\Longrightarrow$ *v* $\notin$ *finished s*
using *finished-not-restricted* **by** *blast*

lemma *restricted-not-discovered*:
dfs-constructable dfs s $\Longrightarrow$ *v* $\in$ *dfs-restrict dfs* $\Longrightarrow$ *v* $\notin$ *discovered s*
using *discovered-not-restricted* **by** *blast*

lemma *restricted-not-stack*:
dfs-constructable dfs s $\Longrightarrow$ *v* $\in$ *dfs-restrict dfs* $\Longrightarrow$ *v* $\notin$ *set* (*stack s*)
using *stack-not-restricted* **by** *blast*

lemma *stack-implies-not-finished*:
dfs-constructable dfs s $\Longrightarrow$ *v* $\in$ *set* (*stack s*) $\Longrightarrow$ *v* $\notin$ *finished s*
proof (*induction rule*: *dfs-constructable-induct*)
case (*empty - - x*) **with** *stack-distinct*[*OF empty*(*2*)] **show** *?case* **by** (*cases v* = *x*) *auto*
next
case (*visit s s' e*) **then have** *f-eq*: *finish s* = *finish s'* **and** *e-stack*: *stack s'* = *e* # *stack s* **by** *simp-all*
with *visit*(*9*) *not-discovered-implies-not-finished*[*OF visit*(*2*)] **have** *e* $\notin$ *finished s* **by** *simp*
with *e-stack visit.IH visit*(*12*) *f-eq* **show** *?case* **by** *auto*
qed *simp+*

lemma *finished-implies-not-stack*:
dfs-constructable dfs s $\Longrightarrow$ *x* $\in$ *finished s* $\Longrightarrow$ *x* $\notin$ *set* (*stack s*)
by (*metis stack-implies-not-finished*)

lemma *stack-in-Suc-succs-restr-all*:
dfs-constructable dfs s $\Longrightarrow$ $\forall$ *n* < (*length* (*stack s*)) − *1*. (*stack s* ! *n*) $\in$ *succs* (*stack s* ! *Suc n*) − *dfs-restrict dfs*
proof (*induction rule*: *dfs-constructable-induct*)
case (*empty - - - xs*) **thus** *?case* **by** (*induct xs*) *auto*
next
case (*visit s s' e - - w*) **hence** *stack s'* = *e*#*stack s* **by** *simp*
with *wl-subset-succs*[*OF visit*(*2*), *of 0*] *visit* **have** *stack s'* ! *0* $\in$ *succs* (*stack s'* ! *Suc 0*) − *dfs-restrict dfs* **using** *dfs-next-stack-notempty*[*OF visit*(*1*), *THEN hd-conv-nth*] **by** *auto*

moreover
from *visit* **have** $\bigwedge$ *n*. *n* < (*length* (*stack s'*)) − *1* $\Longrightarrow$ *n* > *0* $\Longrightarrow$ (*stack s'* ! *n*) $\in$ *succs* (*stack s'* ! *Suc n*) − *dfs-restrict dfs* **by** *fastforce*

ultimately show *?case* **by** *blast*
qed *simp+*

lemma *stack-in-Suc-succs-restr*:
dfs-constructable dfs s $\Longrightarrow$ *n* < *length* (*stack s*) − *1*
$\Longrightarrow$ *stack s* ! *n* $\in$ *succs* (*stack s* ! *Suc n*) − *dfs-restrict dfs*
by (*metis stack-in-Suc-succs-restr-all*)

lemma *Suc-stack-stack-in-restr-edges*:
dfs-constructable dfs s $\Longrightarrow$ *n* < *length* (*stack s*) − *1* $\Longrightarrow$ (*stack s* ! *Suc n*, *stack s* ! *n*) $\in$ (*rel-restrict E* (*dfs-restrict dfs*))
using *succs-implies-edge rel-restrict-notR stack-in-Suc-succs-restr restricted-not-stack*
by (*smt Diff-iff nth-mem*)

lemma *Suc-stack-stack-in-edges*:

dfs-constructable dfs s $\Longrightarrow$ $n <$ *length* (*stack s*) $-$ *1* $\Longrightarrow$ (*stack s* ! *Suc n*, *stack s* ! *n*) $\in$ *E*
using *Suc-stack-stack-in-restr-edges rel-restrict-lift*
by *metis*

lemma *Suc-reachable-stack*:
dfs-constructable dfs s $\Longrightarrow$ $n <$ (*length* (*stack s*)) $-$ *1* $\Longrightarrow$ *stack s* ! *Suc n* $\rightarrow\backslash$*dfs-restrict dfs+ stack s* ! *n*
using *restr-reachable1-trancl Suc-stack-stack-in-restr-edges*
by *blast*

lemma *prev-reachable-stack*:
dfs-constructable dfs s $\Longrightarrow$ $n <$ *length* (*stack s*) $\Longrightarrow$ $m < n \Longrightarrow$ *stack s* ! *n* $\rightarrow\backslash$*dfs-restrict dfs+ stack s* ! *m*
by (*metis restr-reachable1-trancl nth-step-trancl Suc-stack-stack-in-restr-edges*)

lemma *tl-reachable-stack-hd*:
assumes *constr*: *dfs-constructable dfs s*
and $x \in$ *set* (*tl* (*stack s*))
shows $x \rightarrow\backslash$*dfs-restrict dfs+ hd* (*stack s*)
proof $-$
from *assms* **have** *ne*: *stack s* $\neq$ [] **by** *auto*
from *assms* **obtain** *n* **where** $n <$ *length* (*tl* (*stack s*)) *tl* (*stack s*) ! *n* $=$ *x* **unfolding** *in-set-conv-nth* **by** *auto*
with *nth-tl*[*OF ne*] **have** *Suc n* $<$ *length* (*stack s*) *stack s* ! (*Suc n*) $=$ *x* **by** *simp-all*
moreover with *prev-reachable-stack*[*OF constr*, *of Suc n 0*] **have** *stack s* ! *Suc n* $\rightarrow\backslash$*dfs-restrict dfs+ stack s* ! *0* **by** *simp*
ultimately show *?thesis* **using** *hd-conv-nth*[*OF ne*] **by** *simp*
qed

lemma *start-reachable-stack*:
assumes *constr*: *dfs-constructable dfs s*
and $x \in$ *set* (*stack s*)
shows *start s* $\rightarrow\star$ *x*
proof (*cases x* $=$ *start s*)
case *True* **with** *self-reachable dfs-start-in-verts*[*OF constr*] **show** *?thesis* **by** *simp*
next
case *False* **with** *assms* **have** *stack s* $\neq$ [] **by** *auto*
with *last-stack-is-start*[*OF constr*] *last-conv-nth* **have** *s-nth*: *start s* $=$ *stack s* ! (*length* (*stack s*) $-$ *1*) **by** *metis*

from ‹*x* $\in$ *set* (*stack s*)› **obtain** *n* **where** *n*: $n <$ *length* (*stack s*) *stack s* ! *n* $=$ *x* **unfolding** *in-set-conv-nth* **by** *auto*

have $n <$ *length* (*stack s*) $-$ *1*
proof (*rule ccontr*)
assume $\neg$ $n <$ *length* (*stack s*) $-$ *1*
with *n* **have** *n* $=$ *length* (*stack s*) $-$ *1* **by** *auto*

```
    with s-nth n False show False by simp
  qed
   with prev-reachable-stack[OF constr] have stack s ! (length (stack s) − 1)
→\dfs-restrict dfs+ stack s ! n by auto
  with s-nth n have start s →\dfs-restrict dfs+ x by auto
  thus ?thesis by blast
qed

lemma ran-discover-lt-counter:
  dfs-constructable dfs s ⟹ v ∈ ran (discover s) ⟹ v < counter s
proof (induction rule: dfs-constructable-induct)
  case (visit s s' e) hence
        d': discover s' = (discover s)(e ↦ counter s)
    and c': counter s' = Suc (counter s)
  by simp-all
   with visit(9) have r': ran (discover s') = ran (discover s) ∪ {counter s} by
auto

  from c' show ?case
  proof (cases v = counter s)
    case False with visit.prems r' have v ∈ ran (discover s) by simp
    with visit.IH c' show ?thesis by simp
  qed simp
qed auto

lemma discover-lt-counter:
  assumes constr: dfs-constructable dfs s
  and     nn:     v ∈ discovered s
  shows δ s v < counter s
proof −
  from nn have δ s v ∈ ran (discover s) by (auto intro: ranI)
  with ran-discover-lt-counter[OF constr] show ?thesis .
qed

lemma ran-finish-lt-counter:
  dfs-constructable dfs s ⟹ v ∈ ran (finish s) ⟹ v < counter s
proof (induction rule: dfs-constructable-induct)
  case (empty s s' x xs)
  hence f': finish s' = (finish s)(x ↦ counter s)
    and c': counter s' = Suc (counter s)
    by simp-all

  with stack-implies-not-finished[OF empty(2)] empty(4) have x ∉ finished s by
simp
  with f' have r': ran (finish s') = ran (finish s) ∪ {counter s} by auto

  from c' show ?case
  proof (cases v = counter s)
    case False with empty.prems r' have v ∈ ran (finish s) by simp
```

```
    with f' c' empty.IH show ?thesis by simp
  qed simp
qed auto

lemma finish-lt-counter:
  assumes constr: dfs-constructable dfs s
  and     nn:     v ∈ finished s
  shows φ s v < counter s
proof −
  from nn have φ s v ∈ ran (finish s) by (auto intro: ranI)
  with ran-finish-lt-counter[OF constr] show ?thesis .
qed

lemma finish-gt-discover:
  dfs-constructable dfs s ⟹ v ∈ finished s ⟹ φ s v > δ s v
proof (induction rule: dfs-constructable-induct)
  case (empty s s' x xs)
  hence f': finish s' = (finish s)(x ↦ counter s)
    and d': discover s = discover s'
    and c': counter s' = Suc (counter s)
    by simp-all

  show ?case
  proof (cases v = x)
    case False with empty.prems f' have v ∈ finished s φ s v = φ s' v by auto
    with d' empty.IH show ?thesis by auto
  next
    case True with f' have v ∈ finished s' by simp
     with finished-implies-discovered[OF empty(3)] d' have v ∈ discovered s by
simp
    with discover-lt-counter[OF empty(2)] d' have δ s' v < counter s by simp
    with f' c' True show ?thesis by simp
  qed
next
  case (visit s s' e)
  hence d': discover s' = (discover s)(e ↦ counter s)
    and f': finish s = finish s'
    by simp-all

  show ?case
  proof (cases v = e)
    case False with d' f' visit show ?thesis by force
  next
   case True with f' have v ∉ finished s' using not-discovered-implies-not-finished[OF
visit(2) visit(9)] by simp
    with visit.prems show ?thesis by contradiction
  qed
qed simp+
```

lemma *discover-finish-distinct*:
dfs-constructable dfs s $\Longrightarrow$ *ran* (*finish s*) $\cap$ *ran* (*discover s*) = {}
proof (*induction rule*: *dfs-constructable-induct*)
case (*empty s s' x xs*) **then have** $x \notin$ *finished s* **using** *stack-implies-not-finished*[*OF empty*(*2*)] *hd-in-set* **by** *simp*
with *empty* **have** *ran* (*finish s'*) = *insert* (*counter s*) (*ran* (*finish s*)) *ran* (*discover s'*) = *ran* (*discover s*)
by *auto*
moreover
with *ran-discover-lt-counter*[*OF empty*(*2*)] **have** *counter s* $\notin$ *ran* (*discover s'*) **by** *auto*
ultimately show *?case* **using** *empty.IH* **by** *auto*
next
case (*visit s s' e*) **hence** *d'*: *discover s'* = *discover s* (*e* $\mapsto$ *counter s*) **by** *simp*
with *visit*(*9*) **have** *ran* (*discover s'*) = *insert* (*counter s*) (*ran* (*discover s*)) **by** *auto*

moreover
from *visit* **have** *finish s* = *finish s'* **by** *simp*

moreover
with *ran-finish-lt-counter*[*OF visit*(*2*)] **have** *counter s* $\notin$ *ran* (*finish s'*) **by** *auto*

ultimately show *?case* **using** *visit.IH* **by** *auto*
qed *simp+*

lemma *discover-neq-finish*:
assumes *constr*: *dfs-constructable dfs s*
and *verts*: *v* $\in$ *V w* $\in$ *V*
and *discovered*: *v* $\in$ *discovered s*
shows *discover s v* $\neq$ *finish s w*
using *discovered*
proof (*cases finish s w*)
case (*Some x*) **then have** $x \notin$ *ran* (*discover s*) **using** *discover-finish-distinct*[*OF constr*] **by** (*auto intro*: *ranI*)
with *Some discovered* **show** *?thesis* **by** (*metis ranI*)
qed *auto*

lemma *discover-card-ran-dom*:
dfs-constructable dfs s $\Longrightarrow$ *card* (*ran* (*discover s*)) = *card* (*dom* (*discover s*))
proof (*induction rule*: *dfs-constructable-induct*)
case (*visit s s'*) **with** *discovered-finite*[*OF visit*(*2*)] *visit*(*9*) **have** *card* (*dom* (*discover s'*)) = *card* (*dom* (*discover s*)) + *1* **by** *simp*

moreover
from *visit ran-discover-lt-counter*[*OF visit*(*2*)] **have** *ran* (*discover s'*) = *insert* (*counter s*) (*ran* (*discover s*)) *counter s* $\notin$ *ran* (*discover s*) **by** *auto*
with *ran-discover-finite*[*OF visit*(*2*)] **have** *card* (*ran* (*discover s'*)) = *card* (*ran* (*discover s*)) + *1* **by** *simp*

ultimately show *?case* **using** *visit.IH* **by** *simp*
qed *simp+*

lemma *discover-neq-discover*:
assumes *constr*: *dfs-constructable dfs s*
and *verts*: $v \in V$ $w \in V$
and *ne*: $v \neq w$
and *discovered*: $v \in$ *discovered s*
shows *discover s v* $\neq$ *discover s w*
proof −
from *map-card-eq-iff*[*OF discovered-finite*, *OF constr*] *assms discover-card-ran-dom*[*OF constr*] **have** *discover s v* = *discover s w* $\longleftrightarrow$ $v = w$ **by** *auto*
with *ne* **show** *?thesis* **by** *simp*
qed

lemma *finish-card-ran-dom*:
dfs-constructable dfs s $\Longrightarrow$ *card* (*ran* (*finish s*)) = *card* (*dom* (*finish s*))
proof (*induction rule*: *dfs-constructable-induct*)
case (*empty s s' x*) **then have** *f'*: *finish s'* = *finish s* ($x \mapsto$ *counter s*) **by** *simp*
with *empty* **have** $*$: $x \notin$ *finished s* **using** *stack-implies-not-finished* **by** *fastforce*
with *finished-finite*[*OF empty*(*2*)] *f'* **have** *card* (*dom* (*finish s'*)) = *card* (*dom* (*finish s*)) + *1* **by** *simp*

moreover
from *f' ran-finish-lt-counter*[*OF empty*(*2*)] $*$ **have** *ran* (*finish s'*) = *insert* (*counter s*) (*ran* (*finish s*)) *counter s* $\notin$ *ran* (*finish s*) **by** *auto*
with *ran-finish-finite*[*OF empty*(*2*)] **have** *card* (*ran* (*finish s'*)) = *card* (*ran* (*finish s*)) + *1* **by** *simp*

ultimately show *?case* **using** *empty.IH* **by** *simp*
qed *simp+*

lemma *finish-neq-finish*:
assumes *constr*: *dfs-constructable dfs s*
and *verts*: $v \in V$ $w \in V$
and *ne*: $v \neq w$
and *discovered*: $v \in$ *finished s*
shows *finish s v* $\neq$ *finish s w*
proof −
from *map-card-eq-iff*[*OF finished-finite*, *OF constr*] *assms finish-card-ran-dom*[*OF constr*] **have** *finish s v* = *finish s w* $\longleftrightarrow$ $v = w$ **by** *auto*
with *ne* **show** *?thesis* **by** *simp*
qed

lemma *prev-stack-discover-all*:
dfs-constructable dfs s $\Longrightarrow \forall\ n <$ *length* (*stack s*). $\forall\ v \in$ *set* (*drop* (*Suc n*) (*stack s*)). $\delta\ s$ (*stack s* ! *n*) > $\delta\ s\ v$
proof (*induction rule*: *dfs-constructable-induct*)

case (*visit s s′ e*)
hence *d′*: *discover s′* = (*discover s*)(*e* ↦ *counter s*)
and *s′*: *stack s′* = *e* # *stack s*
by *simp-all*
with *stack-distinct*[*OF visit*(*3*)] **have** *t-ni*: *e* ∉ *set*(*stack s*) **by** *simp*

from *visit*(*2*) **have** ⋀*v*. *v* ∈ *set* (*stack s*) ⟹ *δ s v* < *counter s* **by** (*metis discover-lt-counter set-mp stack-subset-discovered*)
with *t-ni d′* **have** ⋀*v*. *v* ∈ *set* (*stack s*) ⟹ *δ s′ v* < *δ s′ e* **by** *auto*
with *s′* **have** ⋀*v*. *v* ∈ *set* (*drop* (*Suc 0*) (*stack s′*)) ⟹ *δ s′ v* < *δ s′* (*stack s′* ! *0*) **by** *auto*

moreover
from *s′ d′ t-ni* **have** ⋀ *n*. *n* < (*length* (*stack s′*)) ⟹ *n* > *0* ⟹ *δ s′* (*stack s′* ! *n*) = *δ s* (*stack s′* ! *n*) **by** *auto*
with *visit.IH s′ d′ t-ni* **have** ⋀ *n*. *n* < (*length* (*stack s′*)) − *1* ⟹ *n* > *0* ⟹ ∀ *v* ∈ *set* (*drop* (*Suc n*) (*stack s′*)). *δ s′* (*stack s′* ! *n*) > *δ s′ v* **by** (*smt drop-Cons′ fun-upd-other in-set-dropD length-drop nth-Cons-pos*)

ultimately show *?case* **by** (*smt length-drop length-pos-if-in-set*)
qed *auto*

lemma *prev-stack-discover*:
dfs-constructable dfs s ⟹ *n* < *length* (*stack s*) ⟹ *v* ∈ *set* (*drop* (*Suc n*) (*stack s*)) ⟹ *δ s* (*stack s* ! *n*) > *δ s v*
by (*metis prev-stack-discover-all*)

lemma *Suc-stack-discover*:
assumes *constr*: *dfs-constructable dfs s*
and *n*: *n* < (*length* (*stack s*)) − *1*
shows *δ s* (*stack s* ! *n*) > *δ s* (*stack s* ! *Suc n*)
proof −
from *prev-stack-discover assms* **have** ⋀ *v*. *v* ∈ *set* (*drop* (*Suc n*) (*stack s*)) ⟹ *δ s* (*stack s* ! *n*) > *δ s v* **by** *fastforce*
moreover from *n* **have** *stack s* ! *Suc n* ∈ *set* (*drop* (*Suc n*) (*stack s*)) **using** *in-set-conv-nth* **by** *fastforce*
ultimately show *?thesis* **.**
qed

lemma *tl-lt-stack-hd-discover*:
assumes *dfs-constructable dfs s*
and *notempty*: *stack s* ≠ []
and *x* ∈ *set* (*tl* (*stack s*))
shows *δ s x* < *δ s* (*hd* (*stack s*))
proof −
from *notempty* **obtain** *y ys* **where** *stack s* = *y*#*ys* **by** (*metis list.exhaust*)
with *assms* **show** *?thesis*
using *prev-stack-discover*
by (*cases ys*) *force*+

qed

lemma *obtain-discovered-predecessor*:
assumes *dfs-constructable tdfs s*
and $v \in$ *discovered s*
and $v \neq$ *start s*
obtains *y* **where** $v \in$ *succs y* **and** $y \in$ *discovered s* **and** $\delta\ s\ v > \delta\ s\ y$
using *assms*
proof *induction*
case (*visit s s' e x xs w*) **hence** $*$: *discover s'* $=$ *discover s* $(e \mapsto$ *counter s*$)$ *start s'* $=$ *start s stack s'* $= e\#x\#xs$ **by** *simp-all*

show *?case*
proof (*cases v* $=$ *e*)
case *True* **with** *visit* **have** $v \in w$ **by** *simp*
with *visit wl-subset-succs*[*OF visit*(*2*), *of 0*] **have** $v \in$ *succs x* **by** *auto*
moreover from *tl-lt-stack-hd-discover*[*OF visit*(*3*), *of x*] $*$ *True* **have** $\delta\ s'\ x < \delta\ s'\ v$ **by** *simp*
moreover from *stack-subset-discovered*[*OF visit*(*3*)] $*$ **have** $x \in$ *discovered s'* **by** *auto*
ultimately show *?thesis* **using** *visit.prems* **by** *simp*
next
case *False* **with** *visit* $*$ **have** $**$: $v \in$ *discovered s* $v \neq$ *start s* **by** *auto*
{
fix *y*
assume *A*: $v \in$ *succs y* $y \in$ *discovered s* $\delta\ s\ y < \delta\ s\ v$
with *visit*(*9*) *False* $*$ **have** $y \in$ *discovered s'* $\delta\ s'\ y < \delta\ s'\ v$ **by** *auto*
with *A visit.prems* **have** *thesis* **by** *simp*
}
with *visit.IH*[*OF* - $**$] **show** *?thesis* **.**
qed
qed *simp-all*

lemma *stack-finish-discover*:
dfs-constructable dfs s $\Longrightarrow$ $v \in$ *set* (*stack s*) $\Longrightarrow$ $w \in$ *finished s* $\Longrightarrow$ $\varphi\ s\ w < \delta\ s\ v \vee \delta\ s\ v < \delta\ s\ w$
proof (*induction rule*: *dfs-constructable-induct*)
case (*empty s s' x*) **then have**
d': *discover s* $=$ *discover s'* **and**
f': *finish s'* $=$ *finish s* $(x \mapsto$ *counter s*$)$ **and**
v: $v \in$ *set* (*stack s*)
by *simp-all*

show *?case*
proof (*cases w* $=$ *x*)
case *True* **with** *empty* **have** $\delta\ s\ w > \delta\ s\ v$ **using** *prev-stack-discover*[*OF empty*(*2*)] **by** *fastforce*
with *d'* **show** *?thesis* **by** *simp*
next

```
    case False with empty.prems empty.IH v f' d' show ?thesis by force
  qed
next
  case (visit s s' e)
  hence d': discover s' = discover s (e ↦ counter s)
    and s': stack s' = e # stack s
    and f': finish s = finish s'
    by simp-all

  show ?case
  proof (cases v = e)
    case True with d' have δ s' v = counter s by simp
   moreover from visit.prems f' have φ s' w < counter s using finish-lt-counter[OF
visit(2)] by simp
    ultimately show ?thesis by simp
  next
    case False with visit.prems s' d' have v ∈ set (stack s) and discover s v =
discover s' v by simp-all
    with f' visit.prems visit.IH have φ s' w < δ s' v ∨ δ s' v < δ s w by force

   moreover from visit.prems s' have w ≠ e using finished-implies-not-stack[OF
visit(3)] by simp
    with d' have discover s w = discover s' w by simp
    ultimately show ?thesis by simp
  qed
qed simp+

lemma interval-inclusion:
  assumes dfs-constructable dfs s
  and v ∈ discovered s
  and w ∈ finished s
  and δ s v < δ s w
  and v ∉ finished s ∨ δ s w < φ s v
  shows v ∉ finished s ∨ φ s w < φ s v
using assms
proof induction
  case (empty s s' x) hence ne: v ≠ w and f': finish s' = finish s (x↦counter s)
by auto
  show ?case
  proof (cases v = x)
    case True with f' ne empty.prems have w ∈ finished s and φ s' v = counter
s  by auto
     moreover with finish-lt-counter[OF empty(2)] empty.prems have counter s
> φ s w by simp
    with f' ne True have counter s > φ s' w by simp
    ultimately show ?thesis by simp
  next
    case False
    {
```

```
      assume v ∈ finished s
     moreover with empty False have v ∈ finished s' v ∈ discovered s by simp-all
      moreover with empty False have ∗: δ s w < φ s v δ s v < δ s w by auto
      moreover have w ≠ x
      proof (rule notI)
        assume w = x
        with empty.hyps have w ∈ set (stack s) by simp
        with stack-finish-discover[OF empty(2)] ‹v ∈ finished s› ∗ show False by
force
      qed
      with empty have w ∈ finished s by auto
      ultimately have φ s w < φ s v using empty.IH by simp
      with False f' ‹w≠x› have ?thesis by simp
    }
    with False f' show ?thesis by force
  qed
next
  case (visit s s' e) hence
    f': finish s' = finish s and
    d': discover s' = discover s (e ↦ counter s) by simp-all

  from visit have w ≠ e using stack-implies-not-finished[OF visit(3)] by auto

  show ?case
  proof (cases v = e)
    case True with visit(9) have v ∉ discovered s by simp
     with not-discovered-implies-not-finished[OF visit(2)] have v ∉ finished s by
simp
    with f' show ?thesis by simp
  next
    case False with ‹w ≠ e› visit.prems f' d' have v ∈ discovered s w ∈ finished
s δ s v < δ s w v ∉ finished s ∨ δ s w < φ s v by auto
    with visit.IH have v ∉ finished s ∨ φ s w < φ s v .
    with f' show ?thesis by simp
  qed
qed simp-all

lemma correct-order:
  assumes constr: dfs-constructable dfs s
  and verts: v ∈ finished s w ∈ finished s
  and disc: δ s v < δ s w
  shows φ s v < δ s w ∨ φ s v > φ s w
proof (cases δ s w < φ s v)
  case True with interval-inclusion finished-implies-discovered assms show ?thesis
by metis
next
  case False hence φ s v ≤ δ s w by simp
  moreover from assms finished-subset-verts have v ∈ V w ∈ V by auto
  with discover-neq-finish finished-implies-discovered assms have discover s w ≠
```

finish s v **by** *metis*
with *verts(1) finished-implies-discovered[OF constr verts(2)]* **have** $\delta\ s\ w \neq \varphi\ s\ v$ **by** *auto*
ultimately have $\varphi\ s\ v < \delta\ s\ w$ **by** *simp*
thus *?thesis* **..**
qed

lemma *discover-finish-implies-reach*:
dfs-constructable dfs s $\Longrightarrow$
$v \in$ *discovered s* $\Longrightarrow$ $w \in$ *discovered s* $\Longrightarrow$
$\delta\ s\ v < \delta\ s\ w \Longrightarrow v \notin$ *finished s* $\vee$ ($w \in$ *finished s* $\wedge\ \varphi\ s\ v > \varphi\ s\ w$)
$\Longrightarrow v \rightarrow \backslash$*dfs-restrict dfs+ w*
proof (*induction rule*: *dfs-constructable-induct*)
case (*empty s s' x*) **hence**
s': *discover s* = *discover s'* **and**
f': *finish s'* = *finish s* ($x \mapsto$ *counter s*)
by *simp-all*

show *?case*
proof (*cases v* = *x*)
case *True* **with** *empty f'* **have** $v \notin$ *finished s* **and** *finish s' w* = *finish s w*
using *stack-implies-not-finished[OF empty(2)]* **by** *auto*
with *empty.prems empty.IH s'* **show** *?thesis* **by** *force*
next
case *False* **with** *f'* **have** *fv*: *finish s v* = *finish s' v* **by** *simp*
show *?thesis*
proof (*cases w* $\in$ *finished s'*)
case *False* **with** *empty.IH s' empty.prems fv* **show** *?thesis* **by** *fastforce*
next
case *True* **note** *wf'* = *this*
thus *?thesis*
proof (*cases v* $\in$ *finished s'*)
case *True* **with** *empty(12) wf'* **have** $*$: $\varphi\ s'\ w < \varphi\ s'\ v$ **by** *simp*
have $w \neq x$
proof (*rule notI*)
assume $w = x$
with *f'* **have** $\varphi\ s'\ w$ = *counter s* **by** *simp*

moreover
from *True fv* **have** $v \in$ *finished s* **by** *auto*
hence $\varphi\ s\ v <$ *counter s* **using** *finish-lt-counter[OF empty(2)]* **by** *simp*
with *fv* **have** $\varphi\ s'\ v <$ *counter s* **by** *simp*
ultimately show *False* **using** $*$ **by** *simp*
qed
with $*$ *fv f' wf'* **have** $w \in$ *finished s* $\wedge\ \varphi\ s\ w < \varphi\ s\ v$ **by** *simp*
with *empty.prems s' empty.IH* **show** *?thesis* **by** *simp*
next
case *False* **with** *empty.prems empty.IH s' f'* **show** *?thesis* **by** *simp*
qed

```
    qed
  qed
next
  case (visit s s' e)
  hence d': discover s' = discover s (e ↦ counter s)
    and s': stack s' = e # stack s
    and f': finish s' = finish s
    and ne: v ≠ w
    by auto

  show ?case
  proof (cases v = e)
    case True with d' ne have δ s' v = counter s discover s w = discover s' w
by auto

    moreover
    then have w ∈ discovered s using visit.prems by auto
    hence δ s w < counter s using discover-lt-counter[OF visit(2)] by simp

    ultimately have False using visit.prems by simp
    thus ?thesis ..
  next
    case False with d' have dv: discover s v = discover s' v by simp
    show ?thesis
    proof (cases w = e)
      case True with s' have w ∉ finished s' using stack-implies-not-finished[OF
visit(3)] by simp
    with visit.prems have v ∈ set (stack s') using discovered-not-finished-implies-stack[OF
visit(3)] by simp
      with False True s' show ?thesis using tl-reachable-stack-hd[OF visit(3)] by
simp
    next
      case False with d' have discover s w = discover s' w by simp
     with dv f' visit.prems visit.IH show ?thesis by (cases v ∈ finished s') force+
    qed
  qed
qed simp+

lemma no-reach-discover-finish:
  assumes constr: dfs-constructable dfs s
  and no-reach: ¬v →⋆ w
  shows v ∉ discovered s ∨ w ∉ discovered s ∨ w ∉ finished s ∨ δ s v > δ s w ∨
(v ∈ finished s ∧ φ s v < φ s w)
proof (cases v ∈ V ∧ w ∈ V)
 case False hence w ∉ discovered s ∨ v ∉ discovered s using discovered-subset-verts[OF
constr] by auto
  thus ?thesis by simp
next
  case True hence verts: v ∈ V w ∈ V by simp-all
```

with *no-reach* **have** *ne*: $v \neq w$ **using** *reachable-ewalk*[*of v w*] *ewalk-empty-iff*[*of v w*] **by** *simp*
from *no-reach* **have** $\neg\ v{\rightarrow}\backslash$*dfs-restrict dfs*+ w **by** (*metis reachable1-reachable restr-reachable1-reachable1*)
thus *?thesis*
apply (*rule contrapos-np*)
apply (*rule discover-finish-implies-reach*[*OF constr*, *of v w*])
apply (*insert discover-neq-discover*[*OF constr verts ne*] *finish-neq-finish*[*OF constr verts ne*])
by *force*+
qed

lemma *both-no-reach-discover-finish*:
assumes *constr*: *dfs-constructable dfs s*
and *no-reach*: $\neg\ v \rightarrow\star\ w\ \neg\ w \rightarrow\star\ v$
and *fin*: $v \in$ *finished s* $w \in$ *finished s*
shows $\delta\ s\ w > \varphi\ s\ v \vee \varphi\ s\ w < \delta\ s\ v$
proof −
from *constr fin* **have** *disc*: $v \in$ *discovered s* $w \in$ *discovered s* **using** *finished-implies-discovered* **by** *metis*+
with *constr no-reach no-reach-discover-finish fin* **have** $\delta\ s\ v > \delta\ s\ w \vee \varphi\ s\ v < \varphi\ s\ w\ \delta\ s\ w > \delta\ s\ v \vee \varphi\ s\ w < \varphi\ s\ v$ **by** *metis*+
thus *?thesis* **using** *correct-order constr fin* **by** *smt*
qed

lemma *rev-stack-vwalk-or-empty*:
assumes *dfs-constructable dfs s*
shows *vwalk* (*rev* (*stack s*)) $\mathcal{G} \vee$ *stack* $s = []$
using *assms*
proof (*induction*)
case (*empty s s'*)
hence *vwalk* (*butlast* (*rev* (*stack s*))) $\mathcal{G} \vee$ *butlast* (*rev* (*stack s*)) $= []$
using *vwalk-butlast*[*of rev* (*stack s*)] **by** *simp*
with *empty* **show** *?case* **by** (*simp add*: *butlast-rev-tl*)
next
case (*visit s s' e x xs*) **with** *wl-subset-verts*[*OF visit*(*2*)] **have** $e \in V$ **by** *auto*

moreover
from *visit* **have** $x \in V$ **and** *set xs* $\subseteq V$ **using** *stack-subset-verts*[*OF visit*(*2*)] **by** *auto*

moreover
from *visit wl-subset-succs*[*OF visit*(*2*), *of 0*] **have** $e \in$ *succs x* **by** *auto*
with *visit* **have** *set* (*vwalk-edges* (*rev* (*stack s*))) $\subseteq E$ **and** $(x,e) \in E$ **unfolding** *succs-def* **by** *force*+
with *visit*(*4*) **have** *set* (*vwalk-edges* (*rev* (*stack s*) @ [*e*])) $\subseteq E$ **using** *vwalk-edges-append*[*of rev* (*stack s*)] **by** *simp*

ultimately have *vwalk* (*rev* (*stack s*) @ [*e*]) $\mathcal{G}$

using *visit* **by** (*intro vwalkI*) *simp-all*

with *visit* **show** *?case* **by** *simp*
qed *auto*

lemma *rev-stack-vwalk*:
dfs-constructable dfs s $\Longrightarrow$ *stack s* $\neq$ [] $\Longrightarrow$ *vwalk* (*rev* (*stack s*)) $\mathcal{G}$
by (*metis rev-stack-vwalk-or-empty*)

lemma *discover-start-eq-0*:
dfs-constructable dfs s $\Longrightarrow$ δ *s* (*start s*) = *0*
proof (*induct rule*: *dfs-constructable-induct*)
case *start* **thus** *?case* **unfolding** *dfs-constr-start-def* **by** *simp*
next
case (*visit s s' e*) **hence** *start'*: δ *s* (*start s'*) = *0* **using** *dfs-step-preserves-start*[*OF dfs-next-dfs-step*] **by** *auto*

from *visit.hyps* **have** *start s* = *last* (*stack s*) **using** *last-stack-is-start*[*OF visit*(*2*)] **by** *simp*
with *dfs-step-preserves-start*[*OF dfs-next-dfs-step*] *visit* **have** *start s'* = *last* (*stack s*) **by** *auto*
with *visit stack-distinct*[*OF visit*(*3*)] **have** *e* $\neq$ *start s'* **by** *auto*
with *start' visit* **show** *?case* **by** *simp*
qed *auto*

lemma *discovered-neq-0*:
assumes *constr*: *dfs-constructable dfs s*
and *discovered*: *v* $\in$ *discovered s*
and *ne*: *v* $\neq$ *start s*
shows δ *s v* > *0*
proof −
from *discovered constr* **have** *v* $\in$ *V start s* $\in$ *V start s* $\in$ *discovered s* **using** *discovered-subset-verts dfs-start-in-verts start-discovered* **by** *blast+*
with *ne discovered* **have** δ *s* (*start s*) $\neq$ δ *s v* **using** *discover-neq-discover*[*OF constr*] **by** *force*
hence δ *s v* $\neq$ *0* **using** *discover-start-eq-0*[*OF constr*] **by** *simp*
thus *?thesis* **by** *simp*
qed

lemma *discover-gt-start*:
dfs-constructable dfs s $\Longrightarrow$ *v* $\in$ *discovered s* $\Longrightarrow$ *v* $\neq$ *start s* $\Longrightarrow$ δ *s v* > δ *s* (*start s*)
by (*metis discovered-neq-0 discover-start-eq-0*)

lemma *start-restr-reach-discovered*:
assumes *constr*: *dfs-constructable dfs s*
and *stack*: *stack s* $\neq$ []
and *discovered*: *v* $\in$ *discovered s*
and *ne*: *v* $\neq$ *start s*

shows *start s* $\rightarrow\backslash$*dfs-restrict dfs+ v*
proof −
from *assms* **have** $\delta\ s\ v > \delta\ s$ (*start s*) **using** *discover-gt-start* **by** *metis*

moreover
from *stack* **have** *start s* ∈ *set* (*stack s*) **using** *last-in-set last-stack-is-start*[*OF constr*] **by** *force*
with *stack-implies-not-finished*[*OF constr*] **have** *start s* ∉ *finished s* **by** *simp*

ultimately show *?thesis* **using** *discover-finish-implies-reach*[*OF constr start-discovered*[*OF constr*] *discovered*] **by** *auto*
qed

lemma *start-reach-discovered*:
dfs-constructable dfs s ⟹ *stack s* ≠ [] ⟹ *v* ∈ *discovered s* ⟹ *start s* →⋆ *v*
by (*metis dfs-start-in-verts restr-reachable1-reachable self-reachable start-restr-reach-discovered verts-reduce*)

lemma *constr-from-restr-reach-stack*:
assumes *constr*: *s* ∈ *dfs-constr-from dfs x*
and *stack*: *v* ∈ *set* (*stack s*)
and *ne*: *v* ≠ *x*
shows *x* $\rightarrow\backslash$*dfs-restrict dfs+ v*
proof −
from *stack* **have** *sne*: *stack s* ≠ [] **by** *auto*

from *constr* **have** *c*: *dfs-constructable dfs s* **using** *dfs-constr-from-constructable* **by** *metis*
with *stack* **have** *in-d*: *v* ∈ *discovered s* **using** *stack-subset-discovered* **by** *auto*

from *constr* **have** *start s* = *x* **using** *constr-from-implies-start* **by** *blast*
thus *?thesis* **using** *start-restr-reach-discovered*[*OF c sne in-d*] *ne* **by** *simp*
qed

lemma *constr-from-reach-stack*:
assumes *constr*: *s* ∈ *dfs-constr-from dfs x*
and *stack*: *v* ∈ *set* (*stack s*)
shows *x* →⋆ *v*
proof (*cases x* = *v*)
case *True* **with** *stack stack-subset-verts constr* **have** *v* ∈ *V* **by** *auto*
with *True self-reachable* **show** *?thesis* **by** *simp*
next
case *False* **with** *constr-from-restr-reach-stack*[*OF assms*] **show** *?thesis* **by** *blast*
qed

lemma *finished-implies-succs-discovered*:
dfs-constructable dfs s ⟹ *v* ∈ *finished s* ⟹ *succs v* − *dfs-restrict dfs* ⊆ *discovered s*
proof (*induction rule*: *dfs-constructable-induct*)

case (*empty s s′ x - w*) **with** *empty-wl-succs-discovered* **have** *succs x − dfs-restrict dfs ⊆ discovered s′* **by** *fastforce*
moreover from *empty* **have** *finished s′ = insert x* (*finished s*) **by** *simp*
ultimately show *?case* **using** *empty* **by** (*cases v = x*) *bestsimp+*
qed *auto*

lemma *finished-implies-succs-discoveredI*:
⟦ *dfs-constructable dfs s*; *v ∈ finished s*; *w ∈ succs v*; *w ∉ dfs-restrict dfs* ⟧ ⟹ *w ∈ discovered s*
by (*auto dest*!: *finished-implies-succs-discovered*)

lemma *finished-no-reach-implies-succs-finished*:
dfs-constructable dfs s ⟹ *v ∈ finished s* ⟹ *w ∈ succs v* ⟹ *w ∉ dfs-restrict dfs* ⟹ ¬ *w* →⋆ *v* ⟹ *w ∈ finished s*
proof (*induction rule*: *dfs-constructable-induct*)
case (*empty s s′ x*) **then have** *finished s′ = insert x* (*finished s*) **and** *d′*: *discovered s = discovered s′* **by** *simp-all*
with *empty.prems empty.IH* **show** *?case*
proof (*cases v = x*)
case *True* **with** *empty.prems* **have** *wd*: *w ∈ discovered s′* **using** *finished-implies-succs-discoveredI*[*OF empty*(*3*)] **by** *simp*

{
assume *w ∉ finished s′*
with *wd* **have** *w ∈ set* (*stack s′*) **using** *discovered-not-finished-implies-stack*[*OF empty*(*3*)] **by** *simp*
with *empty True* **have** *w* →⋆ *v* **using** *tl-reachable-stack-hd*[*OF empty*(*2*)] **by** *auto*
with *empty.prems* **have** *False* **by** *simp*
}
thus *?thesis* **by** (*rule ccontr*)
qed *simp*
qed *simp+*

lemma *not-finished-succs-impl-not-finished*:
dfs-constructable dfs s ⟹ *v ∈ discovered s* ⟹ *w ∈ discovered s* ⟹ *δ s v < δ s w* ⟹ *w ∉ finished s* ⟹ *w ∈ succs v* ⟹ *v ∉ finished s*
proof (*induction rule*: *dfs-constructable-induct*)
case (*empty s s′ x xs*) **then have**
d′: *discover s = discover s′* **and**
f′: *finished s′ = insert x* (*finished s*) **and**
s: *stack s = x # xs* **and** *s′*: *stack s′ = xs*
by *simp-all*

from *empty.prems* **have** *w ∈ set* (*stack s′*) **using** *discovered-not-finished-implies-stack*[*OF empty*(*3*)] **by** *simp*
with *s s′* **have** *ws*: *w ∈ set* (*stack s*) **by** *simp*

have *v ≠ x*

```
  proof (rule notI)
    assume v = x
    with ws s empty.prems have δ s w < δ s v using tl-lt-stack-hd-discover[OF
empty(2)] by auto
    with d' have δ s' w < δ s' v by simp
    with empty.prems show False by simp
  qed
  with empty.prems empty.IH f' d' show ?case by simp
next
  case (visit s s' e) hence
    d': discover s' = discover s (e ↦ counter s) and
    s': stack s' = e # stack s and
    f': finish s = finish s'
    by simp-all

  from visit.prems have ws': w ∈ set (stack s') using discovered-not-finished-implies-stack[OF
visit(3)] by simp
  show ?case
  proof (cases w = e)
    case True with visit(8,9) have w ∉ discovered s and w ∉ dfs-restrict dfs by
simp-all
    hence v ∉ finished s using visit.prems finished-implies-succs-discoveredI[OF
visit(2)] by metis
    with f' show ?thesis by simp
  next
    case False with d' have wd: discover s w = discover s' w by simp
    show ?thesis
    proof (cases v = e)
      case True with s' stack-implies-not-finished[OF visit(3)] show ?thesis by
simp
    next
      case False with d' have discover s v = discover s' v by simp
      with wd f' visit.prems visit.IH show ?thesis by fastforce
    qed
  qed
 qed simp+

lemma finished-succs-finish:
  dfs-constructable dfs s ⟹ v ∈ discovered s ⟹ w ∈ finished s ⟹ w ∈ succs v
⟹ δ s w > δ s v ⟹ v ∉ finished s ∨ φ s w < φ s v
proof (induction rule: dfs-constructable-induct)
  case (empty s s' x) hence
    d': discover s = discover s' and
    f': finish s' = finish s (x ↦ counter s)
    by simp-all

  from empty.prems have ne: w ≠ v by auto
  show ?case
  proof (cases w = x)
```

```
   case True with empty have w ∉ finished s using stack-implies-not-finished[OF
empty(2)] by simp
    with empty.prems d' have v ∉ finished s
    using not-finished-succs-impl-not-finished[OF empty(2)] finished-implies-discovered[OF
empty(3)]
     by metis
    with True ne f' show ?thesis by force
  next
    case False with f' empty.prems have fw: finish s w = finish s' w by auto
    show ?thesis
    proof (cases v = x)
     case True with f' have φ s' v = counter s by simp
     moreover from empty.prems fw finish-lt-counter[OF empty(2)] have φ s' w
< counter s by force
     ultimately show ?thesis by simp
    next
     case False with f' have finish s v = finish s' v by simp
     with fw empty.prems d' empty.IH show ?thesis by force
    qed
  qed
next
  case (visit s s' e) hence
    d': discover s' = discover s (e ↦ counter s) and
    s': stack s' = e # stack s and
    f': finish s = finish s'
    by simp-all

  with visit.prems finished-implies-not-stack[OF visit(3)] s' have w ≠ e by simp
  thus ?case
  proof (cases v = e)
    case True with s' show ?thesis using stack-implies-not-finished[OF visit(3)]
by simp
  next
    case False with d' ‹w ≠ e› have discover s v = discover s' v and discover s
w = discover s' w by auto
    with f' visit.prems visit.IH show ?thesis by force
  qed
qed simp+

lemma finished-no-reach-nondiscovered:
  assumes constr: dfs-constructable dfs s
  and noreach: ∀ x ∈ finished s. ∀ y ∈ set(stack s). ¬ x →\dfs-restrict dfs+ y
  and nondisc: z ∉ discovered s
  shows ∀ x ∈ finished s. ¬ x →\dfs-restrict dfs+ z
proof (rule ccontr)
  assume ¬ (∀ x∈finished s. ¬ x →\dfs-restrict dfs+ z)
  hence ∃ x ∈ finished s. x →\dfs-restrict dfs+ z by simp
  then obtain x where x →\dfs-restrict dfs+ z x ∈ finished s by blast
  thus False using nondisc
```

```
  proof induction
    case (base y) with finished-implies-succs-discoveredI constr show False by
blast
  next
    case (trans y z) hence y-disc: y ∈ discovered s by blast
    have y ∈ finished s
    proof (rule ccontr)
      assume y ∉ finished s
       with y-disc discovered-not-finished-implies-stack[OF constr] have y ∈ set
(stack s) by simp
      with trans noreach show False by simp
    qed
     with finished-implies-succs-discoveredI[OF constr] trans have z ∈ finished s
by blast
    with trans finished-subset-discovered[OF constr] show False by auto
  qed
qed

lemma epath-generate-sws [case-names step finished, consumes 2]:
  assumes epath v p w and not-R: set (ewalk-verts v p) ∩ dfs-restrict dfs = {}
  obtains
    (step) s where s ∈ dfs-constr-from dfs v and
                   stack s = rev (ewalk-verts v p) and
                   discovered s = set (ewalk-verts v p) and
                   wl s ≠ [] hd (wl s) = succs w and
                   dfs-cond dfs (state s)
  | (finished) s where s ∈ dfs-constr-from dfs v ¬ dfs-cond dfs (state s)
proof −
  from assms have ewalk v p w and distinct (ewalk-verts v p) unfolding epath-def
by simp-all
  thus ?thesis using that ‹ewalk v p w› not-R
  proof (induct rule: ewalk-induct-rev)
    case (Base v) thus ?case
    proof (cases dfs-cond dfs (state (dfs-constr-start dfs v)))
      case True with Base(1,2,6) Base(3)[of dfs-constr-start dfs v] show ?thesis
by simp
    next
      case False with Base(1,6) Base(4)[of dfs-constr-start dfs v] show ?thesis
by simp
    qed
  next
    case (Snoc x y e es)

    from Snoc(1,2,7) have everts: ewalk-verts v (es@[e]) = (ewalk-verts v es)@[y]
using ewalk-verts-join-l[of v es [e]] by (simp add: ewalk-verts-def)
     with Snoc(4,8) have d-everts: distinct (ewalk-verts v es) and y-not-R: y ∉
dfs-restrict dfs and es-not-R: set (ewalk-verts v es) ∩ dfs-restrict dfs = {} by auto

    from ewalk-join-inner-ends[OF Snoc(7)] Snoc(2) have last (ewalk-verts v es)
```

```
= x by (simp add: ewalk-verts-def)
    with ewalk-join-ewalk-l[OF Snoc(7)] have ewalk: ewalk v es x by simp

    {
      fix s
     assume *: s ∈ dfs-constr-from dfs v stack s = rev (ewalk-verts v es) discovered
s = set (ewalk-verts v es) wl s ≠ [] hd (wl s) = succs x dfs-cond dfs (state s)
      moreover with Snoc have y ∈ succs x hd (wl s) ≠ {} unfolding succs-def
by auto
      moreover from * ewalk-verts-non-Nil have stack s ≠ [] by simp
      then obtain a as where stack s = a#as by (metis list.exhaust)
      moreover
     let ?s' = s(| state := dfs-action dfs (state s) s y, stack := y#a#as, wl := succs
y#(succs x − {y})#tl (wl s), discover := (discover s)(y ↦ counter s), counter :=
Suc (counter s)|)
      from Snoc(4) everts have y ∉ set (ewalk-verts v es) by auto
      with * have y ∉ discovered s by simp
      ultimately have dfs-next dfs s ?s'
        unfolding dfs-next-def dfs-cond-compl-def
        using dfs-step-simps(3)[of s a as dfs] y-not-R
          apply (simp add: dfs-sws.defs)
          apply (intro exI[where x=y])
          apply fastforce
        done

      with * have ?s' ∈ dfs-constr-from dfs v using dfs-constr-from.step by simp
      with Snoc(6) have thesis
      proof (cases dfs-cond dfs (state ?s'))
       case True with * everts ‹stack s = a#as› have stack ?s' = rev (ewalk-verts
v (es@[e])) discovered ?s' = set (ewalk-verts v (es@[e])) wl ?s' ≠ [] hd (wl ?s') =
succs y
         by simp-all
        with True ‹?s' ∈ dfs-constr-from dfs v› show ?thesis using Snoc(5) by
metis
      qed simp
    }
    with Snoc(3)[OF d-everts - - ewalk es-not-R] Snoc(6) show ?case by metis
  qed
qed

lemma reach-generate-sws [case-names step finished, consumes 1]:
  assumes v →\dfs-restrict dfs+ w
  obtains
    s where s ∈ dfs-constr-from dfs v and
          stack s ≠ [] hd (stack s) = w and
          wl s ≠ [] hd (wl s) = succs w and
          dfs-cond dfs (state s)
  | s where s ∈ dfs-constr-from dfs v ¬ dfs-cond dfs (state s)
proof −
```

from *assms* **obtain** *p* **where** *ewalk v p w set* (*ewalk-verts v p*) ∩ *dfs-restrict dfs* = {} **using** *restr-reachable1-def* **by** *blast*
with *epath-ewalk-to-epath epath-verts-sub-ewalk-verts* **have** *epath v* (*ewalk-to-epath p*) *w set* (*ewalk-verts v* (*ewalk-to-epath p*)) ∩ *dfs-restrict dfs* = {} **by** *blast+*
thus *?thesis*
proof (*cases rule*: *epath-generate-sws*[**where** *dfs*=*dfs*])
case (*step s*)
from ‹*epath v* (*ewalk-to-epath p*) *w*› **have** *ewalk v* (*ewalk-to-epath p*) *w* **unfolding** *epath-def* **by** *simp*
hence *ewlast v* (*ewalk-to-epath p*) = *w* **using** *ewalk-conv* **by** *simp*
hence *hd* (*rev* (*ewalk-verts v* (*ewalk-to-epath p*))) = *w* **using** *hd-rev*[*OF ewalk-verts-non-Nil*] **by** *simp*
with *step* **have** *hd* (*stack s*) = *w* **by** *simp*
with *that*(*1*) *step* **show** *?thesis* **by** *simp*
qed (*metis that*(*2*))
qed

Now introduce a notion of a finished search. This enables us to proof properties about the result of a search.

definition *dfs-finished* :: ('*S*, '*n*, '*X*) *dfs-algorithm-scheme* ⇒ ('*S*, '*n*) *dfs-sws* ⇒ *bool* **where**
dfs-finished dfs s ≡ *dfs-constructable dfs s* ∧ ¬(∃ *s'*. *dfs-next dfs s s'*)

lemma *dfs-finishedE*:
⟦*dfs-finished dfs s*; ⟦ *dfs-constructable dfs s*; ¬(∃ *s'*. *dfs-next dfs s s'*) ⟧ ⟹ *P* ⟧ ⟹ *P*
unfolding *dfs-finished-def*
by *metis*

lemma *dfs-finishedI*:
dfs-constructable dfs s ⟹ ¬(∃ *s'*. *dfs-next dfs s s'*) ⟹ *dfs-finished dfs s*
unfolding *dfs-finished-def*
by *metis*

lemma *dfs-finished-stackI*:
dfs-constructable dfs s ⟹ *stack s* = [] ⟹ *dfs-finished dfs s*
unfolding *dfs-finished-def dfs-next-def dfs-cond-compl-def*
by *simp*

lemma *dfs-finished-wlI*:
dfs-constructable dfs s ⟹ *wl s* = [] ⟹ *dfs-finished dfs s*
using *dfs-finished-stackI length-wl-eq-stack*
by *fastforce*

lemma *dfs-finished-constructable*[*intro*]:
dfs-finished dfs s ⟹ *dfs-constructable dfs s*
unfolding *dfs-finished-def*
by *simp*

lemma *not-dfs-cond-implies-finished*:
dfs-constructable dfs s $\Longrightarrow$ $\neg$ *dfs-cond dfs* (*state s*) $\Longrightarrow$ *dfs-finished dfs s*
unfolding *dfs-finished-def dfs-next-def dfs-cond-compl-def*
by *simp*

definition *dfs-completed dfs s* $\equiv$ *dfs-finished dfs s* $\wedge$ *dfs-cond dfs* (*state s*)

lemma *dfs-completed-finished*:
dfs-completed dfs s $\Longrightarrow$ *dfs-finished dfs s*
unfolding *dfs-completed-def*
by *simp*

lemma *finished-is-completed-or-not-cond*:
dfs-finished dfs s $\Longrightarrow$ *dfs-completed dfs s* $\vee$ $\neg$ *dfs-cond dfs* (*state s*)
unfolding *dfs-completed-def*
by *simp*

lemma *dfs-completed-constructable*[*dest*,*simp*]:
dfs-completed dfs s $\Longrightarrow$ *dfs-constructable dfs s*
by (*metis dfs-completed-finished dfs-finished-constructable*)

lemma *dfs-completedE*:
$[\![$ *dfs-completed dfs s*; $[\![$ *dfs-finished dfs s*; *dfs-cond dfs* (*state s*) $]\!] \Longrightarrow P \,]\!] \Longrightarrow P$
unfolding *dfs-completed-def* **by** *metis*

lemma *dfs-completed-empty*:
dfs-completed dfs s $\Longrightarrow$ *stack s* $= []$ $\vee$ *wl s* $= []$
unfolding *dfs-finished-def dfs-next-def dfs-cond-compl-def dfs-completed-def*
using *dfs-step-exists*
by *auto*

lemma *dfs-completed-empty-stack*:
dfs-completed dfs s $\Longrightarrow$ *stack s* $= []$
using *length-wl-eq-stack dfs-completed-empty*
by *force*

lemma *dfs-completed-empty-wl*:
dfs-completed dfs s $\Longrightarrow$ *wl s* $= []$
using *length-wl-eq-stack dfs-completed-empty*
by *force*

lemma *dfs-completed-stackI*:
dfs-constructable dfs s $\Longrightarrow$ *stack s* $= []$ $\Longrightarrow$ *dfs-cond dfs* (*state s*) $\Longrightarrow$ *dfs-completed dfs s*
unfolding *dfs-completed-def*
using *dfs-finished-stackI*
by *simp*

lemma *dfs-completed-wlI*:

dfs-constructable dfs s $\Longrightarrow$ *wl s* = [] $\Longrightarrow$ *dfs-cond dfs* (*state s*) $\Longrightarrow$ *dfs-completed dfs s*
using *dfs-completed-stackI length-wl-eq-stack*
by *fastforce*

lemma *dfs-completed-revE* [*case-names prev*, *induct pred*: *dfs-completed*]:
assumes *finished*: *dfs-completed dfs s*
and *prev*: $\bigwedge s\ r.\ [\![$ *dfs-constructable dfs r*; *dfs-next dfs r s*; *dfs-completed dfs s*; *s* = *r*(| *stack* := [], *wl* := [], *finish* := (*finish r*)(*hd* (*stack r*) $\mapsto$ *counter r*), *counter* := *Suc* (*counter r*), *state* := *dfs-post dfs* (*state r*) *r* (*hd* (*stack r*))|) $]\!] \Longrightarrow$ *P s*
shows *P s*
proof −
from *finished* **have** *sempty*: *stack s* = [] *wl s* = [] **using** *dfs-completed-empty-stack dfs-completed-empty-wl* **by** *auto*

from *finished* **have** *dfs-constructable dfs s* **by** (*rule dfs-completed-constructable*)
thus *?thesis*
proof (*cases rule*: *dfs-constructable.cases*)
case *start* **thus** *?thesis* **using** *sempty* **by** (*simp add*: *dfs-constr-start-def*)
next
case (*step r*)
then have *s* = *r*(| *stack* := [], *wl* := [], *finish* := (*finish r*)(*hd* (*stack r*) $\mapsto$ *counter r*), *counter* := *Suc* (*counter r*), *state* := *dfs-post dfs* (*state r*) *r* (*hd* (*stack r*)) |)
using *sempty*
by (*cases rule*: *dfs-next-cases-elem*) *auto*
with *assms step* **show** *?thesis* **by** *simp*
qed
qed

lemma *dfs-completed-prev-stack*:
assumes *finished*: *dfs-completed dfs s*
and *step*: *dfs-next dfs r s*
and *constr*: *dfs-constructable dfs r*
shows *stack r* = [*start s*]
proof −
note *emp* = *dfs-completed-empty-stack*[*OF finished*]
show *?thesis*
using *step emp assms*
proof (*cases*)
case (*empty x xs*) **with** *emp* **obtain** *e* **where** *stack r* = [*e*] **by** *simp*
with *last-stack-is-start*[*OF constr*] **have** *stack r* = [*start r*] **by** *simp*
with *step dfs-step-preserves-start*[*OF dfs-next-dfs-step*, *OF step*] **show** *?thesis* **by** *auto*
qed *auto*
qed

lemma *completed-start-finished*:

dfs-completed dfs s $\Longrightarrow$ *start s* $\in$ *finished s*
proof (*induct rule*: *dfs-completed-revE*)
case (*prev s r*) **then have** *finish s* = *finish r* (*hd* (*stack r*) $\mapsto$ *counter r*) **by** *simp*
with *prev* **have** *finish s* = *finish r* (*start s* $\mapsto$ *counter r*) **using** *dfs-completed-prev-stack*[*OF prev*(*3*,*2*,*1*)] **by** *force*
thus *?case* **by** *auto*
qed

lemma *completed-start-finish-eq-counter*:
dfs-completed dfs s $\Longrightarrow$ φ *s* (*start s*) = *counter s* − *1*
proof (*induct rule*: *dfs-completed-revE*)
case (*prev s r*) **then have** *finish s* = *finish r* (*hd* (*stack r*) $\mapsto$ *counter r*) **by** *simp*
with *prev* **have** *finish s* = *finish r* (*start s* $\mapsto$ *counter r*) **using** *dfs-completed-prev-stack*[*OF prev*(*3*,*2*,*1*)] **by** *force*
with *prev* **show** *?case* **by** *simp*
qed

lemma *completed-finish-lt-start*:
dfs-completed dfs s $\Longrightarrow$ *v* $\in$ *finished s* $\Longrightarrow$ *v* $\neq$ *start s* $\Longrightarrow$ φ *s v* < φ *s* (*start s*)
proof (*induct rule*: *dfs-completed-revE*)
case (*prev s r*) **then have** *finish s* = *finish r* (*hd* (*stack r*) $\mapsto$ *counter r*) **by** *simp*
with *prev* **have** ∗:*finish s* = *finish r* (*start s* $\mapsto$ *counter r*) **using** *dfs-completed-prev-stack*[*OF prev*(*3*,*2*,*1*)] **by** *force*
hence φ *s* (*start s*) = *counter r* **by** *simp*

moreover
from *prev.prems* ∗ **have** *fv*: *finish r v* = *finish s v* **by** *simp*
with *prev.prems finish-lt-counter*[*OF prev*(*1*)] **have** φ *r v* < *counter r* **by** *force*
with *fv* **have** φ *s v* < *counter r* **by** *simp*

ultimately show *?case* **by** *simp*
qed

lemma *completed-finished-eq-discovered*:
dfs-completed dfs s $\Longrightarrow$ *finished s* = *discovered s*
using *dfs-completed-empty-stack*
using *finished-stack-eq-discovered*[*OF dfs-completed-constructable*]
by (*metis List.set.simps*(*1*) *Un-empty-right*)

lemma *completed-start-finished-restr-reach*:
assumes *compl*: *dfs-completed dfs s*
and *finished*: *v* $\in$ *finished s*
and *ne*: *v* $\neq$ *start s*
shows *start s* $\rightarrow$*dfs-restrict dfs*+ *v*
using *assms*
proof (*induct rule*: *dfs-completed-revE*)

case (*prev s r*) **with** *finished-implies-discovered* **have** $v \in$ *discovered s stack r* $\neq$ [] *discovered s* = *discovered r* **by** (*blast*, *simp-all*)
with *prev* **have** *start r* $\rightarrow\backslash$*dfs-restrict dfs*+ *v* **using** *start-restr-reach-discovered*[*OF prev*(*1*)] **by** *simp*
with *prev* **show** *?case* **by** *simp*
qed

lemma *completed-start-finished-reach*:
assumes *compl*: *dfs-completed dfs s*
and *finished*: $v \in$ *finished s*
shows *start s* $\rightarrow\star$ *v*
proof (*cases start s* = *v*)
case *True* **with** *dfs-start-in-verts compl self-reachable* **show** *?thesis* **by** *auto*
next
case *False* **with** *assms completed-start-finished-restr-reach* **show** *?thesis* **by** *blast*
qed

lemma *completed-reach-implies-finished*:
assumes *compl*: *dfs-completed dfs s*
and *fin*: $v \in$ *finished s*
and *reach*: *v* $\rightarrow\backslash$*dfs-restrict dfs*+ *w*
shows $w \in$ *finished s*
proof −
{
fix *v z*
assume $v \in$ *finished s z* $\in$ *succs v z* $\notin$ *dfs-restrict dfs*
with *finished-implies-succs-discoveredI compl* **have** $z \in$ *discovered s* **by** *blast*
with *compl completed-finished-eq-discovered* **have** $z \in$ *finished s* **by** *blast*
}
with *reach fin* **show** *?thesis* **by** *induction blast*+
qed

lemma *completed-start-reach-finished*:
dfs-completed dfs s $\Longrightarrow$ *start s* $\rightarrow\backslash$*dfs-restrict dfs*+ *v* $\Longrightarrow$ $v \in$ *finished s*
by (*metis completed-start-finished completed-reach-implies-finished*)

lemma *completed-finished-eq-reachable*:
assumes *compl*: *dfs-completed dfs s*
shows *finished s* = {*v*. *v* = *start s* $\vee$ *start s* $\rightarrow\backslash$*dfs-restrict dfs*+ *v*} (**is** ... = *?F*)
proof (*rule set-eqI*[*OF iffI*])
fix *x*
assume $x \in$ *finished s*
hence *x* = *start s* $\vee$ *start s* $\rightarrow\backslash$*dfs-restrict dfs*+ *x*
using *completed-start-finished-restr-reach*[*OF compl*] *completed-start-finished*[*OF compl*]
by (*cases x* = *start s*) *simp-all*
thus $x \in$ *?F* **..**
next
fix *x*

assume $x \in ?F$
hence $x = start\ s \vee start\ s \rightarrow\backslash dfs\text{-}restrict\ dfs+\ x$ **..**
thus $x \in finished\ s$ **using** *completed-start-finished*[*OF compl*] *completed-start-reach-finished*[*OF compl*] **by** *blast*
qed

lemma *no-restrict-finished-eq-reachable*:
assumes *compl*: *dfs-completed dfs s*
and *no-restr*: *dfs-restrict dfs* $\cap$ $V = \{\}$
shows *finished* $s = \{v.\ start\ s \rightarrow\star\ v\}$
proof −
from *compl* **have** *finished* $s = \{v.\ v = start\ s \vee start\ s \rightarrow\backslash dfs\text{-}restrict\ dfs+\ v\}$ **by** (*fact completed-finished-eq-reachable*)
also from *no-restr* **have** ... $= \{v.\ v = start\ s \vee start\ s \rightarrow+\ v\}$ **using** *restr-reachable1-no-verts* **by** *simp*
also from *reachable1-reachable* **have** ... $= \{v.\ v = start\ s \vee start\ s \rightarrow\star\ v\}$ **by** *blast*
also from *self-reachable dfs-start-in-verts compl* **have** ... $= \{v.\ start\ s \rightarrow\star\ v\}$ **by** *force*
finally show *?thesis* **.**
qed

lemma *completed-finished-succs-finish*:
assumes *compl*: *dfs-completed dfs s*
and *finished*: $v \in finished\ s$
and *d-lt*: $\delta\ s\ v < \delta\ s\ w$
and *succs*: $w \in succs\ v$
and $w \notin dfs\text{-}restrict\ dfs$
shows $\varphi\ s\ w < \varphi\ s\ v$
proof −
from *compl* **have** *constr*: *dfs-constructable dfs s* **by** *simp*
moreover from *finished* **have** $v \in discovered\ s$ **using** *finished-implies-discovered*[*OF constr*] **by** *simp*
moreover hence $v \notin dfs\text{-}restrict\ dfs$ **using** *discovered-not-restricted*[*OF constr*] **by** *simp*
with ‹$w \notin dfs\text{-}restrict\ dfs$› *succs succs-restricted* **have** $v \rightarrow\backslash dfs\text{-}restrict\ dfs+\ w$ **by** *simp*
with *completed-reach-implies-finished*[*OF compl finished*] **have** $w \in finished\ s$ **.**

ultimately show *?thesis* **using** *finished-succs-finish succs d-lt finished* **by** *fastforce*
qed

lemma *dfs-finished-cases* [*case-names start step*, *consumes 1*]:
assumes *fin*: *dfs-finished dfs s*
and *start*: $\neg$ *dfs-cond dfs* (*dfs-start dfs* (*start s*)) $\Longrightarrow P$
and *step*: $\bigwedge r.$ ⟦ *dfs-constructable dfs r*; *dfs-next dfs r s* ⟧ $\Longrightarrow P$
shows P
proof (*cases s* = *dfs-constr-start dfs* (*start s*))
case *True* **hence** *stack s* $\neq$ [] **and** *st*: *state s* = *dfs-start dfs* (*start s*) **using**

dfs-constr-start-simps[*of dfs start s*] **by** *simp-all*
hence ¬ *dfs-completed dfs s* **using** *dfs-completed-empty-stack* **by** *blast*
with *fin finished-is-completed-or-not-cond* **have** ¬ *dfs-cond dfs* (*state s*) **by** *blast*
with *st start* **show** *?thesis* **by** *simp*
next
case *False*
from *fin* **have** *dfs-constructable dfs s* **by** *auto*
hence ∃ *r*. *dfs-constructable dfs r* ∧ *dfs-next dfs r s* **using** *False*
by (*cases rule*: *dfs-constructable.cases*) *auto*
with *step* **show** *?thesis* **by** *blast*
qed
end

1.3 Lifting into the while'd

We add an invariant field to the dfs.

record ('*S*, '*n*) *dfs-algorithm-invar* = ('*S*, '*n*) *dfs-algorithm* +
dfs-invar :: ('*S*, '*n*) *dfs-sws* ⇒ *bool* — an invariant that each state built during the search must satisfy. This is dropped in the implementation.

context *finite-digraph*
begin

dfs-invar-compl specifies the invariants that must hold on each working state. If a state does not fulfill parts of the invariant, the behavior of the dfs-algorithm is undefined. As stated later on by *dfs-relation-terminates*, it is guaranteed, that when starting with an invariant-obeying state, all states reached by the algorithms also obey them.

Similar to *dfs-cond-compl* it is split between invariants of the framework and invariants of the implementation. Here the invariant of the framework is, that the state is constructable, as given by *dfs-constructable* defined earlier.

definition *dfs-invar-compl* :: ('*S*, '*n*, '*X*) *dfs-algorithm-invar-scheme* ⇒ '*n* ⇒ ('*S*, '*n*) *dfs-sws* ⇒ *bool*
where
dfs-invar-compl dfs x s ≡ *s* ∈ *dfs-constr-from dfs x* ∧ *dfs-invar dfs s*

lemma *dfs-invar-compl-invar*[*dest*]:
dfs-invar-compl dfs x s ⟹ *dfs-invar dfs s*
unfolding *dfs-invar-compl-def*
by *simp*

lemma *dfs-invar-compl-constr-from*[*dest*]:
dfs-invar-compl dfs x s ⟹ *s* ∈ *dfs-constr-from dfs x*
unfolding *dfs-invar-compl-def*
by *auto*

lemma *dfs-invar-compl-constr*[*dest*]:

dfs-invar-compl dfs x s $\Longrightarrow$ *dfs-constructable dfs s*
by *auto*

lemma *dfs-invar-complE*:
$[\![$ *dfs-invar-compl dfs x s*; $[\![$ *s* $\in$ *dfs-constr-from dfs x*; *dfs-invar dfs s* $]\!] \Longrightarrow P\ ]\!]$
$\Longrightarrow P$
unfolding *dfs-invar-compl-def*
by *metis*

lemma *dfs-invar-complI*:
$[\![$ *s* $\in$ *dfs-constr-from dfs x*; *dfs-invar dfs s* $]\!] \Longrightarrow$ *dfs-invar-compl dfs x s*
unfolding *dfs-invar-compl-def*
by *simp*

And now we initialize the datastructures of the while-framework and show, that we behave as the framework expects us to behave.

definition *dfs-fun* :: $('S,'n,'X)$ *dfs-algorithm-invar-scheme* $\Rightarrow$ $'n \Rightarrow$ $('S,'n)$ *dfs-sws nres* **where**
dfs-fun dfs x $\equiv$
$WHILE_T$(*dfs-invar-compl dfs x*) (*dfs-cond-compl dfs*) (λ *s*. *SPEC* ($\lambda s'$. *dfs-next dfs s s'*)) (*dfs-constr-start dfs x*)

We now state certain predicates the implementations have to obey. These predicates are used as assumptions for specifying properties of the DFS-algorithm.

Start with the need of the implementation to preserve its invariants.

definition *dfs-preserves-invar* :: $('S,\ 'n,\ 'X)$ *dfs-algorithm-invar-scheme* $\Rightarrow$ *bool* **where**
dfs-preserves-invar dfs $\equiv$
($\forall$ *x* $\in$ *V*. *dfs-invar dfs* (*dfs-constr-start dfs x*)) $\wedge$
($\forall$ *x s s'*. *dfs-invar-compl dfs x s* $\wedge$ *dfs-next dfs s s'* $\longrightarrow$
dfs-invar dfs s')

lemma *dfs-preserves-invarI*:
$[\![\bigwedge x\ s\ s'.\ [\![$ *dfs-invar-compl dfs x s*; *dfs-next dfs s s'* $]\!] \Longrightarrow$ *dfs-invar dfs s'*; $\bigwedge x.\ x$ $\in$ *V* $\Longrightarrow$ *dfs-invar dfs* (*dfs-constr-start dfs x*) $]\!]$
$\Longrightarrow$ *dfs-preserves-invar dfs*
unfolding *dfs-preserves-invar-def*
by *blast*

lemma *dfs-preserves-invarD*[*intro*]:
$[\![$ *dfs-preserves-invar dfs*; *dfs-invar-compl dfs x s*; *dfs-next dfs s s'* $]\!] \Longrightarrow$ *dfs-invar dfs s'*
unfolding *dfs-preserves-invar-def*
by *blast*

lemma *dfs-preserves-invarE*[*elim*]:
assumes *dfs-preserves-invar dfs*

and $[\![\bigwedge x\ s\ s'.\ [\![\ \mathit{dfs\text{-}invar\text{-}compl}\ \mathit{dfs}\ x\ s;\ \mathit{dfs\text{-}next}\ \mathit{dfs}\ s\ s'\ ]\!] \Longrightarrow \mathit{dfs\text{-}invar}\ \mathit{dfs}\ s'\ ]\!]$
$\Longrightarrow P$
shows P
using *assms*
unfolding *dfs-preserves-invar-def*
by *blast*

lemma *dfs-preserves-invar-compl*:
assumes *dfs-preserves-invar dfs*
and *dfs-invar-compl dfs x s*
and *dfs-next dfs s s'*
shows *dfs-invar-compl dfs x s'*
using *assms*
by (*metis dfs-constr-from.step dfs-invar-compl-def dfs-preserves-invarD*)

lemma *constr-from-invar-compl*:
$s \in$ *dfs-constr-from dfs x* $\Longrightarrow$ *dfs-preserves-invar dfs* $\Longrightarrow$ *dfs-invar-compl dfs x s*
proof (*induct rule*: *dfs-constr-from.induct*)
case (*step s s'*) **with** *dfs-preserves-invar-compl* **show** *?case* **by** *simp*
next
case *start* **thus** *?case* **by** (*simp add*: *dfs-invar-compl-def dfs-preserves-invar-def*)
qed

lemma *constr-from-invarI*[*intro*!]:
$s \in$ *dfs-constr-from dfs x* $\Longrightarrow$ *dfs-preserves-invar dfs* $\Longrightarrow$ *dfs-invar dfs s*
by (*auto dest*: *constr-from-invar-compl*)

lemma *dfs-constructable-invarI*[*intro*!]:
dfs-constructable dfs s $\Longrightarrow$ *dfs-preserves-invar dfs* $\Longrightarrow$ *dfs-invar dfs s*
by (*auto elim*!: *dfs-constructable-constr-from-start*)

Introduce a measure, that lessens with each search-step. Therefore we define a measure to be a four-tuple, s.t. with each step one element gets smaller and all the previous ones remain equal:

1. undiscovered nodes (lessens in the *visit* phase)
2. length of the search stack (lessens in the *empty* phase)
3. unchecked successors (lessens in the phases *visit* and *remove*, but for *visit* the first item already applies)

abbreviation
dfs-step-measure $\equiv$ *less-than* $<*lex*>$ *less-than* $<*lex*>$ *less-than*

lemma *wf-dfs-step-measure*:
wf dfs-step-measure

by *auto*

definition *ws-to-measure* :: (*'n*, *'X*) *dfs-ws-scheme* ⇒ (*nat* × *nat* × *nat*)
where
ws-to-measure ws ≡ (*card* (*V* − *discovered ws*), *length* (*stack ws*), *card* (*hd* (*wl ws*)))

definition *ws-rel* :: ((*'n*, *'X*) *dfs-ws-scheme* × (*'n*, *'X*) *dfs-ws-scheme*) *set* **where**
ws-rel = {(*ws'*, *ws*). (*ws-to-measure ws'*, *ws-to-measure ws*) ∈ *dfs-step-measure*}

lemma *ws-rel-alt-def*:
ws-rel = *inv-image dfs-step-measure ws-to-measure*
unfolding *ws-rel-def inv-image-def*
by *simp*

theorem *wf-ws-rel*:
wf ws-rel
unfolding *ws-rel-alt-def*
by (*rule wf-inv-image*[*OF wf-dfs-step-measure*])

lemma *ws-rel-intro*:
(*ws-to-measure ws'*, *ws-to-measure ws*) ∈ *dfs-step-measure* ⟹ (*ws'*, *ws*) ∈ *ws-rel*
unfolding *ws-rel-def*
by *simp*

Now show, that this measure indeed applies to the algorithm.

lemma *dfs-next-in-ws-rel*:
assumes *step*: *dfs-next dfs s s'*
and *inv*: *dfs-invar-compl dfs x s*
shows (*s'*, *s*) ∈ *ws-rel*
using *assms*
proof (*cases rule*: *dfs-next-cases-elem*)
case (*empty x xs*)
then have *card* (*V* − *discovered s*) = *card* (*V* − *discovered s'*) **by** *simp*
with *empty* **show** *?thesis* **by** (*simp add*: *ws-to-measure-def ws-rel-intro*)
next
case (*remove e*)
from *inv* **have** *c*: *dfs-constructable dfs s* **by** *auto*
with *wl-subset-succs*[*OF this*, *of 0*] *remove* **have** ∗: *e* ∈ *succs* (*hd* (*stack s'*)) **by** *auto*

from *remove wl-finite*[*OF c*] **have** *F*: *finite* (*hd* (*wl s*)) **using** *hd-in-set* **by** *simp*

from *remove* **have** *card* (*V* − *discovered s*) = *card* (*V* − *discovered s'*) **by** *auto*
moreover from *remove* **have** *length* (*stack s*) = *length* (*stack s'*) **by** *auto*
moreover from *remove* ∗ **have** *hd* (*wl s*) ⊃ *hd* (*wl s'*) **by** *auto*
with *psubset-card-mono*[*OF F*] **have** *card* (*hd* (*wl s*)) > *card* (*hd* (*wl s'*)) **.**

ultimately show *?thesis* **by** (*simp add*: *ws-to-measure-def ws-rel-intro*)

next
case (*restrict e*)
from *inv* **have** *c*: *dfs-constructable dfs s* **by** *auto*
with *wl-subset-succs*[*OF this*, *of 0*] *restrict* **have** ∗: $e \in succs\ (hd\ (stack\ s'))$ **by** *auto*

from *restrict wl-finite*[*OF c*] **have** *F*: *finite* (*hd* (*wl s*)) **using** *hd-in-set* **by** *simp*

from *restrict* **have** *card* $(V - discovered\ s) = card\ (V - discovered\ s')$ **by** *auto*
moreover from *restrict* **have** *length* (*stack s*) = *length* (*stack s*′) **by** *auto*
moreover from *restrict* ∗ **have** *hd* (*wl s*) ⊃ *hd* (*wl s*′) **by** *auto*
with *psubset-card-mono*[*OF F*] **have** *card* (*hd* (*wl s*)) > *card* (*hd* (*wl s*′)) **.**

ultimately show *?thesis* **by** (*simp add*: *ws-to-measure-def ws-rel-intro*)
next
case *visit* **hence** *dsub*: *discovered s* ⊂ *discovered s*′ **by** *auto*

from *finite-V* **have** *finite* $(V - discovered\ s)$ **..**

moreover
from *step inv* **have** *dfs-constructable dfs s*′ **by** (*metis dfs-constructable.step dfs-invar-compl-constr*)
with *visit* **have** *discovered s*′ ⊆ *V* **using** *discovered-subset-verts*[*of dfs s*′] **by** (*simp-all add*: *dfs-sws.defs*)

with *dsub* **have** $V - discovered\ s' \subset V - discovered\ s$ **by** *blast*

ultimately have $card\ (V - discovered\ s) > card\ (V - discovered\ s')$ **by** (*rule psubset-card-mono*)
then show *?thesis* **by** (*simp add*: *ws-to-measure-def ws-rel-intro*)
qed

lemma *exists-finished*:
assumes *pre*: *x* ∈ *V x* ∉ *dfs-restrict dfs*
and *inv*: *dfs-preserves-invar dfs*
shows ∃ *s* ∈ *dfs-constr-from dfs x*. *dfs-finished dfs s*
proof −
from *assms* **have** *dfs-constr-start dfs x* ∈ *dfs-constr-from dfs x* **by** *simp*
with *wfE-min*[*OF wf-ws-rel*] **obtain** *z* **where** *z*: *z* ∈ *dfs-constr-from dfs x* $\bigwedge y.$ (*y*,*z*) ∈ *ws-rel* ⟹ *y* ∉ *dfs-constr-from dfs x* **by** *blast*
moreover with *inv* **have** *dfs-invar-compl dfs x z* **using** *constr-from-invar-compl* **by** *blast*
ultimately have ¬ (∃ *y*. *dfs-next dfs z y*) **by** (*blast intro*: *dfs-constr-from.step dfs-next-in-ws-rel*)
with *z* **show** *?thesis* **by** (*blast intro*: *dfs-finishedI*)
qed

theorem *dfs-fun-correct*:
assumes *dfs-preserves-invar dfs*

and $x \in V\ x \notin$ *dfs-restrict dfs*
shows *dfs-fun dfs x* $\leq$ *SPEC* (λ *s*. *s* $\in$ *dfs-constr-from dfs x* $\wedge$ *dfs-finished dfs s*)
unfolding *dfs-fun-def*
proof (*intro refine-vcg*)
show *wf ws-rel* **by** (*fact wf-ws-rel*)
next
from *assms* **show** *dfs-invar-compl dfs x* (*dfs-constr-start dfs x*) **unfolding** *dfs-invar-compl-def dfs-preserves-invar-def* **by** (*auto intro*!: *dfs-constructable.start*)
next
from *assms* **show** $\bigwedge s\ s'$. $[\![$*dfs-cond-compl dfs s*; *dfs-invar-compl dfs x s*; *dfs-next dfs s s'*$]\!] \Longrightarrow$ *dfs-invar-compl dfs x s'* $\wedge$ (*s'*, *s*) $\in$ *ws-rel*
by (*blast intro*: *dfs-preserves-invar-compl dfs-next-in-ws-rel*)
next
show $\bigwedge s$. $[\![$*dfs-invar-compl dfs x s*; $\neg$ *dfs-cond-compl dfs s*$]\!] \Longrightarrow$ *s* $\in$ *dfs-constr-from dfs x* $\wedge$ *dfs-finished dfs s* **by** (*metis dfs-finishedI dfs-invar-complE dfs-invar-compl-constr dfs-nextE*)
qed

lemma *dfs-fun-nofail*[*refine-pw-simps*]:
assumes *dfs-preserves-invar dfs*
and $x \in V\ x \notin$ *dfs-restrict dfs*
shows *nofail* (*dfs-fun dfs x*)
using *dfs-fun-correct*[*OF assms*]
by (*rule pwD*) *simp*

lemma *dfs-fun-finished*:
assumes *dfs-preserves-invar dfs*
and $x \in V\ x \notin$ *dfs-restrict dfs*
and *inres* (*dfs-fun dfs x*) *s*
shows *dfs-finished dfs s*
proof −
note *dfs-fun-correct*[*OF assms*(*1*−*3*)]
with *assms*(*4*) **have** *inres* (*SPEC* (λ*s*. *s* $\in$ *dfs-constr-from dfs x* $\wedge$ *dfs-finished dfs s*)) *s* **using** *pwD2* **by** *best*
thus *?thesis* **by** *simp*
qed

lemma *dfs-fun-constructable*:
assumes *dfs-preserves-invar dfs*
and $x \in V\ x \notin$ *dfs-restrict dfs*
and *inres* (*dfs-fun dfs x*) *s*
shows *s* $\in$ *dfs-constr-from dfs x*
proof −
note *dfs-fun-correct*[*OF assms*(*1*−*3*)]
with *assms*(*4*) **have** *inres* (*SPEC* (λ*s*. *s* $\in$ *dfs-constr-from dfs x* $\wedge$ *dfs-finished dfs s*)) *s* **using** *pwD2* **by** *best*
thus *?thesis* **by** *simp*
qed

lemma *dfs-step-refine* [*refine-transfer*]:
assumes *dfs-cond-compl dfs s*
shows *RETURN* (*SOME s′. s′ ∈ dfs-step dfs s*) ≤ *SPEC* (λ *s′. dfs-cond-compl dfs s* ∧ *s′ ∈ dfs-step dfs s*)
using *dfs-cond-compl-step-exists*[*OF assms*] *assms*
by (*auto intro*: *someI2-ex*)

schematic-lemma *dfs-fun-code*:
RETURN ?f ≤ *dfs-fun dfs x*
unfolding *dfs-fun-def dfs-next-def*
by (*refine-transfer while.transfer-WHILEIT-esc-cond*)

lemma *dfs-fun-nonempty*:
∃ *s. inres* (*dfs-fun dfs x*) *s*
proof (*cases dfs-fun dfs x*)
case (*RES X*) **with** *dfs-fun-code*[*of dfs x*] **have** *X* ≠ {} **by** *auto*
with *RES* **show** *?thesis* **by** *auto*
qed *simp*

lemma *dfs-fun-pred*:
assumes *x*: *x* ∈ *V x* ∉ *dfs-restrict dfs*
and *pres-inv*: *dfs-preserves-invar dfs*
and *one-d*: ⋀*s*. ⟦ *s* ∈ *dfs-constr-from dfs x*; ¬ *dfs-cond-compl dfs s* ⟧ ⟹ *P s* ⟹ *Q s*
and *snd-d*: ⋀*s*. ⟦ *s* ∈ *dfs-constr-from dfs x*; *dfs-completed dfs s* ⟧ ⟹ *Q s* ⟹ *P s*
and *thrd-d*: ⋀*s*. ⟦ *s* ∈ *dfs-constr-from dfs x*; ¬ *dfs-cond dfs* (*state s*) ⟧ ⟹ *P s*
shows *dfs-fun dfs x* ≤ *SPEC* (λ*s*. *P s* ⟷ *Q s*)
unfolding *dfs-fun-def*
apply (*intro refine-vcg*)
apply (*fact wf-ws-rel*)
apply (*simp add*: *dfs-invar-compl-def x pres-inv*[*unfolded dfs-preserves-invar-def*])
apply (*simp add*: *pres-inv dfs-next-in-ws-rel dfs-preserves-invar-compl*)
proof −
fix *s*
assume *dfs-invar-compl dfs x s* **and** *ncond*: ¬ *dfs-cond-compl dfs s*
hence *constr*: *s* ∈ *dfs-constr-from dfs x* **and**
cond: *stack s* = [] ∨ *wl s* = [] ∨ ¬ *dfs-cond dfs* (*state s*)
unfolding *dfs-cond-compl-def dfs-invar-compl-def* **by** *simp-all*
show *P s* ⟷ *Q s*
proof
assume *P s*
with *one-d constr ncond* **show** *Q s* **by** *metis*
next
assume *Q s*
with *constr cond* **show** *P s*
proof (*cases dfs-cond dfs* (*state s*))

case *True* **with** *cond* **have** *stack s* = [] ∨ *wl s* = [] **by** *simp*
with *length-wl-eq-stack constr* **have** *stack s* = [] **by** *force*
with *True* **have** *dfs-completed dfs s* **using** *dfs-completed-stackI constr* **by** *force*
with ‹*Q s*› **show** *?thesis* **using** *snd-d constr* **by** *metis*
qed (*metis thrd-d*)
qed
qed

end

2 Simple DFS

Introduce a simple dfs algorithm, that ignores all post and pre functions and just returns the filled discover and finish sets.

definition *simple-dfs* :: (*unit*, *'n*) *dfs-algorithm-invar* **where**
simple-dfs = (| *dfs-cond* = λ-. *True*,
dfs-action = λ- - - . (),
dfs-post = λ- - -. (),
dfs-remove = λ- - -. (),
dfs-start = λ-. (),
dfs-restrict = {},
dfs-invar = λ-. *True* |)

lemma *simple-dfs-simps* [*simp*]:
dfs-cond simple-dfs S
dfs-action simple-dfs S s x = ()
dfs-post simple-dfs S s x = ()
dfs-remove simple-dfs S s x = ()
dfs-start simple-dfs x = ()
dfs-invar simple-dfs s
dfs-restrict simple-dfs = {}
by (*simp-all add*: *simple-dfs-def*)

lemma (**in** *finite-digraph*) *simple-dfs-preserves-invar*:
dfs-preserves-invar simple-dfs
by (*rule dfs-preserves-invarI*) *simp-all*
end

theory *Tree-DFS*
imports *Main DFS*
begin

type-synonym (*'n*) *tree* = (*'n* × *'n*) *set*

definition *add-edge* :: *'n tree* ⇒ (*'n tree*, *'n*) *dfs-sws* ⇒ *'n* ⇒ *'n tree* **where**
add-edge e sws n = *insert* (*hd* (*stack sws*), *n*) *e*

definition *tdfs* :: (*'n tree*, *'n*) *dfs-algorithm-invar* **where**
tdfs = (| *dfs-cond* = λ-. *True*, *dfs-action* = *add-edge*, *dfs-post* = λ *s* - -. *s*, *dfs-remove* = λ *s* - -. *s*, *dfs-start* = λ-. {}, *dfs-restrict* = {}, *dfs-invar* = λ-. *True* |)

definition *edges* :: (*'n tree*, *'n*, *'X*) *dfs-sws-scheme* ⇒ *'n tree*
where *edges s* = *state s*

declare *edges-def*[*simp*]

context *finite-digraph*
begin

lemma *tdfs-simps*[*simp*]:
dfs-cond tdfs S
dfs-action tdfs = *add-edge*
dfs-post tdfs S s x = *S*
dfs-remove tdfs S s x = *S*
dfs-start tdfs x = {}
dfs-restrict tdfs = {}
dfs-invar tdfs s
unfolding *tdfs-def*
by *simp-all*

lemma *dfs-next-edges-subset*:
dfs-next tdfs s s' ⟹ *edges s* ⊆ *edges s'*
by (*cases rule*: *dfs-next-cases-elem*) (*auto simp add*: *dfs-sws.defs add-edge-def*)

lemma *no-self-edges*:
dfs-constructable tdfs s ⟹ (*v*, *w*) ∈ *edges s* ⟹ *v* ≠ *w*
proof (*induction rule*: *dfs-constructable-induct*)
case (*visit s s' t x xs*) **hence**
s': *stack s'* = *t*#*x*#*xs* **and**
e': *edges s'* = *insert* (*x*, *t*) (*edges s*)
by (*simp-all add*: *add-edge-def*)

from *s'* **have** *x* ≠ *t* **using** *stack-distinct*[*OF visit*(*3*)] **by** *auto*
thus *?case*
proof (*cases* (*v*,*w*) = (*x*, *t*))
case *False* **with** *e' visit.prems* **have** (*v*,*w*) ∈ *edges s* **by** *auto*
with *visit.IH* **show** *?thesis* **.**
qed *simp*
qed *simp+*

lemma *stack-tl-subset-edges*:
dfs-constructable tdfs s ⟹ *stack s* ≠ [] ⟹ *set* (*tl*(*stack s*)) ⊆ *Field* (*edges s*)
by (*induct rule*: *dfs-constructable-induct*) (*auto simp add*: *add-edge-def tl-subset*)

lemma *stack-hd-in-edges*:
dfs-constructable tdfs s $\Longrightarrow$ *edges s* $\neq$ {} $\Longrightarrow$ *stack s* $\neq$ [] $\Longrightarrow$ *hd* (*stack s*) $\in$ *Field* (*edges s*)
proof (*induct rule*: *dfs-constructable-induct*)
case (*empty s s'*)
hence *set* (*stack s'*) $\subseteq$ *Field* (*edges s'*) **using** *stack-tl-subset-edges*[*OF empty*(*2*)]
by *simp*
with *empty.prems* **show** *?case* **by** (*cases stack s'*) *simp-all*
qed (*simp add*: *add-edge-def*)+

lemma *stack-subset-edges*:
assumes *constr*: *dfs-constructable tdfs s*
and *ne*: *edges s* $\neq$ {}
shows *set* (*stack s*) $\subseteq$ *Field* (*edges s*)
proof (*cases stack s*)
case (*Cons x xs*) **then have** *set xs* $\subseteq$ *Field* (*edges s*) **using** *stack-tl-subset-edges*[*OF constr*] **by** *simp*
moreover from *Cons* **have** *x* $\in$ *Field* (*edges s*) **using** *stack-hd-in-edges*[*OF assms*] **by** *simp*
ultimately show *?thesis* **using** *Cons* **by** *simp*
qed *simp*

lemma *stack-gt-1-implies-edges*:
assumes *dfs-constructable tdfs s*
and *length* (*stack s*) > *1*
shows *edges s* $\neq$ {}
using *stack-tl-subset-edges*[*OF assms*(*1*)] *assms*(*2*)
by (*cases stack s*) *auto*

lemma *edges-stack-is-discovered*:
assumes *dfs-constructable tdfs s*
and *stack s* $\neq$ []
shows *Field* (*edges s*) $\cup$ *set* (*stack s*) = *discovered s*
using *assms*
proof *induction*
case (*empty s s'*) **hence** *stack s'* = *tl* (*stack s*) **by** *simp*
with ‹*stack s'* $\neq$ []› **have** *length* (*stack s*) > *1* **by** (*smt length-0-conv length-tl*)
with *empty* **have** *set* (*stack s*) $\subseteq$ *Field* (*edges s*) **using** *stack-subset-edges stack-gt-1-implies-edges* **by** *metis*
with *empty* **have** *Field* (*edges s'*) = *discovered s'* **by** *auto*
thus *?case* **using** *stack-subset-discovered*[*OF dfs-constructable.step*[*OF empty*(*1*,*2*)]]
by *auto*
next
case (*visit s s' t x*) **hence**
e': *edges s'* = *insert* (*x*, *t*) (*edges s*) **and**
s': *stack s'* = *t* # *stack s*
by (*simp-all add*: *add-edge-def*)

hence *Field* (*edges s'*) = {*x*, *t*} $\cup$ *Field* (*edges s*) **by** *auto*

with s' **have** *Field* (*edges* s') ∪ *set*(*stack* s') = *Field* (*edges* s) ∪ {x, t} ∪ *set* (*stack* s) **by** *auto*
also with *visit* **have** ... = *insert* t (*Field* (*edges* s) ∪ *set* (*stack* s)) **by** *auto*
also with *visit* **have** ... = *discovered* s' **by** *simp*
finally show *?case* **.**
qed *simp+*

lemma *completed-edges-eq-discovered*:
assumes *dfs-completed tdfs s*
and *edges* $s \neq \{\}$
shows *Field* (*edges* s) = *discovered* s
using *assms*
proof *induct*
case (*prev* s r) **then have** *ne*: *edges* $r \neq \{\}$ **by** *simp*

from *prev* **have** *stack-r*: *stack* r = [*start* s] **using** *dfs-completed-prev-stack* **by** *blast*
with *prev ne* **have** *start* $s \in$ *Field* (*edges* r) **using** *stack-hd-in-edges*[*of* r] **by** *force*
with *edges-stack-is-discovered*[*of* r, *OF prev*(*1*)] *stack-r* **have** *Field* (*edges* r) = *discovered* r **by** *force*
with *prev* **show** *?case* **by** *simp*
qed

lemma *edges-subset-discovered*:
assumes *dfs-constructable tdfs s*
shows *Field* (*edges* s) ⊆ *discovered* s
using *edges-stack-is-discovered*[*OF assms*(*1*)]
proof (*cases stack* s)
case *Cons* **then show** *?thesis* **using** *edges-stack-is-discovered*[*OF assms*(*1*)] **by** *auto*
next
case *Nil* **with** *assms* **have** *dfs-completed tdfs s* **using** *dfs-completed-stackI tdfs-simps*(*1*) **by** *force*
from *completed-edges-eq-discovered*[*OF this*] **show** *?thesis* **by** (*cases edges* s = {}) *simp-all*
qed

lemma *etrancl-is-discovered*:
assumes *constr*: *dfs-constructable tdfs s*
and *etrancl*: (v,w) ∈ (*edges* s)$^+$
shows $v \in$ *discovered* s **and** $w \in$ *discovered* s
proof −
from *edges-subset-discovered*[*OF constr*] **have** (*edges* s)$^+$ ⊆ *discovered* s × *discovered* s **using** *trancl-subset-Field2* **by** *auto*
with *etrancl* **show** $v \in$ *discovered* s $w \in$ *discovered* s **by** *auto*
qed

lemma *edges-subset-E*:
dfs-constructable tdfs s $\Longrightarrow$ *edges s* $\subseteq$ *E*
proof (*induction rule*: *dfs-constructable-induct*)
case (*visit s s' t x*) **hence** *e'*: *edges s'* = *insert* (*x*,*t*) (*edges s*) **by** (*simp add*: *add-edge-def*)
moreover from *wl-subset-succs*[*OF visit*(*2*), *of 0*] *visit* **have** *t* $\in$ *succs x* **by** *auto*
ultimately have (*x*, *t*) $\in$ *E* **unfolding** *succs-def* **by** *simp*
with *e' visit.IH* **show** *?case* **by** *simp*
qed *simp+*

lemma *edge-is-succs*:
assumes *dfs-constructable tdfs s*
and (*v*, *w*) $\in$ *edges s*
shows *w* $\in$ *succs v*
proof −
from *assms* **have** (*v*, *w*) $\in$ *E* **using** *edges-subset-E* **by** *auto*
thus *?thesis* **by** (*auto simp add*: *succs-def*)
qed

lemma *discover-edge*:
dfs-constructable tdfs s $\Longrightarrow$ (*v*,*w*) $\in$ *edges s* $\Longrightarrow$ δ *s v* < δ *s w*
proof (*induction rule*: *dfs-constructable-induct*)
case (*visit s s' t x*) **hence**
e': *edges s'* = *insert* (*x*, *t*) (*edges s*) **and**
d': *discover s'* = (*discover s*)(*t* $\mapsto$ *counter s*)
by (*simp-all add*: *add-edge-def*)

show *?case*
proof (*cases* (*v*,*w*) $\in$ *edges s*)
case *True*
from *visit*(*9*) **have** *t* $\notin$ *Field* (*edges s*) **using** *visit*(*2*) *edges-subset-discovered* **by** *auto*
with *True* **have** *v* $\neq$ *t* **and** *w* $\neq$ *t* **unfolding** *Field-def* **by** *auto*
with *d' visit.IH True* **show** *?thesis* **by** *auto*
next
case *False*
with *visit e'* **have** (*v*,*w*) = (*x*, *t*) **by** *auto*

moreover from *visit* **have** *t* $\neq$ *x* **using** *stack-distinct*[*OF visit*(*3*)] **by** *auto*

moreover
from *visit* **have** *x* $\in$ *discovered s* **using** *stack-subset-discovered*[*OF visit*(*2*)] **by** *auto*
hence δ *s x* < *counter s* **using** *discover-lt-counter*[*OF visit*(*2*)] **by** *simp*

ultimately show *?thesis* **using** *d'* **by** *simp*
qed
qed *simp+*

lemma *discover-etrancl*:
assumes *constr*: *dfs-constructable tdfs s*
and *reach*: $(v,w) \in (edges\ s)^+$
shows $\delta\ s\ v < \delta\ s\ w$
using *reach discover-edge*[*OF constr*]
by *induct force+*

lemma *no-cycle*:
dfs-constructable tdfs s $\Longrightarrow$ $(v,w) \in (edges\ s)^+ \Longrightarrow v \neq w$
by (*blast dest*: *discover-etrancl*)

lemma *stack-contains-edges-all*:
dfs-constructable tdfs s $\Longrightarrow \forall\ n < length\ (stack\ s) - 1.\ (stack\ s\ !\ Suc\ n, stack\ s\ !\ n) \in edges\ s$
proof (*induction rule*: *dfs-constructable-induct*)
case (*empty s s' x xs*) **then show** *?case* **by** (*induct xs*) *auto*
next
case (*visit s s' t x*) **hence**
e': *edges s'* = *insert* (*x*, *t*) (*edges s*) **and**
s': *stack s'* = *t* # *stack s*
by (*simp-all add*: *add-edge-def*)

with *visit* **have** *t* = *stack s'* ! *0* **and** *x* = *stack s'* ! *Suc 0* **using** *nth-Cons-0 hd-conv-nth* **by** *simp-all*
with *e'* **have** (*stack s'* ! *Suc 0*, *stack s'* ! *0*) ∈ *edges s'* **by** *simp*

moreover
from *e' s' visit.IH* **have** $\bigwedge n.\ n < (length\ (stack\ s')) - 1 \Longrightarrow n > 0 \Longrightarrow (stack\ s'\ !\ Suc\ n, stack\ s'\ !\ n) \in edges\ s'$
by (*smt gr0-implies-Suc insertI2 list.size*(*4*) *nth-Cons-Suc*)

ultimately show *?case* **by** *smt*
qed *simp+*

lemma *stack-contains-edges*:
dfs-constructable tdfs s $\Longrightarrow n < length\ (stack\ s) - 1 \Longrightarrow (stack\ s\ !\ Suc\ n, stack\ s\ !\ n) \in edges\ s$
by (*metis stack-contains-edges-all*)

lemma *nth-stack-hd-in-etrancl*:
dfs-constructable tdfs s $\Longrightarrow n < length\ (stack\ s) \Longrightarrow n > 0 \Longrightarrow (stack\ s\ !\ n, stack\ s\ !\ 0) \in (edges\ s)^+$
by (*metis nth-step-trancl stack-contains-edges*)

lemma *hd-stack-no-cycle*:
assumes *constr*: *dfs-constructable tdfs s*
and *edge*: (*hd* (*stack s*), *w*) ∈ (*edges s*)$^+$
and *ne*: *stack s* ≠ []

shows $w \notin set\ (stack\ s)$
proof (*rule notI*)
assume *A*: $w \in set\ (stack\ s)$
then obtain n **where** n: $n < length\ (stack\ s)$ $stack\ s\ !\ n = w$ **unfolding** *in-set-conv-nth* **by** *auto*

from *A edge no-cycle*[*OF constr*] **have** $w \neq hd\ (stack\ s)$ **by** *auto*
with n **have** $n > 0$ **by** (*smt hd-conv-nth ne*)
with n *nth-stack-hd-in-etrancl*[*OF constr*] **have** $(w,\ hd(stack\ s)) \in (edges\ s)^+$ **using** *hd-conv-nth*[*OF ne*] **by** *auto*
with *edge* **have** $(w,w) \in (edges\ s)^+$ **by** *auto*
with *no-cycle*[*OF constr*] **show** *False* **by** *auto*
qed

lemma *no-edge-to-stack*:
$dfs\text{-}constructable\ tdfs\ s \Longrightarrow (v,\ w) \in edges\ s \Longrightarrow v \notin set\ (stack\ s) \Longrightarrow w \notin set\ (stack\ s)$
proof (*induction rule*: *dfs-constructable-induct*)
case (*empty s s'*) **hence** *edge*: $(v,w) \in edges\ s$ **by** *simp*
from *empty* **have** *ne*: $stack\ s \neq []$ **and** *tl*: $stack\ s' = tl\ (stack\ s)$ **by** *simp-all*
hence $w \notin set\ (stack\ s)$
proof (*cases* $v = hd\ (stack\ s)$)
case *True* **thus** *?thesis* **using** *hd-stack-no-cycle*[*OF empty*(*2*) - *ne*] *edge* **by** *auto*
next
case *False* **with** *empty.prems ne tl* **have** $v \notin set\ (stack\ s)$ **by** (*fastforce intro*: *list.exhaust*)
with *edge empty.IH* **show** *?thesis* **by** *simp*
qed
with *tl ne* **show** *?case* **by** (*fastforce intro*: *list.exhaust*)
next
case (*visit s s' t x*) **hence** $t \notin discovered\ s$ **by** *simp*
hence *no-edge*: $t \notin Field\ (edges\ s)$ **using** *edges-subset-discovered*[*OF visit*(*2*)] **by** *auto*

from *visit* **have** $v \notin set\ (stack\ s)$ **by** *simp*
moreover with *visit* **have** $(v,w) \in edges\ s$ **by** (*auto simp add*: *add-edge-def*)
ultimately have $w \notin set\ (stack\ s)$ **using** *visit.IH* **by** *simp*
with *visit* **show** *?case*
proof (*cases* $w = t$)
case *True* **with** ‹$(v,w) \in edges\ s$› **have** $t \in Field\ (edges\ s)$ **unfolding** *Field-def* **by** *auto*
with *no-edge* **show** *?thesis* **by** *contradiction*
qed *simp*
qed *simp+*

lemma *no-edge-to-stack-trancl*:
assumes *constr*: *dfs-constructable tdfs s*
and $(v,w) \in (edges\ s)^+$

and $v \notin set\ (stack\ s)$
shows $w \notin set\ (stack\ s)$
using *assms*(*2*,*3*) *no-edge-to-stack*[*OF constr*]
by (*induct rule*: *trancl-induct*) *simp-all*

lemma *finished-etrancl-finished*:
assumes *constr*: *dfs-constructable tdfs s*
and *edge*: $(v,w) \in (edges\ s)^+$
and *finished*: $v \in finished\ s$
shows $w \in finished\ s$
proof −
from *constr finished* **have** $v \notin set\ (stack\ s)$ **by** (*rule finished-implies-not-stack*)
with *no-edge-to-stack-trancl*[*OF constr edge*] **have** $w \notin set\ (stack\ s)$ **.**

moreover
from *edge* **have** $w \in discovered\ s$ **using** *etrancl-is-discovered*[*OF constr*] **by** *simp*
ultimately show *?thesis* **using** *discovered-non-stack-implies-finished*[*OF constr*]
by *auto*
qed

lemma *finish-edge*:
assumes *constr*: *dfs-constructable tdfs s*
and *edge*: $(v,w) \in edges\ s$
and *fin*: $v \in finished\ s$
shows $\varphi\ s\ v > \varphi\ s\ w$
proof −
from *assms* **have** $w \in finished\ s$ **using** *finished-etrancl-finished*[*OF constr*] **by** *force*
moreover from *fin* **have** $v \in discovered\ s$ **using** *finished-implies-discovered*[*OF constr*] **by** *simp*
moreover from *discover-edge constr edge* **have** $\delta\ s\ v < \delta\ s\ w$ **.**
moreover from *edge* **have** $w \in succs\ v$ **using** *edge-is-succs*[*OF constr*] **by** *simp*
ultimately show *?thesis* **using** *finished-succs-finish assms* **by** *blast*
qed

lemma *finish-etrancl*:
assumes *constr*: *dfs-constructable tdfs s*
and *reach*: $(v,w) \in (edges\ s)^+$
and *fin*: $v \in finished\ s$
shows $\varphi\ s\ v > \varphi\ s\ w$
using *reach finish-edge*[*OF constr*] *fin finished-etrancl-finished*[*OF constr*]
by *induct fastforce*+

lemma *completed-finish-etrancl*:
assumes *fin*: *dfs-completed tdfs s*
and *edge*: $(v,w) \in (edges\ s)^+$
shows $\varphi\ s\ v > \varphi\ s\ w$
proof −
from *fin* **have** $Field\ (edges\ s) \subseteq finished\ s$ **using** *edges-subset-discovered completed-finished-eq-discovered*

```
by fastforce
  with edge have v ∈ finished s using trancl-subset-Field2 by auto
  with fin edge show ?thesis using finish-etrancl by auto
qed


lemma discover-finish-implies-in-etrancl:
  assumes dfs-constructable tdfs s
  and v ∈ discovered s and w ∈ discovered s
  and δ s v < δ s w and v ∉ finished s ∨ (w ∈ finished s ∧ φ s w < φ s v)
  shows (v,w) ∈ (edges s)⁺
  using assms
proof induction
  case (empty s s' x) hence
    s': discover s = discover s' and
    f': finish s' = finish s (x ↦ counter s) and
    e': (edges s')⁺ = (edges s)⁺
    by simp-all

  show ?case
  proof (cases v = x)
    case True with empty f' have v ∉ finished s and finish s' w = finish s w
using stack-implies-not-finished[OF empty(2)] by auto
    with empty.prems empty.IH s' e' show ?thesis by simp
  next
    case False  with f' have fv: finish s v = finish s' v by simp
    show ?thesis
    proof (cases w ∈ finished s')
      case False with empty.IH s' e' empty.prems fv show ?thesis by fastforce
    next
      case True note wf' = this
      thus ?thesis
      proof (cases v ∈ finished s')
        case True with empty(12) wf' have *: φ s' w < φ s' v by simp
        have w ≠ x
        proof (rule notI)
          assume w = x
          with f' have φ s' w = counter s by simp

          moreover
          from True fv have v ∈ finished s by auto
          hence φ s v < counter s using finish-lt-counter[OF empty(2)] by simp
          with fv have φ s' v < counter s by simp
          ultimately show False using * by simp
        qed
        with * fv f' wf' have w ∈ finished s ∧ φ s w < φ s v by simp
        with empty.prems s' e' empty.IH show ?thesis by simp
      next
        case False with empty.prems empty.IH s' f' e' show ?thesis by simp
```

```
    qed
   qed
  qed
next
  case (visit s s' e x)
  hence d': discover s' = discover s (e ↦ counter s)
    and s': stack s' = e # stack s
    and f': finish s' = finish s
    and ne: v ≠ w
    and e': edges s' = insert (x, e) (edges s)
    by (auto simp add: add-edge-def)

  show ?case
  proof (cases v = e)
    case True with d' ne have δ s' v = counter s discover s w = discover s' w
by auto

    moreover
    then have w ∈ discovered s using visit.prems by auto
    hence δ s w < counter s using discover-lt-counter[OF visit(2)] by simp

    ultimately have False using visit.prems by simp
    thus ?thesis ..
  next
    case False with d' have dv: discover s v = discover s' v by simp
    show ?thesis
    proof (cases w = e)
      case True with s' have w ∉ finished s' using stack-implies-not-finished[OF
visit(3)] by simp
     with visit.prems have v ∈ set (stack s') using discovered-not-finished-implies-stack[OF
visit(3)] by simp
      then obtain n where n: n < length (stack s') stack s' ! n = v unfolding
in-set-conv-nth by auto
      with False s' have n > 0 by (smt nth-Cons')
     with True e' s' n show ?thesis using nth-stack-hd-in-etrancl[OF visit(3), of
n] by simp
    next
     case False with d' dv f' visit.prems visit.IH have (v, w) ∈ (Tree-DFS.edges
s)+ by fastforce
      with e' trancl-insert show ?thesis by force
    qed
  qed
qed simp+

lemma not-in-etrancl-discover-finish:
  assumes constr: dfs-constructable tdfs s
  and ne: v ≠ w
  and not-in-etrancl: (v,w) ∉ (edges s)+
  shows v ∉ discovered s ∨ w ∉ discovered s ∨ w ∉ finished s ∨ δ s v > δ s w ∨
```

(*v* ∈ *finished s* ∧ *φ s v* < *φ s w*)
proof (*cases v* ∈ *V* ∧ *w* ∈ *V*)
case *False* **hence** *w* ∉ *discovered s* ∨ *v* ∉ *discovered s* **using** *discovered-subset-verts*[*OF constr*] **by** *auto*
thus *?thesis* **by** *simp*
next
case *True* **hence** *verts*: *v* ∈ *V w* ∈ *V* **by** *simp-all*
from *not-in-etrancl* **show** *?thesis*
apply (*rule contrapos-np*)
apply (*rule discover-finish-implies-in-etrancl*[*OF constr*, *of v w*])
apply (*insert discover-neq-discover*[*OF constr verts ne*] *finish-neq-finish*[*OF constr verts ne*])
by *force+*
qed

lemma *both-not-in-etrancl-discover-finish*:
assumes *constr*: *dfs-constructable tdfs s*
and *ne*: *v* ≠ *w*
and *not-in-etrancl*: (*v*,*w*) ∉ (*edges s*)$^+$ (*w*,*v*) ∉ (*edges s*)$^+$
and *fin*: *v* ∈ *finished s w* ∈ *finished s*
shows *δ s w* > *φ s v* ∨ *φ s w* < *δ s v*
proof −
from *constr fin* **have** *disc*: *v* ∈ *discovered s w* ∈ *discovered s* **using** *finished-implies-discovered* **by** *metis+*
with *constr ne not-in-etrancl not-in-etrancl-discover-finish fin* **have** *δ s v* > *δ s w* ∨ *φ s v* < *φ s w δ s w* > *δ s v* ∨ *φ s w* < *φ s v* **by** *metis+*
thus *?thesis* **using** *correct-order constr fin* **by** *smt*
qed

end

declare *edges-def*[*simp del*]

2.1 The DFS search tree

We now are interested in the search-tree that is built by a DFS. For the sake of easier proofs, we assume the graph is non-empty, because this case is trivial and uninteresting.

locale *dfs-run* = *dfs*: *finite-digraph V E* + *G*: *finite-digraph V*$_G$ *E*$_G$
for *V V*$_G$ :: *'n set* **and** *E E*$_G$ :: (*'n* × *'n*) *set* +
fixes *s* :: (*'n tree*, *'n*) *dfs-sws*
assumes *E-edges*: *E* = *edges s*
and *E-ne*: *E* ≠ {}
and *V-discovered*: *V* = *discovered s*
and *completed*: *dfs-completed tdfs s*
begin

abbreviation *G-fd-to-graph* (*𝒢*$_G$) **where**
G-fd-to-graph ≡ *G.fd-to-graph*

abbreviation *G-fd-reachable* (- $\rightarrow G\star$ - [*100*,*100*] *40*) **where**
  G-fd-reachable $\equiv$ *G.fd-reachable*
abbreviation *G-fd-reachable1* (- $\rightarrow G+$ - [*100*,*100*] *40*) **where**
  G-fd-reachable1 $\equiv$ *G.fd-reachable1*

abbreviation *dfs-fd-to-graph* ($\mathcal{G}_d$) **where**
  dfs-fd-to-graph $\equiv$ *dfs.fd-to-graph*
abbreviation *dfs-fd-reachable* (- $\rightarrow d\star$ - [*100*,*100*] *40*) **where**
  dfs-fd-reachable $\equiv$ *dfs.fd-reachable*
abbreviation *dfs-fd-reachable1* (- $\rightarrow d+$ - [*100*,*100*] *40*) **where**
  dfs-fd-reachable1 $\equiv$ *dfs.fd-reachable1*

lemma *dfs-reach-etrancl*:
  $v \rightarrow d+ \; w \longleftrightarrow (v,w) \in (edges\ s)^+$
by (*metis E-edges dfs.reachable1-trancl*)

lemma *constructable*:
  dfs-constructable tdfs s
by (*rule dfs-completed-constructable*[*OF completed*])

lemma *dfs-edges-sub*:
  $E \subseteq E_G$
by (*metis constructable edges-subset-E E-edges*)

lemma *dfs-vertices-sub*:
  $V \subseteq V_G$
by (*metis constructable discovered-subset-verts V-discovered*)

lemma *verts-eq-edges*:
  $V = Field\ E$
by (*metis completed V-discovered completed-edges-eq-discovered E-ne E-edges*)

lemma *s-discovered*[*simp*, *intro*]:
  $x \in V \Longrightarrow x \in discovered\ s$
using *V-discovered*
by *simp*

lemma *s-finished*[*simp*,*intro*]:
  $x \in V \Longrightarrow x \in finished\ s$
using *completed-finished-eq-discovered*[*OF completed*] *s-discovered*
by *auto*

lemma *vert-reach-implies-vert*:
  assumes $x \in V$
  and *reach*: $x \rightarrow G+ \; y$
  shows $y \in V$
proof −
  have *fin*: $V = finished\ s$ **using** *V-discovered completed-finished-eq-discovered*[*OF completed*] **by** *simp*

with ‹$x \in V$› **have** $x \in$ *finished s* **by** *simp*
with *reach completed-reach-implies-finished*[*OF completed*] **have** $y \in$ *finished s*
by *auto*
with *fin* **show** *?thesis* **by** *simp*
qed

lemma *vert-ewalk-ewalk-verts*:
assumes *v*: $x \in V$
and *walk*: *ewalk x p y*
shows *set* (*ewalk-verts x p*) $\subseteq V$
using *walk v walk*
proof *induction*
case *Base* **thus** *?case* **by** *simp*
next
case (*Cons w1 w2 e es*) **with** *ewalk-Cons-iff* **have** *ewalk w2 es y* **by** *simp*
moreover from *edge-implies-reach1*[*of e*] *vert-reach-implies-vert Cons* **have** *w2*
$\in V$ **by** *simp*
ultimately have *set* (*ewalk-verts w2 es*) $\subseteq V$ **using** *Cons.IH* **by** *simp*
with *ewalk-verts-conv*[*OF* ‹*ewalk w2 es y*›] **have** *snd* ‘ *set es* $\subseteq V$ **by** *simp*
with *Cons ewalk-verts-conv*[*OF Cons*(*5*)] ‹*w2* $\in V$› **show** *?case* **by** *simp*
qed

lemma *vert-ewalk-edge-verts*:
assumes *v*: $x \in V$
and *walk*: *ewalk x p y*
and *edge*: $e \in$ *set p*
shows {*fst e*, *snd e*} $\subseteq V$
proof −
from *v walk* **have** *set* (*ewalk-verts x p*) $\subseteq V$ **using** *vert-ewalk-ewalk-verts* **by**
simp
with *G.ewalk-edge-in-ewalk-verts*[*OF walk edge*] **show** *?thesis* **by** *auto*
qed
end

sublocale *dfs-run* $\subseteq$ *finite-subgraph* V_G V E_G E
using *dfs-edges-sub verts-eq-edges* **by** *unfold-locales*

context *dfs-run*
begin

no-notation *disc* (δ)
no-notation *fin* (φ)

abbreviation $\delta \equiv$ *disc s*
abbreviation $\varphi \equiv$ *fin s*

lemma *times-relation*:
$v \in V \Longrightarrow \delta\ v < \varphi\ v$
using *finish-gt-discover*[*OF constructable*] *s-finished*

by *force*

lemma *no-self-reach*:
$v \rightarrow d+ \ w \Longrightarrow v \neq w$
using *dfs-reach-etrancl no-cycle*[*OF constructable*]
by *simp*

lemma *correct-order*:
$v \in V \Longrightarrow w \in V \Longrightarrow \delta \ v < \delta \ w \Longrightarrow \ \varphi \ v < \delta \ w \vee \varphi \ v > \varphi \ w$
by (*metis G.correct-order*[*OF constructable*] *s-finished*)

lemma *reachE* [*case-names reach-vw reach-wv no-reach*, *consumes 2*]:
assumes *verts*: $v \in V \ w \in V$
obtains
(*reach-vw*) $v \rightarrow d\star \ w$
| (*reach-wv*) $w \rightarrow d\star \ v$
| (*no-reach*) $\neg \ v \rightarrow d\star \ w \ \neg \ w \rightarrow d\star \ v$
proof (*cases* $v \rightarrow d\star \ w$)
case *True* **with** *reach-vw* **show** *?thesis* **.**
next
case *False* **note** *no-vw* = *this*
show *?thesis*
proof (*cases* $w \rightarrow d\star \ v$)
case *True* **with** *reach-wv* **show** *?thesis* **.**
next
case *False* **with** *no-vw no-reach* **show** *?thesis* **by** *simp*
qed
qed

lemma *discover-neq-finish*:
$v \in V \Longrightarrow w \in V \Longrightarrow \delta \ v \neq \varphi \ w$
proof −
assume *v*: $v \in V$ **and** *w*: $w \in V$
with *dfs-vertices-sub* **have** $v \in V_G \ w \in V_G$ **by** *auto*
hence *discover s v* ≠ *finish s w* **using** *discover-neq-finish*[*OF constructable*]
s-discovered[*OF v*] **by** *simp*
with *s-discovered*[*OF v*] *s-finished*[*OF w*] **show** $\delta \ v \neq \varphi \ w$ **by** *auto*
qed

lemma *discover-neq-discover*:
$v \in V \Longrightarrow w \in V \Longrightarrow v \neq w \Longrightarrow \delta \ v \neq \delta \ w$
proof −
assume *v*: $v \in V$ **and** *w*: $w \in V$ **and** *ne*: $v \neq w$
with *dfs-vertices-sub* **have** $v \in V_G \ w \in V_G$ **by** *auto*
hence *discover s v* ≠ *discover s w* **using** *discover-neq-discover*[*OF constructable*]
s-discovered[*OF v*] *ne* **by** *simp*
with *s-discovered*[*OF v*] *s-discovered*[*OF w*] **show** $\delta \ v \neq \delta \ w$ **by** *auto*
qed

lemma *finish-neq-finish*:
$v \in V \Longrightarrow w \in V \Longrightarrow v \neq w \Longrightarrow \varphi\ v \neq \varphi\ w$
proof −
assume *v*: $v \in V$ **and** *w*: $w \in V$ **and** *ne*: $v \neq w$
with *dfs-vertices-sub* **have** $v \in V_G\ w \in V_G$ **by** *auto*
hence *finish s v* ≠ *finish s w* **using** *finish-neq-finish*[*OF constructable*] *s-finished*[*OF v*] *ne* **by** *simp*
with *s-finished*[*OF v*] *s-finished*[*OF w*] **show** $\varphi\ v \neq \varphi\ w$ **by** *auto*
qed

lemma *reach-discover*:
$v \rightarrow d+\ w \Longrightarrow \delta\ v < \delta\ w$
by (*metis discover-etrancl*[*OF constructable*] *dfs-reach-etrancl*)

lemma *reach-finish*:
$v \rightarrow d+\ w \Longrightarrow \varphi\ v > \varphi\ w$
by (*metis completed-finish-etrancl*[*OF completed*] *dfs-reach-etrancl*)

lemma *no-reach-obtain-split*:
assumes *nr*: $\neg\ x \rightarrow d\star\ y$
and *vert*: $x \in V$
and *ewalk*: *G.ewalk x p y*
obtains *e* **where** $e \in set\ p$ **and** $x \rightarrow d\star\ fst\ e$ **and** $\neg\ x \rightarrow d\star\ snd\ e$
proof −
note *vert-ewalk-edge-verts*[*OF vert ewalk*]
with *ewalk nr vert that* **show** *?thesis*
proof (*induction rule*: *G.ewalk-induct-rev*)
case *Base* **hence** *False* **using** *dfs.self-reachable* **by** *simp*
thus *?case* **..**
next
case (*Snoc v1 v2 e es*)
from *Snoc.prems*(*4*)[*of e*] *Snoc.hyps* **have** $v1 \in V\ v2 \in V$ **by** *simp-all*
show *?case*
proof (*cases v1* $\rightarrow d\star$ *v2*)
case *True* **with** *Snoc* **have** $\neg\ x \rightarrow d\star\ v1$ **using** *dfs.reachable-trans* **by** *metis*
with *Snoc.IH Snoc.prems* **show** *?thesis* **by** *auto*
next
case *False* **with** *Snoc* **show** *?thesis*
proof (*cases x* $\rightarrow d\star$ *v1*)
case *True* **with** *Snoc Snoc.prems*(*3*)[*of e*] **show** *?thesis* **by** *auto*
qed *auto*
qed
qed
qed

lemma *discover-finish-implies-reach*:
$v \in V \Longrightarrow w \in V \Longrightarrow \delta\ v < \delta\ w \Longrightarrow \varphi\ v > \varphi\ w \Longrightarrow v \rightarrow d+\ w$
by (*metis discover-finish-implies-in-etrancl*[*OF constructable*] *s-discovered s-finished dfs-reach-etrancl*)

lemma *no-reach-discover-finish*:
assumes *verts*: $v \in V\ w \in V$
and *no-reach*: $\neg\ v \rightarrow d\star\ w$
shows $\delta\ v > \delta\ w \vee \varphi\ v < \varphi\ w$
proof −
from *assms* **have** *ne*: $v \neq w$ **using** *dfs.reachable-ewalk*[*of v w*] *dfs.ewalk-empty-iff*[*of v w*] **by** *simp*
from *no-reach* **show** *?thesis*
proof (*rule contrapos-np*)
assume $\neg\ (\delta\ w < \delta\ v \vee \varphi\ v < \varphi\ w)$
with *verts ne* **have** $v \rightarrow d+\ w$ **using** *discover-finish-implies-reach*[*of v w*] *discover-neq-discover finish-neq-finish* **by** *smt*
thus $v \rightarrow d\star\ w$ **by** *blast*
qed
qed

lemma *both-no-reach-discover-finish*:
$v \in V \Longrightarrow w \in V \Longrightarrow \neg\ v \rightarrow d\star\ w \Longrightarrow \neg\ w \rightarrow d\star\ v \Longrightarrow \delta\ w > \varphi\ v \vee \varphi\ w < \delta\ v$
using *no-reach-discover-finish correct-order*
by *smt*

lemma *treeish*:
assumes *reach*: $v \rightarrow d+\ w$
shows $\neg\ w \rightarrow d\star\ v$
proof (*rule notI*)
assume $w \rightarrow d\star\ v$
with *no-self-reach*[*OF reach*] *dfs.reachable-neq-reachable1* **have** $w \rightarrow d+\ v$ **by** *simp*
hence $\varphi\ w > \varphi\ v$ **using** *reach-finish* **by** *simp*
moreover from *reach* **have** $\varphi\ v > \varphi\ w$ **using** *reach-finish* **by** *simp*
ultimately show *False* **by** *simp*
qed

lemma *discover-finish-impl-disj*:
assumes $v \in V\ w \in V$
and $\delta\ v < \delta\ w\ \varphi\ v < \varphi\ w$
shows $\varphi\ v < \delta\ w$
using *assms*
by (*smt correct-order*)

lemma *disjointI*:
assumes *verts*: $v \in V\ w \in V$
and *fini*: $\varphi\ v < \delta\ w$
shows $\delta\ v < \delta\ w$ **and** $\delta\ v < \varphi\ w$ **and** $\varphi\ v < \varphi\ w$
proof −
from *verts* **have** $\delta\ v < \varphi\ v$ **and** $\delta\ w < \varphi\ w$ **by** (*metis times-relation*)+
with *fini* **show** $\delta\ v < \delta\ w$ **and** $\delta\ v < \varphi\ w$ **and** $\varphi\ v < \varphi\ w$ **by** *auto*
qed

We now introduce the well-known notions of white, grey and black nodes.

definition $WHITE\ u\ x \equiv \delta\ x \geq u$
definition $GREY\ u\ x \equiv \delta\ x < u \wedge \varphi\ x \geq u$
definition $BLACK\ u\ x \equiv \varphi\ x < u$

lemma *single-color*:
  assumes $x \in V$
  shows $WHITE\ u\ x \neq (GREY\ u\ x \vee BLACK\ u\ x)$
  and $GREY\ u\ x \neq (WHITE\ u\ x \vee BLACK\ u\ x)$
  and $BLACK\ u\ x \neq (WHITE\ u\ x \vee GREY\ u\ x)$
proof −
  from *assms* **have** $\delta\ x < \varphi\ x$ **using** *finish-gt-discover*[*OF constructable*] *s-finished s-discovered* **by** *force*
  thus $WHITE\ u\ x \neq (GREY\ u\ x \vee BLACK\ u\ x)$
    and $GREY\ u\ x \neq (WHITE\ u\ x \vee BLACK\ u\ x)$
    and $BLACK\ u\ x \neq (WHITE\ u\ x \vee GREY\ u\ x)$
  unfolding *WHITE-def GREY-def BLACK-def* **by** *auto*
qed

lemma *grey-impl-reach*:
  assumes *verts*: $v \in V\ w \in V$
  and *grey*: $GREY\ u\ v\ GREY\ u\ w$
  shows $v \rightarrow d\star\ w \vee w \rightarrow d\star\ v$
using *verts*
proof (*cases rule*: *reachE*)
  case *no-reach* **with** *verts* **have** $\delta\ w > \varphi\ v \vee \delta\ v > \varphi\ w$ **using** *both-no-reach-discover-finish* **by** *simp*
  with *grey* **show** *?thesis* **unfolding** *GREY-def* **by** *auto*
qed *simp-all*

The following two notions for disjunction and inclusion are just for expressing the parenthesis theorem in better-known ways.

definition $DISJ\ v\ w \equiv (\ \varphi\ v < \delta\ w \vee \varphi\ w < \delta\ v) \wedge \neg\ v \rightarrow d\star\ w \wedge \neg\ w \rightarrow d\star\ v$
definition $IN\ v\ w \equiv (\ \delta\ v < \delta\ w \wedge \varphi\ v > \varphi\ w) \wedge v \rightarrow d+\ w$

lemma *reach-eq-IN*:
  $v \rightarrow d+\ w \longleftrightarrow IN\ v\ w$
unfolding *IN-def*
by (*force dest*: *reach-finish reach-discover*)

lemma *not-reach-eq-DISJ*:
  $v \in V \Longrightarrow w \in V \Longrightarrow \neg\ v \rightarrow d\star\ w \wedge \neg\ w \rightarrow d\star\ v \longleftrightarrow DISJ\ v\ w$
unfolding *DISJ-def*
by (*blast dest*: *both-no-reach-discover-finish*)

theorem *parenthesis*:
  assumes *verts*: $v \in V\ w \in V$
  and $v \neq w$
  shows $DISJ\ v\ w \neq (IN\ v\ w \vee IN\ w\ v)$

and *IN v w* ≠ (*DISJ v w* ∨ *IN w v*)
using *assms*
by (*metis not-reach-eq-DISJ reach-eq-IN treeish dfs.reachable-neq-reachable1 dfs.reachable1-reachable*)+

2.1.1 White Path Theorem

definition *p-white-path* **where**
p-white-path p x y ≡ *G.ewalk x p y* ∧ (∀ *v* ∈ *set* (*G.ewalk-verts x p*). *WHITE* (*δ x*) *v*)

definition *white-path* **where**
white-path x y ≡ ∃ *p*. *p-white-path p x y*

lemma *white-path-iff*:
white-path x y ⟷ (∃ *p*. *G.ewalk x p y* ∧ (∀ *v* ∈ *set* (*ewalk-verts x p*). *WHITE* (*δ x*) *v*))
unfolding *p-white-path-def white-path-def*
by (*simp add*: *ewalk-verts-eq*)

lemma *reach-impl-white-path*:
assumes *x* →*d*⋆ *y*
shows *white-path x y*
proof (*cases x* = *y*)
case *True* **with** *assms* **have** *G.ewalk x* [] *y* **using** *G.ewalk-empty-iff dfs-vertices-sub dfs.reach-implies-vert* **by** *auto*
moreover
hence *G.ewalk-verts x* [] = [*x*] **by** *simp*
hence ∀ *v* ∈ *set* (*G.ewalk-verts x* []). *WHITE* (*δ x*) *v* **unfolding** *WHITE-def* **by** *simp*
ultimately show *?thesis* **unfolding** *white-path-iff* **by** *force*
next
case *False* **with** *assms* **have** *x* →*d*+ *y* **by** *auto*
with *reach-discover* **have** *disc*: *δ x* < *δ y* **.**

from *assms* **obtain** *p* **where** *ewalk*: *dfs.ewalk x p y* **using** *dfs.reachable-ewalk* **by** *blast*
hence *dfs.ves p* **unfolding** *dfs.ewalk-def* **by** *simp*

have ⋀*v*. *v* ∈ *set* (*dfs.ewalk-verts x p*) ⟹ *WHITE* (*δ x*) *v*
proof −
fix *v*
assume *v* ∈ *set* (*dfs.ewalk-verts x p*)
with *ewalk* ‹*dfs.ves p*› **have** *x* →*d*⋆ *v*
proof *induction*
case (*Base w*) **thus** *?case* **using** *dfs.self-reachable* **by** *simp*
next
case (*Cons w1 w2 e es*)
show *?case*
proof (*cases v* = *w1*)

case *True*
from *Cons* **have** *w1* = *fst e* **by** *simp*
hence *w1* ∈ *V* **using** *Cons*(*1*) *verts-eq-edges fst-in-Field* **by** *force*
with *True* **show** *?thesis* **by** (*simp add*: *dfs.self-reachable*)
next
case *False* **with** *Cons* **have** *w2* →*d*⋆ *v* **using** *dfs.ewalk-verts-Cons dfs.ves-ConsD* **by** *simp*
moreover
from *Cons.hyps* **have** *w1* →*d*⋆ *w2* **using** *dfs.succs-reachable*[*unfolded dfs.succs-def*] **by** *auto*
ultimately show *?thesis* **using** *dfs.reachable-trans* **by** *metis*
qed
qed
thus *WHITE* (*δ x*) *v* **unfolding** *WHITE-def*
proof (*cases x*=*v*)
assume *x* ≠ *v x* →*d*⋆ *v* **hence** *x* →*d*+ *v* **by** *auto*
with *reach-discover* **show** *δ x* ≤ *δ v* **by** *force*
qed *simp*
qed
then show *?thesis* **unfolding** *white-path-iff ewalk-verts-eq* **by** (*blast intro*: *ewalk2-implies-ewalk ewalk*)
qed

lemma *white-path-impl-reach*:
assumes *wp*: *white-path x y*
and *x* ∈ *V*
shows *x* →*d*⋆ *y*
proof (*rule ccontr*)
assume *no-reach*: ¬ *x* →*d*⋆ *y*

from *wp* **obtain** *p* **where** *p*: *p-white-path p x y* **unfolding** *white-path-def* **by** *blast*
hence *ewalk*: *ewalk x p y* **unfolding** *p-white-path-def* **by** *simp*
with *no-reach* ‹*x* ∈ *V*› **obtain** *e* **where**
e: *e* ∈ *set p x* →*d*⋆ *fst e* ¬ *x* →*d*⋆ *snd e*
using *no-reach-obtain-split* **by** *blast*
then obtain *u v* **where** *u*: *u* = *fst e* **and** *v*: *v* = *snd e* **by** *blast*
with *e* **have** *x-u*: *x* →*d*⋆ *u* **and** *x-v*: ¬ *x* →*d*⋆ *v* **by** *simp-all*
hence *u-ne-v*: *u* ≠ *v* **and** *x-ne-v*: *x* ≠ *v* **using** *dfs.self-reachable* ‹*x* ∈ *V*› **by** *force*+

from *e ewalk* **have** *e* ∈ E_G **unfolding** *ewalk-conv* **by** *auto*
with *u v* **have** *succs*: *v* ∈ *G.succs u* **unfolding** *succs-def* **by** *simp*

from *e u v ewalk* **have** *u* ∈ *V v* ∈ *V* **using** *vert-ewalk-edge-verts*[*OF* ‹*x* ∈ *V*›] **by** *simp-all*
note *verts* = *this* ‹*x* ∈ *V*›

from *x-u* **have** *xu-rel*: *δ x* ≤ *δ u φ x* ≥ *φ u* **using** *reach-discover*[*of x u*]

reach-finish[*of x u*] *dfs.reachable-neq-reachable1* **by** (*case-tac* [!] $x = u$) *fastforce+*

from *e v ewalk* **have** $v \in$ *set* (*G.ewalk-verts x p*) **using** *G.ewalk-edge-in-ewalk-verts*
by *auto*
with *p* **have** *WHITE* (δ *x*) *v* **unfolding** *p-white-path-def* **by** *simp*
with *x-ne-v* **have** *xv-rel*: $\delta\ x < \delta\ v$ **unfolding** *WHITE-def* **using** *discover-neq-discover*
verts **by** *force*
with *correct-order*[*OF - - this*] *verts* **have** *xv-relf*: $\varphi\ x < \delta\ v \vee \varphi\ x > \varphi\ v$ **by**
simp

from *verts* **show** *False*
proof (*cases rule*: *reachE*)
case *reach-vw* **thus** *?thesis* **by** (*metis x-u x-v dfs.reachable-trans*)
next
case *reach-wv* **with** *u-ne-v* **have** $v \rightarrow d+\ u$ **by** *auto*
hence $\delta\ v < \delta\ u$ **and** $\varphi\ v > \varphi\ u$ **using** *reach-discover reach-finish* **by** *simp-all*

from *xv-relf* **show** *?thesis*
proof (*rule disjE*)
assume $\varphi\ x < \delta\ v$ **with** $\langle \delta\ v < \delta\ u \rangle$ *times-relation*[*of u*] *verts* **have** $\varphi\ x <$
$\varphi\ u$ **by** *simp*
with *xu-rel* **show** *False* **by** *simp*
next
assume $\varphi\ x > \varphi\ v$ **with** *xv-rel* **have** $x \rightarrow d+\ v$ **using** *discover-finish-implies-reach*
verts **by** *simp*
with *x-v* **show** *False* **by** *auto*
qed
next
case *no-reach* **with** *both-no-reach-discover-finish verts* **have** $\varphi\ v < \delta\ u \vee \varphi\ u$
$< \delta\ v$ **by** *simp*
thus *?thesis*
proof (*rule disjE*)
assume $\varphi\ v < \delta\ u$ **with** *xu-rel xv-rel times-relation verts* **have** $\delta\ x < \delta\ v$
and $\varphi\ x > \varphi\ v$ **by** *force+*
with *discover-finish-implies-reach verts x-v* **show** *False* **by** *auto*
next
assume $\varphi\ u < \delta\ v$ **with** *times-relation*[*of u*] *verts* **have** $\delta\ u < \delta\ v$ **by** *simp*
moreover
with *G.completed-finished-succs-finish*[*OF completed - - succs*] *s-finished verts*
have $\varphi\ v < \varphi\ u$ **by** *simp*
ultimately have $u \rightarrow\star\ v$ **using** *discover-finish-implies-reach verts* **by** *auto*
with *x-u x-v dfs.reachable-trans* **show** *False* **by** *metis*
qed
qed
qed

theorem *white-path*:
$x \in V \Longrightarrow$ *white-path x y* $\longleftrightarrow$ $x \rightarrow d\star\ y$
by (*metis reach-impl-white-path white-path-impl-reach*)

end
end
theory *Nested-DFS*
imports *DFS*
begin

context *finite-digraph*
begin

fun *check-cycle* :: *'n set* ⇒ *bool* ⇒ (*bool*, *'n*) *dfs-sws* ⇒ *'n* ⇒ *bool* **where**
check-cycle - True - - = True
| *check-cycle sup-st False - e* = (*e* ∈ *sup-st*) — if e is on the parent (=super) stack, we have a cycle

definition *sub-dfs-invar* **where**
sub-dfs-invar sup-st s ≡ ¬ *state s* ⟶ (*finished s* ∩ *sup-st* ⊆ {*start s*} ∧
set (*stack s*) ∩ *sup-st* ⊆ {*start s*} ∧
(*start s* ∈ *sup-st* ⟶ ((∀ *x* ∈ *finished s*. *start s* ∉ *succs x*) ∧
(∀ *n* < *length* (*stack s*). *start s* ∉ *wl s* ! *n* ⟶ *start s* ∉ *succs* (*stack s* ! *n*)))))

lemma *sub-dfs-invarI*:
⟦ ¬ *state s*; *finished s* ∩ *sup-st* ⊆ {*start s*}; *set* (*stack s*) ∩ *sup-st* ⊆ {*start s*}; *start s* ∈ *sup-st* ⟹ ∀ *x* ∈ *finished s*. *start s* ∉ *succs x*; *start s* ∈ *sup-st* ⟹ ∀ *n* < *length* (*stack s*). *start s* ∉ *wl s* ! *n* ⟶ *start s* ∉ *succs* (*stack s* ! *n*) ⟧ ⟹ *sub-dfs-invar sup-st s*
unfolding *sub-dfs-invar-def*
by *blast*

definition *sub-dfs* :: *'n set* ⇒ *'n set* ⇒ (*bool*, *'n*) *dfs-algorithm-invar* **where**
sub-dfs sup-st R = (| *dfs-cond* = *Not*, *dfs-action* = *check-cycle sup-st*, *dfs-post* = λ *S* - -. *S*, *dfs-remove* = λ *S s x*. *x* = *start s* ∧ *x* ∈ *sup-st*, *dfs-start* = λ-. *False*, *dfs-restrict* = *R*, *dfs-invar* = *sub-dfs-invar sup-st* |)

lemma *sub-dfs-simps*[*simp*]:
dfs-cond (*sub-dfs ss R*) *S* ⟷ ¬ *S*
dfs-action (*sub-dfs ss R*) = *check-cycle ss*
dfs-post (*sub-dfs ss R*) *S s x* = *S*
dfs-remove (*sub-dfs ss R*) *S s x* ⟷ *x* = *start s* ∧ *x* ∈ *ss*
dfs-start (*sub-dfs ss R*) *x* = *False*
dfs-invar (*sub-dfs ss R*) = *sub-dfs-invar ss*
dfs-restrict (*sub-dfs ss R*) = *R*
unfolding *sub-dfs-def*
by (*simp-all*)

lemma *sub-dfs-state-not-next*:
dfs-constructable (*sub-dfs ss R*) *s* ⟹ *state s* ⟹ ¬ *dfs-next* (*sub-dfs ss R*) *s s'*

unfolding *dfs-next-def* **by** *auto*

lemma *sub-dfs-preserves-invar*[*intro*!]:
dfs-preserves-invar (*sub-dfs ss R*)
proof (*rule dfs-preserves-invarI*)
fix *x s s'*
assume *step*: *dfs-next* (*sub-dfs ss R*) *s s'* **and** *inv-cl*:*dfs-invar-compl* (*sub-dfs ss R*) *x s*
hence *inv1*: *finished s* ∩ *ss* ⊆ {*start s*} **and**
inv2: *set* (*stack s*) ∩ *ss* ⊆ {*start s*} **and**
inv3: *start s* ∈ *ss* ⟶ (∀ *x* ∈ *finished s*. *start s* ∉ *succs x*) **and**
inv4: *start s* ∈ *ss* ⟶ (∀ *n* < *length* (*stack s*). *start s* ∉ *wl s* ! *n* ⟶ *start s* ∉ *succs* (*stack s* ! *n*))
unfolding *dfs-invar-compl-def* **by** (*auto simp add*: *sub-dfs-invar-def sub-dfs-state-not-next*)
note *invs* = *inv1 inv2 inv3 inv4*

from *step inv-cl* **have** ¬ *state s* **unfolding** *dfs-invar-compl-def* **by** (*auto simp add*: *sub-dfs-state-not-next*)

have *sub-dfs-invar ss s'*
proof (*cases state s'*)
case *True* **thus** *?thesis* **by** (*simp add*: *sub-dfs-invar-def*)
next
case *False* **with** *step invs* **show** *?thesis*
proof (*cases rule*: *dfs-next-cases-elem*)
case (*remove e x xs w ws*) **with** *False* **have** *e-neq-s*: *start s* ∈ *ss* ⟹ *e* ≠ *start s* **by** *fastforce*

from *remove invs False* **show** *?thesis*
proof (*intro sub-dfs-invarI*)
case *goal5*
with *inv4* **have** *inv4'*: ∀ *n* < *length* (*stack s*). *start s* ∉ *wl s* ! *n* ⟶ *start s* ∉ *succs* (*stack s* ! *n*) **by** *force*

from *remove* **have** ∗ : *stack s'* = *x*#*xs wl s'* = (*w* − {*e*}) # *ws* **by** *simp-all*
with *remove*(*1*,*2*) **show** *?case* **using** *inv4'*
proof (*induct rule*: *stack-wl-remove-induct*)
case *0* **with** *e-neq-s goal5* **show** *?case* **by** *auto*
qed (*auto simp add*: *remove*)
qed *simp-all*
next
case (*restrict e x xs w ws*) **with** *inv-cl start-not-restr* **have** *e-not-start*: *e* ≠ *start s* **by** *blast*

from *restrict invs False* **show** *?thesis*
proof (*intro sub-dfs-invarI*)
case *goal5*
with *inv4* **have** *inv4'*: ∀ *n* < *length* (*stack s*). *start s* ∉ *wl s* ! *n* ⟶ *start*

$s \notin succs\ (stack\ s\ !\ n)$ **by** *force*

from *restrict* **have** $* : stack\ s' = x\#xs\ wl\ s' = (w - \{e\})\ \#\ ws$ **by** *simp-all*
with *restrict*(*1*,*2*) **show** *?case* **using** *inv4'*
proof (*induct rule*: *stack-wl-remove-induct*)
case *0* **with** *e-not-start goal5* **show** *?case* **by** *auto*
qed (*auto simp add*: *restrict*)
qed *simp-all*
next
case *empty* **with** *invs* **show** *?thesis* **unfolding** *sub-dfs-invar-def* **by** *auto*
next
case (*visit e x xs w ws*) **with** *inv-cl start-discovered* **have** $e \neq start\ s$ **by** *auto*
from *visit False* ‹$\neg$ *state s*› **have** $e \notin ss$ **by** *simp*

with *False visit invs* **show** *?thesis*
proof (*intro sub-dfs-invarI*)
case *goal5*
with *inv4* **have** *inv4'*: $\forall\ n < length\ (stack\ s).\ start\ s \notin wl\ s\ !\ n \longrightarrow start\ s \notin succs\ (stack\ s\ !\ n)$ **by** *force*

from *visit* **have** $* : stack\ s' = e\#x\#xs\ wl\ s' = succs\ e\#(w - \{e\})\ \#\ ws$ **by** *simp-all*
with *visit*(*1*,*2*) **show** *?case*
using *inv4'*
proof (*induct rule*: *stack-wl-visit-induct*)
case *case-1* **with** ‹$e \neq start\ s$› *visit* **show** *?case* **by** *auto*
qed (*auto simp add*: *visit*)
qed *simp-all*
qed
qed
thus $dfs\text{-}invar\ (sub\text{-}dfs\ ss\ R)\ s'$ **by** *simp*
qed (*simp add*: *sub-dfs-invar-def*)

definition *sub-cyclic* **where** *sub-cyclic ss s* $\equiv$
$stack\ s \neq [] \wedge$
$((hd\ (stack\ s) \in ss \wedge start\ s \neq hd\ (stack\ s)) \vee (start\ s \in succs\ (hd\ (stack\ s)) \wedge start\ s \in ss)) \wedge$
state s
definition *sub-cycle* **where** *sub-cycle ss R x* $\equiv \exists s \in dfs\text{-}constr\text{-}from\ (sub\text{-}dfs\ ss\ R)\ x.\ sub\text{-}cyclic\ ss\ s$

lemma *state-implies-sub-cyclic*:
$dfs\text{-}constructable\ (sub\text{-}dfs\ ss\ R)\ s \Longrightarrow state\ s \Longrightarrow sub\text{-}cyclic\ ss\ s$
proof (*induction rule*: *dfs-constructable-induct*)
case (*visit s s' e x*) **hence** $\neg$ *state s* **using** *sub-dfs-state-not-next* **by** *force*
moreover from *visit* **have** *check-cycle ss* (*state s*) *s e* **by** *simp*
ultimately have $e \in ss$ **by** *simp*
moreover from *visit start-discovered* **have** $e \neq start\ s$ **by** *auto*
ultimately show *?case* **using** *visit* **unfolding** *sub-cyclic-def* **by** *force*

next
case (*remove s s' e x*) **hence** ¬ *state s* **using** *sub-dfs-state-not-next* **by** *force*
with *remove* **have** *e* ∈ *ss e* = *start s* **by** *auto*
moreover from *remove* **have** *e* ∈ *succs* (*hd* (*stack s*)) **using** *wl-subset-succs*[*OF remove*(*2*), *of 0*] **by** *auto*
ultimately show *?case* **unfolding** *sub-cyclic-def* **using** *remove* **by** *simp*
qed (*simp-all add*: *sub-dfs-state-not-next sub-cyclic-def*)

lemma *sub-cycle-not-completed*:
assumes *constr*: *s* ∈ *dfs-constr-from* (*sub-dfs ss R*) *x*
and *sub*: *x* →*R*+ *v v* ∈ *ss*
shows ¬ *dfs-completed* (*sub-dfs ss R*) *s*
proof (*rule notI*)
let *?DFS* = *sub-dfs ss R*
assume *compl*: *dfs-completed ?DFS s*
hence *cond*: *dfs-cond ?DFS* (*state s*) **unfolding** *dfs-completed-def* **by** *simp*
moreover from *constr-from-invarI*[*OF constr*] **have** *inv*: *sub-dfs-invar ss s* **by** *auto*
ultimately have *nr*: *finished s* ∩ *ss* ⊆ {*start s*} *start s* ∈ *ss* ⟹ ∀ *x*∈*finished s*. *start s* ∉ *succs x* **unfolding** *sub-dfs-invar-def* **by** *simp-all*

from *constr* **have** *start*: *start s* = *x* **using** *constr-from-implies-start* **by** *blast*

from *sub* **have** *x-reach*: *x* →*dfs-restrict ?DFS*+ *v* **by** *simp*
with *compl start completed-start-finished completed-reach-implies-finished* **have** *v* ∈ *finished s* **by** *metis*
thus *False*
proof (*cases v* = *x*)
case *True* **with** *x-reach* ‹*v* ∈ *ss*› ‹*v* ∈ *finished s*› **show** *False*
proof *induction*
case (*base y*) **with** *base nr start* **show** *False* **by** *blast*
next
case (*trans y z*) **with** *completed-reach-implies-finished*[*OF compl*] ‹*v* ∈ *finished s*› *True* **have** *y* ∈ *finished s* **by** *blast*
with *trans nr start* **show** *False* **by** *blast*
qed
next
case *False*
from ‹*v* ∈ *finished s*› ‹*v* ∈ *ss* › **have** *v* ∈ *finished s* ∩ *ss* **by** *auto*
with *False start nr* **show** *False* **by** *auto*
qed
qed

lemma *sub-cycle-generates-sub-cyclic*:
assumes *reach*: *x* →*R*+ *v*
and *sub-cycle*: *v* ∈ *ss*
obtains *s* **where** *s* ∈ *dfs-constr-from* (*sub-dfs ss R*) *x sub-cyclic ss s*
proof −
let *?DFS* = *sub-dfs ss R*

from *reach* **have** $x \in V$ **and** $x \notin$ *dfs-restrict ?DFS* **using** *restricted-start* **by** *auto*
with *exists-finished*[*OF - - sub-dfs-preserves-invar*] **obtain** *s* **where** *s*: $s \in$ *dfs-constr-from ?DFS x dfs-finished ?DFS s* **by** *blast*
moreover with *sub-cycle-not-completed assms* **have** $\neg$ *dfs-completed ?DFS s* **by** *simp*
ultimately have $\neg$ *dfs-cond ?DFS* (*state s*) **using** *finished-is-completed-or-not-cond* **by** *blast*
hence *state s* **by** *simp*
with *s state-implies-sub-cyclic that constr-from-implies-start* **show** *?thesis* **by** *blast*
qed

lemma *sub-cycle-get-in-sup-st*:
assumes *sub-cycle ss R x*
obtains *v* **where** $x \rightarrow\backslash R+\ v\ v \in ss$
proof −
from *assms* **obtain** *s* **where** *constr*: $s \in$ *dfs-constr-from* (*sub-dfs ss R*) *x* **and** *sub-cyclic ss s* **unfolding** *sub-cycle-def* **by** *auto*
with *constr-from-implies-start* **have** *sne*: *stack s* $\neq$ [] **and** $*$: $(x \neq hd\ (stack\ s) \wedge hd\ (stack\ s) \in ss) \vee (x \in succs\ (hd\ (stack\ s)) \wedge x \in ss)$ **unfolding** *sub-cyclic-def* **by** *auto*

show *?thesis*
proof (*cases* $x \neq hd\ (stack\ s) \wedge hd\ (stack\ s) \in ss$)
case *True* **with** *constr-from-restr-reach-stack*[*OF constr*] *sne* **have** $x \rightarrow\backslash R+\ hd\ (stack\ s)$ **by** *auto*
with *True that* **show** *?thesis* **by** *blast*
next
case *False* **with** $*$ **have** $**$:$x \in succs\ (hd\ (stack\ s)) \wedge x \in ss$ **by** *simp*
moreover from *constr sne* **have** $x \notin$ *dfs-restrict* (*sub-dfs ss R*) $hd\ (stack\ s) \notin$ *dfs-restrict* (*sub-dfs ss R*)
using *constr-from-implies-start stack-not-restricted start-not-restr hd-in-set*
by *blast+*
ultimately have *reach*: $hd\ (stack\ s) \rightarrow\backslash R+\ x$ **using** *succs-restricted* **by** *simp*
show *?thesis*
proof (*cases* $x = hd\ (stack\ s)$)
case *True* **with** *reach that* $**$ **show** *?thesis* **by** *simp*
next
case *False* **with** *constr-from-restr-reach-stack*[*OF constr*] *sne* **have** $x \rightarrow\backslash R+\ hd\ (stack\ s)$ **by** *auto*
with $**$ *restr-reachable1-trans*[*OF - reach*] *that* **show** *?thesis* **by** *blast*
qed
qed
qed

lemma *sub-cycle-iff-in-sup-st*:
sub-cycle ss R x $\longleftrightarrow$ $(\exists\, v.\ x \rightarrow\backslash R+\ v \wedge v \in ss)$
proof

assume *sub-cycle ss R x* **thus** $\exists\, v.\ x \rightarrow \backslash R+\ v \wedge v \in ss$ **by** (*metis sub-cycle-get-in-sup-st*)
next
assume $\exists\, v.\ x \rightarrow \backslash R+\ v \wedge v \in ss$
then guess *v* **..**
then obtain *s* **where** *sub-cyclic ss s s* $\in$ *dfs-constr-from* (*sub-dfs ss R*) *x* **using** *sub-cycle-generates-sub-cyclic* **by** *blast*
thus *sub-cycle ss R x* **unfolding** *sub-cycle-def* **by** *blast*
qed

lemma *dfs-fun-sub-dfs-correct*:
assumes $x \in V\ x \notin R$
shows *dfs-fun* (*sub-dfs ss R*) *x* $\leq$ *SPEC* ($\lambda s.$ *state s* $\longleftrightarrow$ *sub-cycle ss R x*)
using *assms*
proof (*intro dfs-fun-pred*)
fix *s*
assume *s* $\in$ *dfs-constr-from* (*sub-dfs ss R*) *x state s*
moreover hence *sub-cyclic ss s* **using** *state-implies-sub-cyclic* **by** *auto*
ultimately show *sub-cycle ss R x* **by** (*auto simp*: *sub-cycle-def*)
next
fix *s*
assume *s* $\in$ *dfs-constr-from* (*sub-dfs ss R*) *x*
and *dfs-completed* (*sub-dfs ss R*) *s*
and *sub-cycle ss R x*
hence *False* **by** (*metis sub-cycle-iff-in-sup-st sub-cycle-not-completed*)
thus *state s* **..**
qed (*simp-all add*: *sub-dfs-preserves-invar*)

theorem *sub-dfs-correct*:
assumes $x \in V\ x \notin R$
and *inres* (*dfs-fun* (*sub-dfs ss R*) *x*) *s*
shows *state s* $\longleftrightarrow$ *sub-cycle ss R x*
using *assms dfs-fun-sub-dfs-correct*
by (*bestsimp dest*!: *pwD2*)

theorems *sub-dfs-correct-unfolded* = *sub-dfs-correct*[*unfolded sub-cycle-iff-in-sup-st*]

corollary *sub-dfs-correct-compl*:
assumes $x \in V\ x \notin R$
and *inres* (*dfs-fun* (*sub-dfs ss R*) *x*) *s*
shows *dfs-completed* (*sub-dfs ss R*) *s* $\longleftrightarrow \neg$ *sub-cycle ss R x*
proof −
from *sub-dfs-correct*[*OF assms*] **have** $\neg$ *dfs-cond* (*sub-dfs ss R*) (*state s*) $\longleftrightarrow$ *sub-cycle ss R x* **by** *simp*
moreover from *dfs-fun-correct assms* **have** *dfs-finished* (*sub-dfs ss R*) *s* **by** (*bestsimp dest*!: *pwD2*)
ultimately show *?thesis* **by** (*simp add*: *dfs-completed-def*)
qed

lemmas *sub-dfs-correct-compl-unfolded* = *sub-dfs-correct-compl*[*unfolded sub-cycle-iff-in-sup-st*]

lemma *RETURN-sub-cycle-correct*:
assumes $x \in V\ x \notin R$
shows *RETURN* (*sub-cycle ss R x*) = *do* { $s \leftarrow$ *dfs-fun* (*sub-dfs ss R*) *x*; *RETURN* (*state s*) }
apply (*rule pw-eqI*)
apply (*simp-all add*: *pw-bind-inres pw-bind-nofail dfs-fun-nofail*[*OF sub-dfs-preserves-invar*] *assms*)
apply (*metis assms dfs-fun-nonempty sub-dfs-correct*)
done

lemmas *RETURN-sub-cycle-correct-unfolded* = *RETURN-sub-cycle-correct*[*unfolded sub-cycle-iff-in-sup-st*]

lemma *RETURN-sub-cycle-finished-correct*:
assumes $x \in V\ x \notin R$
shows *RETURN* (*if sub-cycle ss R x then None else Some* {$v.\ v = x \vee x \rightarrow\backslash R+ v$}) =
do { $s \leftarrow$ *dfs-fun* (*sub-dfs ss R*) *x*; *RETURN* (*if state s then None else Some* (*finished s*)) } (**is** *?L* = *?R*)
proof (*rule pw-eqI*)
show *nofail ?L* $\longleftrightarrow$ *nofail ?R* **by** (*simp-all add*: *pw-bind-inres pw-bind-nofail dfs-fun-nofail*[*OF sub-dfs-preserves-invar*] *assms*)
next
let *?DFS* = *sub-dfs ss R*
fix *s*
from ‹$x \notin R$› **have** *restr*: $x \notin$ *dfs-restrict ?DFS* **by** *simp*

show *inres ?L s* $\longleftrightarrow$ *inres ?R s*
proof (*cases sub-cycle ss R x*)
case *True*
from *dfs-fun-nonempty* **obtain** s' **where** s': *inres* (*dfs-fun ?DFS x*) s' **by** *blast*
with s' *True sub-dfs-correct*[*OF assms*] **show** *?thesis* **by** (*auto simp add*: *pw-bind-inres pw-bind-nofail dfs-fun-nofail*[*OF sub-dfs-preserves-invar*] *assms*)
next
case *False*
{
fix s'
assume *inres* (*dfs-fun ?DFS x*) s'
moreover with *dfs-fun-constructable assms restr* **have** $s' \in$ *dfs-constr-from ?DFS x* **by** *blast*
moreover hence *start* $s' = x$ **using** *constr-from-implies-start* **by** *blast*
moreover note *assms False*
ultimately have *dfs-completed ?DFS* s' *finished* $s' = \{v.\ v = x \vee x \rightarrow\backslash R+ v\}$
using *completed-finished-eq-reachable*[**where** *dfs=?DFS*] *sub-dfs-correct-compl* **by** (*simp-all*)
} **note** *inres-R-impl* = *this*

show *?thesis*
proof
assume *inres ?L s* **with** *False* **have** $s = Some\ \{v.\ v = x \vee x \rightarrow\backslash R+\ v\}$ **by** *simp*
moreover from *dfs-fun-nonempty* **obtain** s' **where** s': *inres* (*dfs-fun ?DFS x*) s' **by** *blast*
moreover note *inres-R-impl*[*OF this*]
ultimately show *inres ?R s* **by** (*auto simp add*: *pw-bind-inres pw-bind-nofail dfs-fun-nofail*[*OF sub-dfs-preserves-invar*] *assms dfs-completed-def*)
next
assume *inres*: *inres ?R s*
hence $s = Some\ \{v.\ v = x \vee x \rightarrow\backslash R+\ v\}$ **using** *inres-R-impl*[*unfolded dfs-completed-def*] **by** (*auto simp add*: *pw-bind-inres pw-bind-nofail dfs-fun-nofail*[*OF sub-dfs-preserves-invar*] *assms*)
thus *inres ?L s* **using** *False* **by** *simp*
qed
qed
qed

lemmas *RETURN-sub-cycle-finished-correct-unfolded* = *RETURN-sub-cycle-finished-correct*[*unfolded sub-cycle-iff-in-sup-st*]

end

2.2 Now the grand loop

locale *finite-accepting-digraph* = *finite-digraph V E*
for $V :: 'n\ set$ **and** $E :: ('n \times 'n)\ set$ +
fixes $\mathcal{A} :: 'n\ set$
begin

definition *run-sub-dfs'* :: $'n\ set \Rightarrow 'n\ set \Rightarrow 'n \Rightarrow 'n\ set\ option$ **where**
run-sub-dfs' ss R e ≡ *THE b. RETURN b* ≤ *do* { *s* ← *dfs-fun* (*sub-dfs ss R*) *e*; *RETURN* (*if state s then None else Some* (*finished s*)) }

lemma *run-sub-dfs'-correct*:
assumes $e \in V$ **and** $e \notin R$
shows *run-sub-dfs' ss R e* = *None* ⟷ $(\exists v.\ e \rightarrow\backslash R+\ v \wedge v \in ss)$
unfolding *run-sub-dfs'-def*
apply (*rule theI2*)
apply (*simp-all only*: *RETURN-sub-cycle-finished-correct-unfolded*[*OF assms*, *symmetric*])
apply *simp-all*
done

lemma *run-sub-dfs'-finished*:
assumes *e*: $e \in V$ $e \notin R$
and *run-sub-dfs' ss R e* = *Some f*
shows $f = \{v.\ v = e \vee e \rightarrow\backslash R+\ v\}$

proof −
note *RETURN-sub-cycle-finished-correct-unfolded*[*OF e*, *of ss*, *symmetric*]
also from *assms* **have** *run-sub-dfs′ ss R e* ≠ *None* **by** *simp*
with *run-sub-dfs′-correct*[*OF e*] **have** *RETURN* (*if* ∃ *v*. *e* →*R*+ *v* ∧ *v* ∈ *ss* *then None else Some* {*v*. *v* = *e* ∨ *e* →*R*+ *v*}) = *RETURN* (*Some* {*v*. *v* = *e* ∨ *e* →*R*+ *v*}) **by** *simp*
finally have *run-sub-dfs′ ss R e* = (*Some* {*v*. *v* = *e* ∨ *e* →*R*+ *v*}) **unfolding** *run-sub-dfs′-def* **by** *simp*
with *assms* **show** *?thesis* **by** *simp*
qed

fun *run-sub-dfs* :: ((*bool* × *′n set*, *′n*) *dfs-sws* ⇒ *′n set*) ⇒ *bool* × *′n set* ⇒ (*bool* × *′n set*, *′n*) *dfs-sws* ⇒ *′n* ⇒ *bool* × *′n set* **where**
run-sub-dfs M (*True*, *F*) - - = (*True*, *F*)
| *run-sub-dfs M* (*False*, *F*) *s e* = (*if e* ∈ *A then*
case run-sub-dfs′ (*M s*) *F e of*
None ⇒ (*True*, *F*) | *Some F′* ⇒ (*False*, *F* ∪ *F′*)
else (*False*, *F*))

lemma *run-sub-dfs-not-accpt*:
e ∉ *A* ⟹ *run-sub-dfs M S s e* = *S*
by (*cases* (*M*, *S*, *s*, *e*) *rule*: *run-sub-dfs.cases*) *simp-all*

abbreviation *has-cycle* :: (*bool* × *′n set*, *′n*) *dfs-sws* ⇒ *bool* **where**
has-cycle x ≡ *fst* (*state x*)

lemma *run-sub-dfs-casesE*[*consumes 3*, *case-names same cycle no-cycle*]:
assumes ¬ *fst S*
and *e* ∈ *V e* ∉ *snd S*
and *same*: ⟦ *e* ∉ *A*; *run-sub-dfs M S s e* = *S*⟧ ⟹ *P*
and *cycle*: ⋀*v*. ⟦ *e* ∈ *A*; *run-sub-dfs′* (*M s*) (*snd S*) *e* = *None*; *run-sub-dfs M S s e* = (*True*, *snd S*); *e* →*snd S*+ *v*; *v* ∈ *M s* ⟧ ⟹ *P*
and *no-cycle*: ⋀*F′*. ⟦ *e* ∈ *A*; *run-sub-dfs′* (*M s*) (*snd S*) *e* = *Some F′*; *run-sub-dfs M S s e* = (*False*, *snd S* ∪ *F′*); ∀ *v*. *e* →*snd S*+ *v* ⟶ *v* ∉ *M s*; *F′* = {*v*. *v* = *e* ∨ *e* →*snd S*+ *v*} ⟧ ⟹ *P*
shows *P*
proof (*cases e* ∈ *A*)
case *False* **with** *run-sub-dfs-not-accpt same* **show** *?thesis* **by** *simp*
next
case *True*
obtain *F* **where** *F*: *F* = *snd S* **by** *auto*
then obtain *subS* **where** *subS*: *run-sub-dfs′* (*M s*) *F e* = *subS* **by** *auto*
show *?thesis*
proof (*cases subS*)
case *None* **with** *F subS assms True* **have** *run-sub-dfs M S s e* = (*True*, *F*) **by** (*cases* (*M*, *S*, *s*, *e*) *rule*: *run-sub-dfs.cases*) *simp-all*
moreover from *assms subS None run-sub-dfs′-correct*[*of e F*] *F* **obtain** *v* **where** *e* →*snd S*+ *v v* ∈ *M s* **by** *auto*
ultimately show *?thesis* **using** *cycle F True None subS* **by** *simp*

next
case (*Some F'*) **with** *F subS assms True* **have** *run-sub-dfs M S s e* = (*False*, $F \cup F'$) **by** (*cases* (*M*, *S*, *s*, *e*) *rule*: *run-sub-dfs.cases*) *simp-all*
moreover from *assms subS Some run-sub-dfs'-finished*[*of e F - F'*] *F* **have** $F' = \{v.\ v = e \vee e \rightarrow\backslash F+\ v\}$ **by** *simp*
moreover from *Some* **have** *subS* $\neq$ *None* **by** *simp*
with *assms subS run-sub-dfs'-correct*[*of e F M s*] *F* **have** $\forall\, v.\ e \rightarrow\backslash F+\ v \longrightarrow v \notin M\ s$ **by** *simp*
ultimately show *?thesis* **using** *no-cycle F True Some subS* **by** *simp*
qed
qed

definition *nested-dfs-invar* **where**
nested-dfs-invar s $\equiv$ ($\neg$ *has-cycle s* $\longrightarrow$
$(\forall\, x \in \mathcal{A} \cap$ *finished s*. $\neg\ x \rightarrow+\ x) \wedge$
set (*stack s*) $\cap$ *snd* (*state s*) = $\{\} \wedge$
$(\forall\, x.\ x \notin$ *discovered s* $\longrightarrow x \notin$ *snd* (*state s*)) $\wedge$
$(\forall\, x\ y.\ x \rightarrow+\ y \longrightarrow x \in$ *snd* (*state s*) $\longrightarrow y \in$ *snd* (*state s*)))

lemma *nested-dfs-invarI*:
$[\![\neg$ *has-cycle s*;
$\forall\, x \in \mathcal{A} \cap$ *finished s*. $\neg\ x \rightarrow+\ x$;
set (*stack s*) $\cap$ *snd* (*state s*) = $\{\}$;
$\bigwedge x.\ x \notin$ *discovered s* $\Longrightarrow x \notin$ *snd* (*state s*);
$\bigwedge x\ y.\ x \rightarrow+\ y \Longrightarrow x \in$ *snd* (*state s*) $\Longrightarrow y \in$ *snd* (*state s*)$]\!] \Longrightarrow$ *nested-dfs-invar s*
unfolding *nested-dfs-invar-def*
by *simp*

definition *nested-dfs* :: ((*bool* $\times$ *'n set*, *'n*) *dfs-sws* $\Rightarrow$ *'n set*) $\Rightarrow$ (*bool* $\times$ *'n set*, *'n*) *dfs-algorithm-invar* **where**
nested-dfs M = $(\!|$ *dfs-cond* = *Not* $\circ$ *fst*, *dfs-action* = λ *S - -*. *S*, *dfs-post* = *run-sub-dfs M*, *dfs-remove* = λ *S - -*. *S*, *dfs-start* = $\lambda x.$ (*False*, $\{\}$), *dfs-restrict* = $\{\}$, *dfs-invar* = *nested-dfs-invar* $|\!)$

lemma *nested-dfs-simps*[*simp*]:
dfs-cond (*nested-dfs M*) *S* $\longleftrightarrow \neg$ *fst S*
dfs-post (*nested-dfs M*) = *run-sub-dfs M*
dfs-action (*nested-dfs M*) *S s x* = *S*
dfs-remove (*nested-dfs M*) *S s x* = *S*
dfs-start (*nested-dfs M*) *x* = (*False*, $\{\}$)
dfs-invar (*nested-dfs M*) = *nested-dfs-invar*
dfs-restrict (*nested-dfs M*) = $\{\}$
unfolding *nested-dfs-def*
by *simp-all*

lemma *nested-dfs-has-cycle-not-next*:
dfs-constructable (*nested-dfs M*) *s* $\Longrightarrow$ *has-cycle s* $\Longrightarrow \neg$ *dfs-next* (*nested-dfs M*)

s s'
unfolding *dfs-next-def* **by** *auto*

lemma *ewalk-verts-vwalk-to-ewalk*:
xs ≠ [] ⟹ *vwalk xs* 𝒢 ⟹ *hd xs* = *u* ⟹ *ewalk-verts u* (*vwalk-to-ewalk xs*) = *xs*
unfolding *vwalk-to-ewalk-def ewalk-verts-def*
proof (*induct xs rule*: *list-nonempty-induct*)
case (*single x*) **thus** *?case* **by** *simp*
next
case (*cons x xs*) **hence** *xu*: *x* = *u* **by** *simp*
note *thms* = *cons.prems* ‹*xs* ≠ []›

moreover have ∗: ⋀*x y*. (*x*,*y*) ∈ *E* ⟹ (*SOME e*. *e* ∈ *E* ∧ *e* = (*x*, *y*)) = (*x*, *y*) **by** (*intro some-equality*) *simp-all*
moreover from *thms* **have** *set* (*vwalk-edges* (*x*#*xs*)) ⊆ *E* **unfolding** *vwalk-def* **by** *simp*
moreover with *thms* **have** *set* (*vwalk-edges xs*) ⊆ *E* **by** *simp*
with ‹*xs* ≠ []› **have** *map* (*snd* ∘ (λ*ee*. *SOME e*. *e* ∈ *E* ∧ *e* = *ee*)) (*vwalk-edges xs*) = *tl xs* **by** (*induct xs rule*: *list-nonempty-induct*) (*simp-all add*: ∗)
ultimately show *?case* **by** *simp*
qed

lemma *vwalk-restr-reachable1-intro*:
assumes *vwalk xs* 𝒢
and *y* ∈ *set xs*
and *set xs* ∩ *R* = {}
and *y* ≠ *last xs*
shows *y* →*R*+ *last xs*
proof −
from *vwalk-join-split*[*OF assms*(*2*)] **obtain** *q r* **where** *xs-split*: *xs* = *q* ⊕ *r* *joinable q r* **and** *r-def*: *last r* = *last xs hd r* = *y* **by** (*auto simp add*: *vwalk-join-last*)
with *vwalk-join-vwalk2 assms*(*1*,*3*) **have** *vwalk r* 𝒢 **by** *simp*
with *vwalk-has-ewalk r-def* **have** *ewalk y* (*vwalk-to-ewalk r*) (*last xs*) **by** *simp*

moreover from *xs-split* **have** *r* ≠ [] **unfolding** *joinable-def* **by** *simp*
with *r-def assms*(*4*) **have** *tl r* ≠ [] **by** (*metis hd-rev singleton-rev-conv tl-obtain-elem*)
hence *vwalk-edges r* ≠ [] **by** (*induct r*) *simp-all*
hence *vwalk-to-ewalk r* ≠ [] **unfolding** *vwalk-to-ewalk-def* **by** *simp*

moreover from *xs-split* **have** *xs* = *butlast q* @ *r* **by** (*simp add*: *vwalk-join-def2*)
with *assms*(*3*) **have** *set r* ∩ *R* = {} **by** *auto*

moreover from ‹*vwalk r* 𝒢› *ewalk-verts-vwalk-to-ewalk r-def* ‹*r* ≠ []› **have** *ewalk-verts y* (*vwalk-to-ewalk r*) = *r* **by** *simp*

ultimately show *?thesis* **by** (*auto intro*!: *restr-reachable1I*)
qed

lemma *nested-dfs-preserves-invar*[*intro*!]:

assumes *M-hd*: $\bigwedge s.\ stack\ s \neq [] \Longrightarrow hd\ (stack\ s) \in M\ s$
shows *dfs-preserves-invar* (*nested-dfs M*)
proof (*rule dfs-preserves-invarI*)
fix *y s s'*
assume *step*: *dfs-next* (*nested-dfs M*) *s s'* **and** *inv-cl*:*dfs-invar-compl* (*nested-dfs M*) *y s*
hence *inv*: $\forall v \in \mathcal{A} \cap finished\ s.\ \neg\ v \rightarrow\!\!+\ v\ set\ (stack\ s) \cap snd\ (state\ s) = \{\}$
$\forall x.\ x \notin discovered\ s \longrightarrow x \notin snd\ (state\ s) \forall x\ y.\ x \rightarrow\!\!+\ y \longrightarrow x \in snd\ (state\ s) \longrightarrow y \in snd\ (state\ s)$
unfolding *dfs-invar-compl-def* **using** *nested-dfs-has-cycle-not-next nested-dfs-invar-def*
by *simp-all blast+*

from *inv-cl* **have** *constr*: *dfs-constructable* (*nested-dfs M*) *s* **by** *auto*
with *step* **have** *constr'*: *dfs-constructable* (*nested-dfs M*) *s'* **using** *dfs-constructable.step*
by *blast*

have *nested-dfs-invar s'*
proof (*cases has-cycle s'*)
case *True* **thus** *?thesis* **by** (*simp add*: *nested-dfs-invar-def*)
next
case *False* **with** *step inv* **show** *?thesis*
proof (*cases rule*: *dfs-next-cases-elem*)
case (*empty x xs*) **hence** *state'*: *state s'* = *run-sub-dfs M* (*state s*) *s x* **by** *simp*
from *step constr* **have** *no-cycl*: ¬ *has-cycle s* **using** *nested-dfs-has-cycle-not-next*
by *blast*
moreover from *inv*(*1−3*) *empty* **have** *x* ∈ *V x* ∉ *snd* (*state s*) **using** *stack-subset-verts*[*OF constr*] **by** *auto*
ultimately show *?thesis*
proof (*cases rule*: *run-sub-dfs-casesE*[*of state s x M s*])
case *same* **with** *inv no-cycl state' empty* **show** *?thesis* **by** (*intro nested-dfs-invarI*) *simp-all*
next
case *cycle* **with** *False state'* **have** *False* **by** *simp*
thus *?thesis* **..**
next
case (*no-cycle F'*) **with** *inv*(*1−3*) *empty M-hd*[*of s*] **have** *no-x-self*: ¬ *x* →*snd* (*state s*)+ *x* **by** *auto*
have *no-x-stack*: $\bigwedge y.\ x \rightarrow\backslash snd\ (state\ s)\!\!+\ y \Longrightarrow y \notin set\ (stack\ s)$
proof (*rule notI*)
fix *y*
assume *x-y*: *x* →*snd* (*state s*)+ *y* **and** *y-stack*: *y* ∈ *set* (*stack s*)

from *x-y no-cycle* **have** *y* ∉ *M s* **by** *simp*
with *M-hd*[*of s*] *empty y-stack* **have** *y* ∈ *set* (*tl* (*stack s*)) **by** *auto*
with *empty tl-reachable-stack-hd*[*OF constr*] **have** *y* →+ *x* **by** *simp*
moreover from *y-stack empty inv*(*2*) **have** *y* ∉ *snd* (*state s*) *x* ∉ *snd* (*state s*) **by** *auto*
ultimately have *y* →*snd* (*state s*)+ *x* **using** *reach-impl-restr-reach*

```
inv(4) by blast
        with x-y no-x-self restr-reachable1-trans show False by blast
       qed

       show ?thesis
       proof (intro nested-dfs-invarI)
        from no-cycle state' show ¬ has-cycle s' by simp
       next
        have ¬ x →+ x
        proof (rule notI)
         assume x →+ x
         moreover from empty inv(2) have x ∉ snd (state s) by auto
        ultimately show False using no-x-self reach-impl-restr-reach inv(4) by
blast
        qed
        with no-cycle inv(1−3) empty show ∀ x ∈ 𝒜 ∩ finished s'. ¬ x→+x by
auto
       next
        from inv(2) empty have set (stack s') ∩ snd (state s) = {} by simp
        moreover {
         fix y
         assume A: y ∈ set (stack s') y ∈ F'
         with no-cycle have x = y ∨ x →\snd (state s)+ y by auto
         hence False
         proof (rule disjE)
            case goal1 with A empty stack-distinct[OF constr] show False by
simp
         next
          case goal2 with A empty no-x-stack show False by simp
         qed
        }
        hence F' ∩ set (stack s') = {} by auto
       ultimately show set (stack s') ∩ snd (state s') = {} using no-cycle state'
by auto
       next
        fix y
        assume ndisc: y ∉ discovered s'
        hence y ∉ F'
        proof (rule contrapos-nn)
         assume y ∈ F'
         with no-cycle have y = x ∨ x →\snd (state s)+ y by simp
           with empty stack-subset-discovered[OF constr] ndisc have x →\snd
(state s)+ y by auto
         thus y ∈ discovered s'
         proof induction
          case base from empty have x ∈ finished s' by simp
            with finished-implies-succs-discovered[OF constr'] base show ?case
by auto
         next
```

case (*trans y z*) **with** *no-x-stack* **have** $y \notin set\ (stack\ s)$ **by** *simp*
with *empty* **have** $y \notin set\ (stack\ s')$ **by** *simp*
with *trans discovered-non-stack-implies-finished*[*OF constr'*] **have** $y \in$ *finished* s' **by** *simp*
with *finished-implies-succs-discovered*[*OF constr'*] *trans* **show** *?case* **by** *auto*
qed
qed
with *ndisc inv*(*3*) *empty* **have** $y \notin F' \cup snd\ (state\ s)$ **by** *simp*
with *no-cycle state'* **show** $y \notin snd\ (state\ s')$ **by** *simp*
next
fix *y z*
assume *reach*: $y \rightarrow+ z$ **and** *in-s*: $y \in snd\ (state\ s')$
with *no-cycle state'* **have** $y \in snd\ (state\ s) \vee y \in F'$ **by** *simp*
thus $z \in snd\ (state\ s')$
proof (*rule disjE*)
assume $y \in snd\ (state\ s)$ **with** *reach inv*(*4*) *no-cycle state'* **show** *?thesis* **by** *simp*
next
assume *A*: $y \in F'$ **with** *reach no-cycle* ‹$x \notin snd\ (state\ s)$› **have** *y-notin*: $y \notin snd\ (state\ s)$ **using** *restricted-target* **by** *blast*
thus *?thesis* **using** *no-cycle state'*
proof (*cases* $z \in snd\ (state\ s)$)
case *False* **with** *reach-impl-restr-reach inv*(*4*) *y-notin reach* **have** $y \rightarrow\backslash snd\ (state\ s)+ z$ **by** *blast*
with *A no-cycle restr-reachable1-trans* **have** $x \rightarrow\backslash snd\ (state\ s)+ z$ **by** *blast*
with *no-cycle* **have** $z \in F'$ **by** *blast*
with *state' no-cycle* **show** *?thesis* **by** *simp*
qed *simp*
qed
qed
qed
qed (*simp-all add*: *nested-dfs-invar-def*)
qed
thus *dfs-invar* (*nested-dfs M*) *s'* **by** *simp*
qed (*auto simp add*: *nested-dfs-invar-def*)

definition *cyclic* **where** $cyclic\ x\ s \equiv has\text{-}cycle\ s \wedge (\exists v \in finished\ s \cap \mathcal{A}.\ v \rightarrow+ v \wedge x \rightarrow\star v)$
definition *cycle* **where** $cycle\ M\ x \equiv \exists s \in dfs\text{-}constr\text{-}from\ (nested\text{-}dfs\ M)\ x.\ cyclic\ x\ s$

lemma *has-cycle-implies-cyclic*:
$dfs\text{-}constructable\ (nested\text{-}dfs\ M)\ s \Longrightarrow (\bigwedge s.\ stack\ s \neq [] \Longrightarrow hd\ (stack\ s) \in M\ s) \Longrightarrow (\bigwedge s.\ stack\ s \neq [] \Longrightarrow M\ s \subseteq set\ (stack\ s)) \Longrightarrow has\text{-}cycle\ s \Longrightarrow cyclic\ (start\ s)\ s$
proof (*induction rule*: *dfs-constructable-induct*)
case (*empty s s' x*) **hence** *state'*: $state\ s' = run\text{-}sub\text{-}dfs\ M\ (state\ s)\ s\ x$ **by** *simp*

from *start-reachable-stack*[*OF empty*(*2*)] *empty* **have** *sreach*: $\bigwedge x.\ x \in set\ (stack\ s) \Longrightarrow start\ s' \rightarrow\star\ x$ **by** *auto*

from *empty* **have** *no-cycle*: $\neg$ *has-cycle s* **using** *nested-dfs-has-cycle-not-next* **by** *force*

moreover from *empty stack-subset-verts*[*OF empty*(*2*)] **have** $x \in V$ **by** *auto*

moreover
have *inv*: *nested-dfs-invar s* **using** *dfs-constructable-invarI*[*OF empty*(*2*)] *empty* **by** *auto*
with *no-cycle* **have** *set* (*stack s*) $\cap$ *snd* (*state s*) = {} **unfolding** *nested-dfs-invar-def* **by** *simp*
with *empty* **have** $x \notin$ *snd* (*state s*) **by** *simp*

ultimately show *?case*
proof (*cases rule*: *run-sub-dfs-casesE*[*of state s x M s*])
case *same* **with** *state'* *no-cycle empty.prems* **have** *False* **by** *simp*
thus *?thesis* **..**
next
case (*cycle v*)
thus *?thesis*
proof (*cases* $v = x$)
case *True* **with** *cycle* **show** *?thesis* **using** *empty sreach* **unfolding** *cyclic-def* **by** *auto*
next
case *False* **from** *cycle* **have** $x \rightarrow+\ v$ **by** *blast*
also from *empty* **have** *stack s* $\neq$ [] **by** *simp*
with ‹$v \in M\ s$› *empty.prems*(*2*) **have** $v \in set\ (stack\ s)$ **by** *auto*
with ‹$v \neq x$› *empty* **have**$v \in set\ (tl\ (stack\ s))$ **by** *auto*
with *tl-reachable-stack-hd*[*OF empty*(*2*)] *empty* **have** $v \rightarrow+\ x$ **by** *simp*
finally show *?thesis* **using** *empty cycle sreach* **unfolding** *cyclic-def* **by** *auto*
qed
next
case *no-cycle* **with** *state' empty.prems* **have** *False* **by** *simp*
thus *?thesis* **..**
qed
qed (*simp-all add*: *nested-dfs-has-cycle-not-next cyclic-def*)

lemma *cycle-not-completed*:
assumes *M-hd*: $\bigwedge s.\ stack\ s \neq [] \Longrightarrow hd\ (stack\ s) \in M\ s$
and *reach*: $x \rightarrow\star\ v$
and *cycle*: $v \rightarrow+\ v$
and *fin*: $v \in \mathcal{A}$
and *constr*: $s \in$ *dfs-constr-from* (*nested-dfs M*) x
shows $\neg$ *dfs-completed* (*nested-dfs M*) *s*
proof (*rule notI*)
assume *compl*: *dfs-completed* (*nested-dfs M*) *s*

hence ¬ *has-cycle s* **by** (*simp add*: *dfs-completed-def*)

from *compl no-restrict-finished-eq-reachable*[*of nested-dfs M s*] *constr-from-implies-start*[*OF constr*] **have** *finished s* = {*v*. *x* →⋆ *v*} **by** *simp*
hence *v* ∈ *finished s* **using** *reach* **by** *simp*

moreover
from *compl M-hd* **have** *dfs-invar* (*nested-dfs M*) *s* **by** *blast*
with ‹¬ *has-cycle s*› **have** ∀ *v* ∈ 𝒜 ∩ *finished s*. ¬ *v* →+ *v* **by** (*simp add*: *nested-dfs-invar-def*)
with *cycle fin* **have** *v* ∉ *finished s* **by** *auto*

ultimately show *False* **by** *contradiction*
qed

lemma *cycle-generates-cyclic*:
assumes *M-hd*: ⋀*s*. *stack s* ≠ [] ⟹ *hd* (*stack s*) ∈ *M s* **and** *M-sse*: ⋀*s*. *stack s* ≠ [] ⟹ *M s* ⊆ *set* (*stack s*)
and *reach*: *x* →⋆ *v*
and *cycle*: *v* →+ *v*
and *fin*: *v* ∈ 𝒜
obtains *s* **where** *s* ∈ *dfs-constr-from* (*nested-dfs M*) *x cyclic x s*
proof −
from *reach* **have** *x* ∈ *V* **by** *auto*
with *exists-finished*[*OF - - nested-dfs-preserves-invar*[*OF M-hd*]] **obtain** *s* **where** *s*: *s* ∈ *dfs-constr-from* (*nested-dfs M*) *x dfs-finished* (*nested-dfs M*) *s* **by** *auto*
moreover with *cycle-not-completed assms* **have** ¬ *dfs-completed* (*nested-dfs M*) *s* **by** *simp*
ultimately have ¬ *dfs-cond* (*nested-dfs M*) (*state s*) **using** *finished-is-completed-or-not-cond* **by** *blast*
hence *has-cycle s* **by** *simp*
with *s has-cycle-implies-cyclic that constr-from-implies-start M-hd M-sse* **show** *?thesis* **by** *blast*
qed

lemma *cycle-get-cycle*:
assumes *cycle M x*
obtains *v* **where** *x* →⋆ *v v* ∈ 𝒜 *v* →+ *v*
proof −
from *assms* **obtain** *s* **where** *constr*: *s* ∈ *dfs-constr-from* (*nested-dfs M*) *x* **and** *cyclic x s* **unfolding** *cycle-def* **by** *auto*
then obtain *v* **where** *v* ∈ *finished s* ∩ 𝒜 *v* →+ *v x* →⋆ *v* **unfolding** *cyclic-def* **by** *blast*
with *that* **show** *?thesis* **by** *auto*
qed

lemma *cycle-iff-cycle*:
assumes *M-correct*: ⋀*s*. *stack s* ≠ [] ⟹ *hd* (*stack s*) ∈ *M s* ⋀*s*. *stack s* ≠ [] ⟹ *M s* ⊆ *set* (*stack s*)

shows *cycle M x* $\longleftrightarrow$ ($\exists\, v.\ x \rightarrow\star\ v \wedge v \in \mathcal{A} \wedge v \rightarrow+\ v$)
proof
assume *cycle M x* **thus** $\exists\, v.\ x \rightarrow\star\ v \wedge v \in \mathcal{A} \wedge v \rightarrow+\ v$ **by** (*metis cycle-get-cycle*)
next
assume $\exists\, v.\ \ x \rightarrow\star\ v \wedge v \in \mathcal{A} \wedge v \rightarrow+\ v$
then guess *v* **..**
then obtain *s* **where** *cyclic x s s* $\in$ *dfs-constr-from* (*nested-dfs M*) *x* **using** *cycle-generates-cyclic*[*OF M-correct*] **by** *blast*
thus *cycle M x* **unfolding** *cycle-def* **by** *blast*
qed

lemma *dfs-fun-nested-dfs-correct*:
assumes $x \in V$
and *M-correct*: $\bigwedge s.$ *stack s* $\neq$ [] $\Longrightarrow$ *hd* (*stack s*) $\in$ *M s* $\bigwedge s.$ *stack s* $\neq$ [] $\Longrightarrow$ *M s* $\subseteq$ *set* (*stack s*)
shows *dfs-fun* (*nested-dfs M*) *x* $\leq$ *SPEC* ($\lambda s.$ *has-cycle s* $\longleftrightarrow$ *cycle M x*)
using *assms*
proof (*intro dfs-fun-pred*)
fix *s*
assume *s* $\in$ *dfs-constr-from* (*nested-dfs M*) *x has-cycle s*
moreover hence *cyclic x s* **using** *has-cycle-implies-cyclic*[*OF - M-correct*] *constr-from-implies-start* **by** *force*
ultimately show *cycle M x* **by** (*auto simp*: *cycle-def*)
next
fix *s*
assume *s* $\in$ *dfs-constr-from* (*nested-dfs M*) *x dfs-completed* (*nested-dfs M*) *s cycle M x*
hence *False* **by** (*metis cycle-iff-cycle M-correct cycle-not-completed*)
thus *has-cycle s* **..**
qed (*simp-all add*: *nested-dfs-preserves-invar*)

theorem *nested-dfs-correct*:
assumes $x \in V$
and $\bigwedge s.$ *stack s* $\neq$ [] $\Longrightarrow$ *hd* (*stack s*) $\in$ *M s* $\bigwedge s.$ *stack s* $\neq$ [] $\Longrightarrow$ *M s* $\subseteq$ *set* (*stack s*)
and *inres* (*dfs-fun* (*nested-dfs M*) *x*) *s*
shows *has-cycle s* $\longleftrightarrow$ *cycle M x*
using *assms*(*4*) *dfs-fun-nested-dfs-correct*[*OF assms*(*1*−*3*)]
by (*bestsimp dest*!: *pwD2*)

lemma *RETURN-nested-cycle-correct*:
assumes $x \in V$
and $\bigwedge s.$ *stack s* $\neq$ [] $\Longrightarrow$ *hd* (*stack s*) $\in$ *M s* $\bigwedge s.$ *stack s* $\neq$ [] $\Longrightarrow$ *M s* $\subseteq$ *set* (*stack s*)
shows *RETURN* (*cycle M x*) = *do* { *s* $\leftarrow$ *dfs-fun* (*nested-dfs M*) *x*; *RETURN* (*has-cycle s*) }
apply (*rule pw-eqI*)
apply (*simp-all add*: *pw-bind-inres pw-bind-nofail dfs-fun-nofail*[*OF nested-dfs-preserves-invar*[*OF assms*(*2*)] *assms*(*1*)])

apply (*metis assms dfs-fun-nonempty nested-dfs-correct*)
done

end

2.3 Basic Nested DFS

context *finite-accepting-digraph*
begin

definition $basicM \equiv \lambda s.\ \{\ hd\ (stack\ s)\ \}$
definition *basic-dfs* $\equiv$ *nested-dfs basicM*

lemmas *basic-dfs-simps*[*simp*] = *nested-dfs-simps*[**where** *M*=*basicM*, *folded basic-dfs-def*]

lemma *basicM-correct*[*simp*]:
$stack\ s \neq [] \Longrightarrow hd\ (stack\ s) \in basicM\ s$
$stack\ s \neq [] \Longrightarrow basicM\ s \subseteq set\ (stack\ s)$
unfolding *basicM-def*
by *simp-all*

lemma *basic-dfs-preserves-invar*:
dfs-preserves-invar basic-dfs
by (*simp add*: *basic-dfs-def nested-dfs-preserves-invar*)

lemma *basic-cycle-iff-cycle*:
$cycle\ basicM\ x \longleftrightarrow (\exists\, v.\ x \rightarrow\star\ v \wedge v \in \mathcal{A} \wedge v \rightarrow+\ v)$
by (*simp add*: *cycle-iff-cycle*)

lemma *dfs-fun-basic-dfs-correct*:
$x \in V \Longrightarrow dfs\text{-}fun\ basic\text{-}dfs\ x \leq SPEC\ (\lambda s.\ has\text{-}cycle\ s \longleftrightarrow cycle\ basicM\ x)$
by (*simp add*: *dfs-fun-nested-dfs-correct basic-dfs-def*)

theorem *basic-dfs-correct*:
$x \in V \Longrightarrow inres\ (dfs\text{-}fun\ basic\text{-}dfs\ x)\ s \Longrightarrow has\text{-}cycle\ s \longleftrightarrow cycle\ basicM\ x$
by (*simp add*: *nested-dfs-correct basic-dfs-def*)

theorems *basic-dfs-correct-unfolded* = *basic-dfs-correct*[*unfolded basic-cycle-iff-cycle*]
end

2.4 HPY

context *finite-accepting-digraph*
begin

definition $hpyM \equiv \lambda s.\ set\ (stack\ s)$
definition *hpy-dfs* $\equiv$ *nested-dfs hpyM*

lemmas *hpy-dfs-simps*[*simp*] = *nested-dfs-simps*[**where** *M*=*hpyM*, *folded hpy-dfs-def*]

lemma *hpyM-correct*[*simp*]:
stack s $\neq$ [] $\Longrightarrow$ *hd* (*stack s*) $\in$ *hpyM s*
stack s $\neq$ [] $\Longrightarrow$ *hpyM s* $\subseteq$ *set* (*stack s*)
unfolding *hpyM-def*
by *simp-all*

lemma *hpy-dfs-preserves-invar*:
dfs-preserves-invar hpy-dfs
by (*simp add*: *hpy-dfs-def nested-dfs-preserves-invar*)

lemma *hpy-cycle-iff-cycle*:
cycle hpyM x $\longleftrightarrow$ ($\exists$ *v*. *x* $\rightarrow\star$ *v* $\wedge$ *v* $\in$ $\mathcal{A}$ $\wedge$ *v* $\rightarrow+$ *v*)
by (*simp add*: *cycle-iff-cycle*)

lemma *dfs-fun-hpy-dfs-correct*:
x $\in$ *V* $\Longrightarrow$ *dfs-fun hpy-dfs x* $\leq$ *SPEC* (λ*s*. *has-cycle s* $\longleftrightarrow$ *cycle hpyM x*)
by (*simp add*: *dfs-fun-nested-dfs-correct hpy-dfs-def*)

theorem *hpy-dfs-correct*:
x $\in$ *V* $\Longrightarrow$ *inres* (*dfs-fun hpy-dfs x*) *s* $\Longrightarrow$ *has-cycle s* $\longleftrightarrow$ *cycle hpyM x*
by (*simp add*: *nested-dfs-correct hpy-dfs-def*)

theorems *hpy-dfs-correct-unfolded* = *hpy-dfs-correct*[*unfolded hpy-cycle-iff-cycle*]
end

2.5 Schwoon-Esparza

context *finite-accepting-digraph*
begin

fun *SE-remove* **where**
SE-remove (*True*, *F*) - - = (*True*, *F*)
| *SE-remove* (*False*, *F*) *s x* = ((*x* $\in$ $\mathcal{A}$ $\vee$ *hd* (*stack s*) $\in$ $\mathcal{A}$) $\wedge$ *x* $\in$ *set* (*stack s*), *F*)

definition *SE-dfs* :: (*bool* $\times$ '*n set*, '*n*) *dfs-algorithm-invar* **where**
SE-dfs = *hpy-dfs*(| *dfs-remove* := *SE-remove* |)

lemma *SE-dfs-simps*[*simp*]:
dfs-cond SE-dfs S $\longleftrightarrow$ $\neg$ *fst S*
dfs-post SE-dfs = *run-sub-dfs hpyM*
dfs-action SE-dfs S s x = *S*
dfs-remove SE-dfs = *SE-remove*
dfs-start SE-dfs x = (*False*, {})
dfs-invar SE-dfs = *nested-dfs-invar*
dfs-restrict SE-dfs = {}
unfolding *SE-dfs-def*
by *simp-all*

lemma *SE-dfs-has-cycle-not-next*:
dfs-constructable SE-dfs s $\Longrightarrow$ *has-cycle s* $\Longrightarrow$ $\neg$ *dfs-next SE-dfs s s'*
unfolding *dfs-next-def* **by** *auto*

lemma *SE-to-nested*:
s $\in$ *dfs-constr-from SE-dfs x* $\Longrightarrow$ $\neg$ *has-cycle s* $\Longrightarrow$ *s* $\in$ *dfs-constr-from hpy-dfs x*
proof (*induction rule*: *dfs-constr-from.induct*)
case *start*
have *dfs-constr-start SE-dfs x* = *dfs-constr-start hpy-dfs x* **by** (*simp add*: *dfs-constr-start-def*)
with *start* **show** *?case* **by** (*simp add*: *dfs-constructable.start*)
next
case (*step s s'*) **with** *SE-dfs-has-cycle-not-next* **have** *no*: $\neg$ *has-cycle s* **by** *blast*
with *step.IH* **have** *constr*: *s* $\in$ *dfs-constr-from hpy-dfs x* **.**

from *step* **have** *dfs-cond-compl hpy-dfs s* **unfolding** *dfs-next-def dfs-cond-compl-def*
by *simp*
note *thms* = *step*(*1*,*4*) *no constr this*

from *step*(*2*) **have** *dfs-next hpy-dfs s s'*
proof (*cases rule*: *dfs-next-cases-elem*)
case *empty* **thus** *?thesis* **by** (*simp add*: *dfs-next-def dfs-step-def dfs-sws.defs*
thms)
next
case *restrict* **thus** *?thesis* **by** (*simp add*: *dfs-next-def dfs-step-def dfs-sws.defs*
thms)
next
case (*remove e*)
obtain *b F* **where** *bf*: *state s* = (*b*,*F*) **by** (*cases state s*) *auto*
with *remove* ‹$\neg$ *has-cycle s'*› **have** *state s'* = (*b*, *F*) **by** (*cases b*) *simp-all*
with *remove bf* **show** *?thesis* **by** (*simp add*: *dfs-next-def dfs-sws.defs thms*
dfs-step-def exI[**where** *x*=*e*])
next
case (*visit e*) **obtain** *b F* **where** *bf*: *state s* = (*b*,*F*) **by** (*cases state s*) *auto*
with *visit* **have** *state s'* = (*b*, *F*) **by** *simp*
with *visit bf* **show** *?thesis* **by** (*simp add*: *dfs-next-def dfs-sws.defs thms*
dfs-step-def exI[**where** *x*=*e*])
qed
with *constr* **show** *?case* **by** (*rule dfs-constr-from.step*)
qed

corollary *SE-to-nested'*:
dfs-constructable SE-dfs s $\Longrightarrow$ $\neg$ *has-cycle s* $\Longrightarrow$ *dfs-constructable hpy-dfs s*
using *SE-to-nested dfs-constructable-constr-from-start dfs-constr-from-constructable*
by *metis*

lemma *SE-dfs-preserves-invar*[*intro*]:
dfs-preserves-invar SE-dfs
proof (*rule dfs-preserves-invarI*)
fix *y s s'*

assume *step*: *dfs-next SE-dfs s s'* **and** *inv-cl*:*dfs-invar-compl SE-dfs y s*
hence *inv*: $\forall\, v \in \mathcal{A} \cap$ *finished s*. $\neg\ v \rightarrow\!+\ v$ *set* (*stack s*) $\cap$ *snd* (*state s*) = {}
$\forall\, x.\ x \notin$ *discovered s* $\longrightarrow\ x \notin$ *snd* (*state s*)$\forall\, x\ y.\ x \rightarrow\!+\ y \longrightarrow\ x \in$ *snd* (*state s*) $\longrightarrow\ y \in$ *snd* (*state s*)
unfolding *dfs-invar-compl-def* **using** *SE-dfs-has-cycle-not-next nested-dfs-invar-def* **by** *simp-all blast+*

from *inv-cl* **have** *constr*: $s \in$ *dfs-constr-from SE-dfs y* **by** *auto*
with *step* **have** *constr'*: $s' \in$ *dfs-constr-from SE-dfs y* **using** *dfs-constr-from.step* **by** *blast*

have *nested-dfs-invar s'*
proof (*cases has-cycle s'*)
case *True* **thus** *?thesis* **by** (*simp add*: *nested-dfs-invar-def*)
next
case *False* **with** *step inv* **show** *?thesis*
proof (*cases rule*: *dfs-next-cases-elem*)
case (*empty x xs*) **with** *step* **have** *dfs-next hpy-dfs s s'* **unfolding** *dfs-next-def dfs-cond-compl-def dfs-step-def* **by** *simp*
moreover from *constr step SE-to-nested* **have** $s \in$ *dfs-constr-from hpy-dfs y* **using** *SE-dfs-has-cycle-not-next* **by** *blast*
with *inv-cl* **have** *dfs-invar-compl hpy-dfs y s* **unfolding** *dfs-invar-compl-def* **by** *simp*
ultimately have *dfs-invar hpy-dfs s'* **using** *hpy-dfs-preserves-invar* **by** (*metis dfs-preserves-invarE*)
thus *?thesis* **by** *simp*
next
case *visit* **with** *step inv* **show** *?thesis* **by** (*simp add*: *nested-dfs-invar-def*)
next
case *restrict* **with** *step inv* **show** *?thesis* **by** (*simp add*: *nested-dfs-invar-def*)
next
case (*remove e*) **hence** *snd* (*SE-remove* (*state s*) *s e*) = *snd* (*state s*) **by** (*cases state s*, *cases fst* (*state s*)) *simp-all*
with *remove step inv* **show** *?thesis* **by** (*simp add*: *nested-dfs-invar-def*)
qed
qed
thus *dfs-invar SE-dfs s'* **by** *simp*
qed (*auto simp add*: *nested-dfs-invar-def*)

definition *SE-cyclic* **where** *SE-cyclic s* $\equiv$ *has-cycle s* $\wedge$ (*cyclic* (*start s*) *s* $\vee$ ($\exists$ *x* $\in$ *set* (*stack s*). *x* $\in \mathcal{A} \wedge$ *x* $\rightarrow\!+$ *x*))
definition *SE-cycle* **where** *SE-cycle x* $\equiv \exists\, s.\ s \in$ *dfs-constr-from SE-dfs x* $\wedge$ *SE-cyclic s*

lemma *has-cycle-implies-SE-cyclic*:
dfs-constructable SE-dfs s $\Longrightarrow$ *has-cycle s* $\Longrightarrow$ *SE-cyclic s*
proof (*induction rule*: *dfs-constructable-induct*)
case (*empty s s'*) **with** *SE-dfs-has-cycle-not-next* **have** $\neg$ *has-cycle s* **by** *blast*
with *empty SE-to-nested'* **have** *dfs-constructable hpy-dfs s* **by** *simp*

moreover from *empty* **have** *dfs-next hpy-dfs s s'* **by** (*simp add*: *dfs-next-def dfs-cond-compl-def dfs-step-def*)
ultimately have *dfs-constructable hpy-dfs s'* **using** *dfs-constructable.step* **by** *blast*
with *empty.prems* **have** *cyclic* (*start s'*) *s'* **using** *has-cycle-implies-cyclic*[*OF - hpyM-correct*, *of s'*, *folded hpy-dfs-def*] **by** *metis*
with *empty.prems* **show** *?case* **unfolding** *SE-cyclic-def* **by** *simp*
next
case (*remove s s' e x*) **with** *SE-dfs-has-cycle-not-next* **have** ¬ *has-cycle s* **by** *blast*
with *remove* **have** $e \in set\ (stack\ s) \wedge (e \in \mathcal{A} \vee x \in \mathcal{A})$ **by** (*cases state s*, *cases fst* (*state s*)) *simp-all*
hence *e-stack*: $e \in set\ (stack\ s)$ **and** *A*: $e \in \mathcal{A} \vee x \in \mathcal{A}$ **by** *simp-all*

from *remove* **have** $e \in succs\ x$ **using** *wl-subset-succs*[*OF remove*(*2*), *of 0*] **by** *auto*
with *succs-reachable* **have** *xe*: $x \rightarrow+ e$ **by** *auto*

have $\exists y \in set\ (stack\ s').\ y \rightarrow+ y \wedge y \in \mathcal{A}$
proof (*cases* $e = x$)
case *True* **with** *xe A remove* **show** *?thesis* **by** *auto*
next
case *False* **with** *remove e-stack* **have** $e \in set\ (tl\ (stack\ s))$ **by** *auto*
with *tl-reachable-stack-hd*[*OF remove*(*2*)] *remove* **have** *ex*: $e \rightarrow+ x$ **by** *auto*
from *A* **show** *?thesis*
proof
assume $e \in \mathcal{A}$
moreover note *xe ex*
moreover from *e-stack remove* **have** $e \in set\ (stack\ s')$ **by** *auto*
ultimately show *?thesis* **using** *reachable1-trans* **by** *blast*
next
assume $x \in \mathcal{A}$
moreover note *xe ex*
moreover from *remove* **have** $x \in set\ (stack\ s')$ **by** *auto*
ultimately show *?thesis* **using** *reachable1-trans* **by** *blast*
qed
qed
with *remove.prems* **show** *?case* **unfolding** *SE-cyclic-def* **by** *blast*
qed (*simp-all add*: *SE-dfs-has-cycle-not-next SE-cyclic-def*)

lemma *SE-cycle-not-completed*:
assumes *reach*: $x \rightarrow\star v$
and *cycle*: $v \rightarrow+ v$
and *fin*: $v \in \mathcal{A}$
and *constr*: $s \in dfs\text{-}constr\text{-}from\ SE\text{-}dfs\ x$
shows ¬ *dfs-completed SE-dfs s*
proof (*rule notI*)
assume *compl*: *dfs-completed SE-dfs s*
hence ¬ *has-cycle s* **by** (*simp add*: *dfs-completed-def*)

from *compl no-restrict-finished-eq-reachable*[*of SE-dfs s*] *constr-from-implies-start*[*OF constr*] **have** *finished s* = {*v*. *x* →⋆ *v*} **by** *simp*
hence *v* ∈ *finished s* **using** *reach* **by** *simp*

moreover
from *compl* **have** *dfs-invar SE-dfs s* **by** *blast*
with ‹¬ *has-cycle s*› **have** ∀ *v* ∈ *A* ∩ *finished s*. ¬ *v* →+ *v* **by** (*simp add*: *nested-dfs-invar-def*)
with *cycle fin* **have** *v* ∉ *finished s* **by** *auto*

ultimately show *False* **by** *contradiction*
qed

lemma *cycle-generates-SE-cyclic*:
assumes *reach*: *x* →⋆ *v*
and *cycle*: *v* →+ *v*
and *fin*: *v* ∈ *A*
obtains *s* **where** *s* ∈ *dfs-constr-from SE-dfs x SE-cyclic s*
proof −
from *reach* **have** *x* ∈ *V* **by** *auto*
with *exists-finished*[*OF - - SE-dfs-preserves-invar*] **obtain** *s* **where** *s*: *s* ∈ *dfs-constr-from SE-dfs x dfs-finished SE-dfs s* **by** *auto*
moreover with *SE-cycle-not-completed assms* **have** ¬ *dfs-completed SE-dfs s* **by** *simp*
ultimately have ¬ *dfs-cond SE-dfs* (*state s*) **using** *finished-is-completed-or-not-cond* **by** *blast*
hence *has-cycle s* **by** *simp*
with *s has-cycle-implies-SE-cyclic that* **show** *?thesis* **by** *blast*
qed

lemma *SE-cycle-get-cycle*:
assumes *SE-cycle x*
obtains *v* **where** *x* →⋆ *v v* ∈ *A v* →+ *v*
proof −
from *assms* **obtain** *s* **where** *constr*: *s* ∈ *dfs-constr-from SE-dfs x* **and** *SE-cyclic s* **unfolding** *SE-cycle-def* **by** *auto*
show *?thesis*
proof (*cases cyclic x s*)
case *True* **then obtain** *v* **where** *v* ∈ *finished s* ∩ *A v* →+ *v x* →⋆ *v* **unfolding** *cyclic-def* **by** *blast*
with *that* **show** *?thesis* **by** *blast*
next
case *False*
from *constr constr-from-implies-start* **have** *start*: *start s* = *x* **by** *blast*
with *False* ‹*SE-cyclic s*› **obtain** *v* **where** *v* ∈ *set* (*stack s*) *v* ∈ *A v* →+*v* **unfolding** *SE-cyclic-def* **by** *auto*
with *start start-reachable-stack constr that* **show** *?thesis* **by** *auto*
qed

qed

lemma *SE-cycle-iff-cycle*:
SE-cycle x $\longleftrightarrow$ ($\exists\, v.\ x \rightarrow\star\ v \wedge v \in \mathcal{A} \wedge v \rightarrow+\ v$)
proof
assume *SE-cycle x* **thus** $\exists\, v.\ x \rightarrow\star\ v \wedge v \in \mathcal{A} \wedge v \rightarrow+\ v$ **by** (*metis SE-cycle-get-cycle*)
next
assume $\exists\, v.\ \ x \rightarrow\star\ v \wedge v \in \mathcal{A} \wedge v \rightarrow+\ v$
then guess *v* **..**
then obtain *s* **where** *SE-cyclic s s* $\in$ *dfs-constr-from SE-dfs x* **using** *cycle-generates-SE-cyclic*
by *blast*
thus *SE-cycle x* **unfolding** *SE-cycle-def* **by** *blast*
qed

lemma *dfs-fun-SE-dfs-correct*:
assumes $x \in V$
shows *dfs-fun SE-dfs x* $\leq$ *SPEC* ($\lambda s.$ *has-cycle s* $\longleftrightarrow$ *SE-cycle x*)
using *assms*
proof (*intro dfs-fun-pred*)
fix *s*
assume *s* $\in$ *dfs-constr-from SE-dfs x has-cycle s*
moreover hence *SE-cyclic s* **using** *has-cycle-implies-SE-cyclic* **by** *force*
ultimately show *SE-cycle x* **by** (*auto simp*: *SE-cycle-def*)
next
fix *s*
assume *s* $\in$ *dfs-constr-from SE-dfs x dfs-completed SE-dfs s SE-cycle x*
hence *False* **by** (*metis SE-cycle-iff-cycle SE-cycle-not-completed*)
thus *has-cycle s* **..**
qed (*simp-all add*: *SE-dfs-preserves-invar*)

theorem *SE-dfs-correct*:
assumes $x \in V$
and *inres* (*dfs-fun SE-dfs x*) *s*
shows *has-cycle s* $\longleftrightarrow$ *SE-cycle x*
using *assms dfs-fun-SE-dfs-correct*
by (*bestsimp dest*!: *pwD2*)

theorems *SE-dfs-correct-unfolded* = *SE-dfs-correct*[*unfolded SE-cycle-iff-cycle*]

lemma *RETURN-SE-cycle-correct*:
assumes $x \in V$
shows *RETURN* (*SE-cycle x*) = *do* { *s* $\leftarrow$ *dfs-fun SE-dfs x*; *RETURN* (*has-cycle s*) }
apply (*rule pw-eqI*)
apply (*simp-all add*: *pw-bind-inres pw-bind-nofail dfs-fun-nofail*[*OF SE-dfs-preserves-invar assms*])
apply (*metis assms dfs-fun-nonempty SE-dfs-correct*)
done

lemmas *RETURN-SE-cycle-correct-unfolded* = *RETURN-SE-cycle-correct*[*unfolded SE-cycle-iff-cycle*]
end
end

theory *Empty-Set*
imports
Main
../Libs/Collections/Collections
begin

type-synonym *'a es* = *unit*

abbreviation (*input*) *es-α* :: *'a es* ⇒ *'a set* **where** *es-α x* ≡ {}
abbreviation (*input*) *es-empty* :: *unit* ⇒ *'a es* **where** *es-empty x* ≡ ()
abbreviation (*input*) *es-memb* :: *'a* ⇒ *'a es* ⇒ *bool* **where** *es-memb es x* ≡ *False*
abbreviation (*input*) *es-isEmpty* :: *'a es* ⇒ *bool* **where** *es-isEmpty es* ≡ *True*
abbreviation (*input*) *es-to-list* :: *'a es* ⇒ *'a list* **where** *es-to-list es* ≡ []
abbreviation (*input*) *es-invar* :: *'a es* ⇒ *bool* **where** *es-invar* ≡ λ-. *True*

interpretation *es*: *set-empty es-α es-invar es-empty* **by** *unfold-locales simp-all*
interpretation *es*: *set-memb es-α es-invar es-memb* **by** *unfold-locales simp*
interpretation *es*: *set-isEmpty es-α es-invar es-isEmpty* **by** *unfold-locales simp*
interpretation *es*: *set-to-list es-α es-invar es-to-list* **by** *unfold-locales simp-all*
end
theory *DFS-Impl*
imports
DFS
Empty-Set
../Libs/Collections/Collections
~~/src/HOL/Library/Efficient-Nat
begin

type-synonym (*'a*,*'b*,*'c*) *impl-sws* = (*'a* × *'b* × *'c* × *'c* × *nat* × *'b list* × *'b list list*)

definition *idisc x* ≡ *let* (*S*, *n*, *d*, *f*, *c*, *st*, *wll*) = *x in d*
definition *ifinish x* ≡ *let* (*S*, *n*, *d*, *f*, *c*, *st*, *wll*) = *x in f*
definition *istack x* ≡ *let* (*S*, *n*, *d*, *f*, *c*, *st*, *wll*) = *x in st*
definition *istate x* ≡ *let* (*S*, *n*, *d*, *f*, *c*, *st*, *wll*) = *x in S*
definition *icounter x* ≡ *let* (*S*, *n*, *d*, *f*, *c*, *st*, *wll*) = *x in c*
definition *iwll x* ≡ *let* (*S*, *n*, *d*, *f*, *c*, *st*, *wll*) = *x in wll*
definition *istart x* ≡ *let* (*S*, *n*, *d*, *f*, *c*, *st*, *wll*) = *x in n*

lemma *iaccess-simps* [*simp*]:
idisc (*S*, *n*, *d*, *f*, *c*, *st*, *wll*) = *d*
ifinish (*S*, *n*, *d*, *f*, *c*, *st*, *wll*) = *f*
istack (*S*, *n*, *d*, *f*, *c*, *st*, *wll*) = *st*
istate (*S*, *n*, *d*, *f*, *c*, *st*, *wll*) = *S*

icounter $(S,\ n,\ d,\ f,\ c,\ st,\ wll) = c$
iwll $(S,\ n,\ d,\ f,\ c,\ st,\ wll) = wll$
istart $(S,\ n,\ d,\ f,\ c,\ st,\ wll) = n$
by (*simp-all add*: *idisc-def ifinish-def istack-def istate-def icounter-def iwll-def istart-def*)

record $('S,\ 'n,\ 'm,\ 's)$ *dfs-algorithm-impl* =
dfs-impl-cond :: $'S \Rightarrow$ *bool* — the condition to be satisified by the state S to continue searching from here.
dfs-impl-action :: $'S \Rightarrow ('S,'n,'m)$ *impl-sws* $\Rightarrow 'n \Rightarrow 'S$ — modifies the state for the current node BEFORE visiting the successors.
dfs-impl-post :: $'S \Rightarrow ('S,'n,'m)$ *impl-sws* $\Rightarrow 'n \Rightarrow 'S$ — modifies the state for the current node AFTER having visited the successors (i.e. during backtracking).
dfs-impl-remove :: $'S \Rightarrow ('S,'n,'m)$ *impl-sws* $\Rightarrow 'n \Rightarrow 'S$ — modifies the state if a node has already been visited and is removed from the stack
dfs-impl-start :: $'n \Rightarrow 'S$ — the starting state
dfs-impl-restrict :: $'s$ — the set of restricted nodes
dfs-impl-invar :: $('S,'n,'m)$ *impl-sws* $\Rightarrow$ *bool*

definition *dfs-impl-state-invar* **where**
dfs-impl-state-invar $P \equiv \lambda s.\ P$ (*istate* s)

locale *DFS-Impl'* =
fixes *dfs* :: $('Sa,\ 'n,\ 'morea)$ *dfs-algorithm-invar-scheme*
and *impl-dfs* :: $('S,\ 'n,\ 'm,\ 's,\ 'more)$ *dfs-algorithm-impl-scheme*

locale *sws-impl* = *g*: *finite-digraph V E* +
map: *StdMap mops*
for *mops* :: $('n,\ nat,\ 'm,\ 'X)$ *map-ops-scheme*
and V :: $'n$ *set* **and** E :: $('n \times 'n)$ *set* +
fixes α_σ :: $'S \Rightarrow 'Sa$
begin
definition *impl-sws-α* :: $('S,\ 'n,\ 'm)$ *impl-sws* $\Rightarrow ('Sa,\ 'n)$ *dfs-sws* **where**
impl-sws-α x = (*let* $(S,\ n,\ d,\ f,\ c,\ st,\ wll) = x$ *in dfs-sws.make n st* (*map set wll*) (*map.α d*) (α f) c (α_σ S))

lemma *impl-sws-α-simps* [*simp*]:
stack (*impl-sws-α* $(S,\ n,\ d,\ f,\ c,\ st,\ wll)$) = st
start (*impl-sws-α* $(S,\ n,\ d,\ f,\ c,\ st,\ wll)$) = n
discover (*impl-sws-α* $(S,\ n,\ d,\ f,\ c,\ st,\ wll)$) = *map.α d*
finish (*impl-sws-α* $(S,\ n,\ d,\ f,\ c,\ st,\ wll)$) = *map.α f*
counter (*impl-sws-α* $(S,\ n,\ d,\ f,\ c,\ st,\ wll)$) = c
state (*impl-sws-α* $(S,\ n,\ d,\ f,\ c,\ st,\ wll)$) = α_σ S
wl (*impl-sws-α* $(S,\ n,\ d,\ f,\ c,\ st,\ wll)$) = *map set wll*
by (*simp-all add*: *impl-sws-α-def dfs-sws.defs*)

lemma *impl-sws-α-conv* [*simp*]:
stack (*impl-sws-α* x) = *istack* x
start (*impl-sws-α* x) = *istart* x
discover (*impl-sws-α* x) = *map.α* (*idisc* x)

finish (*impl-sws-α x*) = *map.α* (*ifinish x*)
state (*impl-sws-α x*) = α_σ (*istate x*)
counter (*impl-sws-α x*) = *icounter x*
wl (*impl-sws-α x*) = *map set* (*iwll x*)
by (*cases x*, *simp*)+
end

locale *DFS-Impl* = *DFS-Impl′ dfs impl-dfs* +
sws-impl mops V E α_σ +
set: *set-memb setα setinvar setmemb*
for *dfs* :: (′*Sa*, ′*n*, ′*morea*) *dfs-algorithm-invar-scheme*
and *impl-dfs* :: (′*S*, ′*n*, ′*m*, ′*s*, ′*more*) *dfs-algorithm-impl-scheme*
and *mops* :: (′*n*, *nat*, ′*m*, ′*X*) *map-ops-scheme*
and *setα* :: ′*s* ⇒ ′*n set* **and** *setinvar* :: ′*s* ⇒ *bool* **and** *setmemb* :: ′*n* ⇒ ′*s* ⇒ *bool*
and *V* :: ′*n set* **and** *E* :: (′*n* × ′*n*) *set*
and α_σ :: ′*S* ⇒ ′*Sa* +
fixes γ*succs* :: ′*n* ⇒ ′*n list*
assumes *succs-correct*: *set* (γ*succs y*) = *succs y*
and *succs-distinct*: *distinct* (γ*succs y*)
begin

fun *dfs-step-impl′* :: (′*S*,′*n*,′*m*) *impl-sws* ⇒ (′*S*,′*n*,′*m*) *impl-sws* ⇒ (′*S*,′*n*,′*m*) *impl-sws*
where
dfs-step-impl′ s (*S*, *n*, *d*, *f*, *c*, (*x*#*xs*), ([]#*ys*)) = (*dfs-impl-post impl-dfs S s x*, *n*, *d*, *map.update x c f*, *c* + *1*, *xs*, *ys*)
| *dfs-step-impl′ s* (*S*, *n*, *d*, *f*, *c*, (*x*#*xs*), ((*z*#*zs*)#*ys*)) = (*if setmemb z* (*dfs-impl-restrict impl-dfs*) *then* (*S*, *n*, *d*, *f*, *c*, (*x*#*xs*), (*zs*#*ys*))
else if map.lookup z d ≠ *None then* (*dfs-impl-remove impl-dfs S s z*, *n*, *d*, *f*, *c*, *x* # *xs*, *zs* # *ys*)
else (*dfs-impl-action impl-dfs S s z*, *n*, *map.update z c d*, *f*, *c* + *1*, *z* # *x* # *xs*, (γ*succs z*) # *zs* # *ys*))
| *dfs-step-impl′ s* (*S*, *n*, *d*, *f*, *c*, [], []) = *s*

abbreviation *dfs-step-impl s* ≡ *dfs-step-impl′ s s*

definition *dfs-cond-impl* :: (′*S*, ′*n*, ′*m*) *impl-sws* ⇒ *bool* **where**
dfs-cond-impl s ≡ *istack s* ≠ [] ∧ *iwll s* ≠ [] ∧ *dfs-impl-cond impl-dfs* (*istate s*)

definition *dfs-start-impl* :: ′*n* ⇒ (′*S*, ′*n*, ′*m*) *impl-sws* **where**
dfs-start-impl x = (*dfs-impl-start impl-dfs x*, *x*, *map.sng x 0*, *map.empty* (), (*1*::*nat*), [*x*], [γ*succs x*])

definition *impl-sws-invar* :: (′*S*, ′*n*, ′*m*) *impl-sws* ⇒ *bool* **where**
impl-sws-invar s ≡
dfs-constructable dfs (*impl-sws-α s*) ∧
map.invar (*ifinish s*) ∧ *map.invar* (*idisc s*) ∧
(∀ *w* ∈ *set* (*iwll s*). *distinct w* ∧ *set w* ⊆ *V*)

lemma *impl-sws-invarI*:
⟦ *dfs-constructable dfs* (*impl-sws-α s*);
map.invar (*ifinish s*); *map.invar* (*idisc s*);
⋀*w*. *w* ∈ *set* (*iwll s*) ⟹ *distinct w* ∧ *set w* ⊆ *V* ⟧ ⟹ *impl-sws-invar s*
unfolding *impl-sws-invar-def*
by *metis*

lemma *impl-sws-invarE*:
⟦ *impl-sws-invar s*;
⟦ *dfs-constructable dfs* (*impl-sws-α s*);
map.invar (*ifinish s*); *map.invar* (*idisc s*);
⋀*w*. *w* ∈ *set* (*iwll s*) ⟹ *distinct w* ∧ *set w* ⊆ *V* ⟧ ⟹ *P* ⟧
⟹ *P*
unfolding *impl-sws-invar-def*
by *metis*

lemma *impl-sws-invar-constructable*:
impl-sws-invar s ⟹ *dfs-constructable dfs* (*impl-sws-α s*)
by (*metis impl-sws-invarE*)

lemmas *impl-sws-invar-constructableE*[*elim*] = *impl-sws-invar-constructable*[*elim-format*]

lemma *impl-sws-invar-mapinvars*:
impl-sws-invar s ⟹ *map.invar* (*idisc s*)
impl-sws-invar s ⟹ *map.invar* (*ifinish s*)
unfolding *impl-sws-invar-def*
by *metis*+

lemma *impl-sws-invar-dfs-invar*:
dfs-preserves-invar dfs ⟹ *impl-sws-invar s* ⟹ *dfs-invar dfs* (*impl-sws-α s*)
by *auto*

definition *impl-preserves-invar* **where**
impl-preserves-invar ≡ (∀ *x* ∈ *V*. *x* ∉ *setα* (*dfs-impl-restrict impl-dfs*) ∧ *impl-sws-invar*
(*dfs-start-impl x*) ⟶
dfs-impl-invar impl-dfs (*dfs-start-impl x*)) ∧
(∀ *s s'*. *impl-sws-invar s* ∧ *dfs-cond-impl s* ∧ *s'* = *dfs-step-impl*
s ∧ *dfs-impl-invar impl-dfs s* ⟶
dfs-impl-invar impl-dfs s')

lemma *impl-preserves-invar-start*:
impl-preserves-invar ⟹ *x* ∈ *V* ⟹ *x* ∉ *setα* (*dfs-impl-restrict impl-dfs*) ⟹
impl-sws-invar (*dfs-start-impl x*) ⟹ *dfs-impl-invar impl-dfs* (*dfs-start-impl x*)
by (*simp add*: *impl-preserves-invar-def*)

lemma *impl-preserves-invar-step*:
impl-preserves-invar ⟹ *impl-sws-invar s* ⟹ *dfs-cond-impl s* ⟹ *dfs-impl-invar*
impl-dfs s ⟹ *s'* = *dfs-step-impl s* ⟹ *dfs-impl-invar impl-dfs s'*
unfolding *impl-preserves-invar-def*

by *blast*

lemma *impl-preserves-invarI*:
⟦ ⋀*v*. ⟦*v* ∈ *V*; *v* ∉ *setα* (*dfs-impl-restrict impl-dfs*); *impl-sws-invar* (*dfs-start-impl v*)⟧ ⟹ *dfs-impl-invar impl-dfs* (*dfs-start-impl v*);
⋀*s s'*. ⟦ *impl-sws-invar s*; *dfs-cond-impl s*; *dfs-impl-invar impl-dfs s*; *s'* = *dfs-step-impl s* ⟧ ⟹ *dfs-impl-invar impl-dfs s'*
⟧ ⟹ *impl-preserves-invar*
unfolding *impl-preserves-invar-def*
by *blast*

lemma *state-impl-preserves-invarI*:
assumes *SI*: *dfs-impl-invar impl-dfs* = *dfs-impl-state-invar P*
and *start*: ⋀*x*. ⟦ *x* ∈ *V*; *x* ∉ *setα* (*dfs-impl-restrict impl-dfs*) ⟧ ⟹ *P* (*dfs-impl-start impl-dfs x*)
and *post*: ⋀*s x*. ⟦*x* ∈ *set* (*istack s*); *P* (*istate s*); *impl-sws-invar s*; *dfs-cond-impl s*⟧ ⟹ *P* (*dfs-impl-post impl-dfs* (*istate s*) *s x*)
and *remove*: ⋀*s x*. ⟦*x* ∈ *set* (*hd* (*iwll s*)); *map.lookup x* (*idisc s*) ≠ *None*; *P* (*istate s*); *impl-sws-invar s*; *dfs-cond-impl s*⟧ ⟹ *P* (*dfs-impl-remove impl-dfs* (*istate s*) *s x*)
and *action*: ⋀*s x*. ⟦*x* ∈ *set* (*hd* (*iwll s*)); *map.lookup x* (*idisc s*) = *None*; *P* (*istate s*); *impl-sws-invar s*; *dfs-cond-impl s*⟧ ⟹ *P* (*dfs-impl-action impl-dfs* (*istate s*) *s x*)
shows *impl-preserves-invar*
proof (*rule impl-preserves-invarI*)
case (*goal1 v*) **with** *SI start* **show** *?case* **unfolding** *dfs-impl-state-invar-def dfs-start-impl-def* **by** *simp*
next
note *SIS*[*simp*] = *SI dfs-impl-state-invar-def*
case (*goal2 s s'*) **hence** *inv*: *P* (*istate s*) **and** *length* (*istack s*) = *length* (*iwll s*)
using *length-wl-eq-stack*[*OF impl-sws-invar-constructable*] **by** *simp-all*
with *goal2*(*4*,*1*−*3*) **show** *?case*
proof (*induction s rule*: *dfs-step-impl'.induct*)
case (*goal1 s'*) **hence** *x* ∈ *set* (*x*#*xs*) **by** *simp*
with *post*[*OF - goal1*(*5*)] *goal1* **show** *?case* **by** *simp*
next
case (*goal2 s'*) **hence** *z* ∈ *set* (*hd* ((*z*#*zs*)#*ys*)) **by** *auto*
with *goal2 remove*[*OF - - goal2*(*5*)] *action*[*OF - - goal2*(*5*)] **show** *?case* **by** *simp*
qed *simp-all*
qed
end

locale *DFS-Impl-correct* = *DFS-Impl dfs impl-dfs mops setα setinvar setmemb V E* α_σ *γsucc*s
for *dfs* :: (*'Sa*, *'n*, *'morea*) *dfs-algorithm-invar-scheme*
and *impl-dfs* :: (*'S*, *'n*, *'m*, *'s*, *'more*) *dfs-algorithm-impl-scheme*
and *mops* :: (*'n*, *nat*, *'m*, *'X*) *map-ops-scheme*
and *setα* :: *'s* ⇒ *'n set* **and** *setinvar* :: *'s* ⇒ *bool* **and** *setmemb* :: *'n* ⇒ *'s* ⇒

bool
and $V :: 'n\ set$ **and** $E :: ('n \times 'n)\ set$
and $\alpha_\sigma :: 'S \Rightarrow 'Sa$
and $\gamma succs :: 'n \Rightarrow 'n\ list+$
assumes *action-correct*: $[\![x \in set\ (hd\ (iwll\ s))$; *map.lookup* x $(idisc\ s) = None$; *dfs-impl-invar impl-dfs s*; *impl-sws-invar s*; *dfs-cond-impl s* $]\!] \Longrightarrow \alpha_\sigma$ $(dfs\text{-}impl\text{-}action$ *impl-dfs* $(istate\ s)\ s\ x) = dfs\text{-}action\ dfs\ (\alpha_\sigma\ (istate\ s))\ (impl\text{-}sws\text{-}\alpha\ s)\ x$
and *post-correct*: $[\![x \in set\ (istack\ s)$; *dfs-impl-invar impl-dfs s*; *impl-sws-invar s*; *dfs-cond-impl s* $]\!] \Longrightarrow \alpha_\sigma\ (dfs\text{-}impl\text{-}post\ impl\text{-}dfs\ (istate\ s)\ s\ x) = dfs\text{-}post\ dfs\ (\alpha_\sigma$ $(istate\ s))\ (impl\text{-}sws\text{-}\alpha\ s)\ x$
and *remove-correct*: $[\![x \in set\ (hd\ (iwll\ s))$; *map.lookup* x $(idisc\ s) \neq None$; *dfs-impl-invar impl-dfs s*; *impl-sws-invar s*; *dfs-cond-impl s* $]\!] \Longrightarrow \alpha_\sigma\ (dfs\text{-}impl\text{-}remove$ *impl-dfs* $(istate\ s)\ s\ x) = dfs\text{-}remove\ dfs\ (\alpha_\sigma\ (istate\ s))\ (impl\text{-}sws\text{-}\alpha\ s)\ x$
and *start-correct*: $[\![x \in V; x \notin set\alpha\ (dfs\text{-}impl\text{-}restrict\ impl\text{-}dfs)]\!] \Longrightarrow \alpha_\sigma\ (dfs\text{-}impl\text{-}start$ *impl-dfs* $x) = dfs\text{-}start\ dfs\ x$
and *cond-correct*: *dfs-impl-cond impl-dfs* $S = dfs\text{-}cond\ dfs\ (\alpha_\sigma\ S)$
and *restr-correct*: $set\alpha\ (dfs\text{-}impl\text{-}restrict\ impl\text{-}dfs) = dfs\text{-}restrict\ dfs$
and *restr-invar*: *setinvar* $(dfs\text{-}impl\text{-}restrict\ impl\text{-}dfs)$
and *impl-preserves-invar*: *impl-preserves-invar*
begin

lemma *dfs-cond-impl-correct*:
dfs-cond-impl s $= dfs\text{-}cond\text{-}compl\ dfs\ (impl\text{-}sws\text{-}\alpha\ s)$
unfolding *dfs-cond-impl-def dfs-cond-compl-def*
by (*simp add*: *cond-correct*)

lemma *dfs-cond-impl-conv*:
dfs-cond-impl $(S, n, d, f, c, st, wll) \equiv st \neq [] \wedge wll \neq [] \wedge dfs\text{-}impl\text{-}cond\ impl\text{-}dfs$ S
unfolding *dfs-cond-impl-def istate-def istack-def iwll-def*
by *simp*

lemma *dfs-start-impl-correct*:
$x \in V \Longrightarrow x \notin set\alpha\ (dfs\text{-}impl\text{-}restrict\ impl\text{-}dfs) \Longrightarrow impl\text{-}sws\text{-}\alpha\ (dfs\text{-}start\text{-}impl$ $x) = dfs\text{-}constr\text{-}start\ dfs\ x$
unfolding *dfs-start-impl-def dfs-constr-start-def*
by (*simp add*: *start-correct map.correct succs-correct impl-preserves-invar-start*[*OF impl-preserves-invar*])

lemma *dfs-start-impl-constructable*:
assumes $x \in V$
and $x \notin set\alpha\ (dfs\text{-}impl\text{-}restrict\ impl\text{-}dfs)$
shows *dfs-constructable dfs* $(impl\text{-}sws\text{-}\alpha\ (dfs\text{-}start\text{-}impl\ x))$
proof −
from *assms*(*2*) **have** $x \notin dfs\text{-}restrict\ dfs$ **using** *restr-correct* **by** *simp*
with *assms dfs-start-impl-correct dfs-constructable.start* **show** *?thesis* **by** *simp*
qed

lemma *dfs-step-impl-correct*:

assumes *invars*: *impl-sws-invar s dfs-impl-invar impl-dfs s dfs-cond-impl s*
and $s' = dfs\text{-}step\text{-}impl\ s$
shows $impl\text{-}sws\text{-}\alpha\ s' \in dfs\text{-}step\ dfs\ (impl\text{-}sws\text{-}\alpha\ s)$
proof −
from *invars length-wl-eq-stack* **have** $length\ (istack\ s) = length\ (iwll\ s)$ **by** *force*
moreover from *invars* **have** $istack\ s \neq []$ **unfolding** *dfs-cond-impl-def* **by** *simp*
moreover note *assms*
ultimately show *?thesis*
proof (*induction* (*s*) *rule*: *dfs-step-impl'.induct*)
case (*goal1 s'*) **let** *?s* = (*S*, *n*, *d*, *f*, *c*, *x* # *xs*, [] # *ys*) **let** *?R* = *impl-sws-α ?s*
from *goal1 invars* **have** *inf* :*invar f* **and** *inv*: *dfs-impl-invar impl-dfs ?s* **unfolding** *impl-sws-invar-def* **by** *auto*

from *goal1* **have** *dfs-step dfs ?R* = {*dfs-sws.make n xs* (*map set ys*) (*map.α d*) ((*map.α f*)(*x* ↦ *c*)) (*Suc c*) (*dfs-post dfs* (α_σ *S*) *?R x*)}
using *dfs-step-simps*(*2*)[*of ?R x xs dfs*] **by** (*simp add*: *dfs-sws.defs impl-sws-α-def*)
also from *goal1 post-correct*[*OF - inv*] *update-correct*[*OF inf*] **have** ... = {*dfs-sws.make n xs* (*map set ys*) (*α d*) (*map.α* (*update x c f*)) (*Suc c*) (α_σ (*dfs-impl-post impl-dfs S ?s x*))}
by (*simp add*: *dfs-sws.defs*)
also from *goal1* **have** ... = { *impl-sws-α* (*dfs-step-impl ?s*) } **by** (*simp add*: *impl-sws-α-def*)
finally show *?case* **by** (*simp add*: *goal1*)
next
case (*goal2 s'*) **let** *?s* = (*S*, *n*, *d*, *f*, *c*, *x* # *xs*, (*z* # *zs*) # *ys*) **let** *?R* = *impl-sws-α ?s*
from *goal2* **have** *ind*: *invar d* **and** *inv*: *dfs-impl-invar impl-dfs ?s* **unfolding** *impl-sws-invar-def* **by** *auto*
from *goal2*(*3*) *wl-subset-succs*[**where** *n*= *0*] *succs-correct* **have** *z-in*: $z \in set\ (\gamma succs\ x)$ **by** *fastforce*

from *goal2* **have** $\bigwedge w.\ w \in set\ (iwll\ ?s) \Longrightarrow distinct\ w\ (z\#zs) \in set\ (iwll\ ?s)$ **unfolding** *impl-sws-invar-def* **by** *force*+
hence *z-dist*: *distinct* (*z*#*zs*) **.**

show *?case*
proof (*cases setmemb z* (*dfs-impl-restrict impl-dfs*))
case *True* **with** *restr-correct restr-invar set.memb-correct* **have** $z \in dfs\text{-}restrict\ dfs$ **by** *blast*
with *goal2 z-dist* **have** *dfs-sws.make n* (*x*#*xs*) (*map set* (*zs*#*ys*)) (*α d*) (*α f*) *c* (α_σ *S*) ∈ *dfs-step dfs ?R*
using *dfs-step-simps*(*3*)[*of ?R x xs dfs*]
apply *simp*
apply (*rule-tac x*=*z* **in** *exI*)
by *simp*
with *goal2 True* **show** *?thesis* **by** (*simp add*: *impl-sws-α-def*)
next
case *False* **with** *restr-correct restr-invar set.memb-correct* **have** *z-not-R*: $z \notin$

dfs-restrict dfs **by** *blast*
show *?thesis*
proof (*cases lookup z d*)
case *None* **with** *z-in* **have** *set* (*γsuccs x*) − {*v*. *lookup v d* ≠ *None*} ≠ {}
by *auto*
with *ind None succs-correct z-in* **have** *z* ∉ *dom* (*α d*) **and** *succs x* − *dom* (*α d*) ≠ {} **and** *z* ∈ *succs x* **by** (*auto simp add*: *correct*)
with *goal2 z-not-R None z-dist* **have** *dfs-sws.make n* (*z*#*x*#*xs*) (*map set* (*γsuccs z*#*zs*#*ys*)) ((*α d*)(*z* ↦ *c*)) (*α f*) (*Suc c*) (*dfs-action dfs* (*α*$_\sigma$ *S*) *?R z*) ∈ *dfs-step dfs ?R*
using *dfs-step-simps*(*3*)[*of ?R x xs dfs*]
apply *simp*
apply (*rule-tac x*=*z* **in** *exI*)
by (*auto simp add*: *succs-correct*)
with *goal2 None action-correct*[*OF - - inv*] *update-correct*[*OF ind*] **have** *dfs-sws.make n* (*z*#*x*#*xs*) (*map set* (*γsuccs z*#*zs*#*ys*)) (*α* (*map.update z c d*)) (*α f*) (*Suc c*) (*α*$_\sigma$ (*dfs-impl-action impl-dfs S ?s z*)) ∈ *dfs-step dfs ?R*
by (*simp add*: *dfs-sws.defs*)
with *None False goal2* **show** *?thesis* **by** (*simp add*: *impl-sws-α-def*)
next
case (*Some e*) **with** *ind* **have** *z* ∈ *dom* (*α d*) **by** (*auto simp add*: *correct*)
with *goal2 Some z-not-R z-dist* **have** *dfs-sws.make n* (*x*#*xs*) (*map set* (*zs*#*ys*)) (*α d*) (*α f*) *c* (*dfs-remove dfs* (*α*$_\sigma$ *S*) *?R z*) ∈ *dfs-step dfs ?R*
using *dfs-step-simps*(*3*)[*of ?R x xs dfs*]
apply *simp*
apply (*rule-tac x*=*z* **in** *exI*)
by *simp*
with *remove-correct*[*OF - - inv*] *Some False goal2* **show** *?thesis* **by** (*auto simp add*: *impl-sws-α-def*)
qed
qed
qed *simp-all*
qed

lemma *dfs-step-impl-dfs-next*:
assumes *invar*: *impl-sws-invar s dfs-impl-invar impl-dfs s*
and *cond*: *dfs-cond-impl s*
shows *dfs-next dfs* (*impl-sws-α s*) (*impl-sws-α* (*dfs-step-impl s*))
proof −
from *cond dfs-cond-impl-correct* **have** *dfs-cond-compl dfs* (*impl-sws-α s*) **by** *simp*
moreover from *cond invar* **have** *impl-sws-α* (*dfs-step-impl s*) ∈ *dfs-step dfs* (*impl-sws-α s*) **by** (*simp add*: *dfs-step-impl-correct*)
ultimately show *?thesis* **unfolding** *dfs-next-def* **by** *simp*
qed

lemma *dfs-impl-constructable*:
assumes *impl-sws-invar s dfs-impl-invar impl-dfs s*
and *constr*: *dfs-constructable dfs* (*impl-sws-α s*)
and *dfs-cond-impl s*

shows *dfs-constructable dfs* (*impl-sws-α* (*dfs-step-impl s*))
proof −
from *assms* **have** *dfs-next dfs* (*impl-sws-α s*) (*impl-sws-α* (*dfs-step-impl s*)) **using** *dfs-step-impl-dfs-next* **by** *blast*
with *constr* **show** *?thesis* **using** *dfs-constructable.step* **by** *metis*
qed

lemma *dfs-step-impl-invars*:
assumes *invars*: *impl-sws-invar s dfs-impl-invar impl-dfs s dfs-cond-impl s*
and *step*: $s' = dfs\text{-}step\text{-}impl\ s$
shows *impl-sws-invar s′*
proof −
from *invars step dfs-impl-constructable* **have** *dfs-constructable dfs* (*impl-sws-α s′*) **by** *blast*
moreover from *invars length-wl-eq-stack* **have** *length* (*istack s*) = *length* (*iwll s*) **by** *force*
moreover from *invars* **have** *istack s* ≠ [] **unfolding** *dfs-cond-impl-def* **by** *simp*
moreover note *assms*
ultimately show *?thesis*
proof (*induction s rule*: *dfs-step-impl′.induct*)
case (*goal1 s′*) **with** *goal1*[*unfolded impl-sws-invar-def*] **show** *?case* **by** (*intro impl-sws-invarI*) (*simp-all add*: *map.correct*)
next
case (*goal2 s′*)
note *g2* = *goal2*[*unfolded impl-sws-invar-def*]

show *?case*
proof (*cases setmemb z* (*dfs-impl-restrict impl-dfs*) ∨ *lookup z d* ≠ *None*)
case *True* **with** *g2* **show** *?thesis*
apply (*intro impl-sws-invarI*)
apply (*case-tac* [!] *setmemb z* (*dfs-impl-restrict impl-dfs*))
apply *auto*
done
next
case *False* **hence** *membFalse*: ¬ *setmemb z* (*dfs-impl-restrict impl-dfs*) *map.lookup z d* = *None* **by** *simp-all*
with *g2* **show** *?thesis*
proof (*intro impl-sws-invarI*)
from *g2* **have** ∗: ⋀ *w*. *w* ∈ *set* ((*z*#*zs*)#*ys*) ⟹ *distinct w* ∧ *set w* ⊆ *V* **by** *auto*
moreover
from ∗[*of z*#*zs*] *distinct-tl* **have** *distinct zs* **by** *simp*
moreover from *succs-correct* **have** *set* (*γsuccs z*) ⊆ *V* **using** *succs-in-V* **by** *auto*
ultimately have ⋀*w*. *w* ∈ *set* (*γsuccs z*#*zs*#*ys*) ⟹ *distinct w* ∧ *set w* ⊆ *V* **by** (*metis succs-distinct ListMem-iff elem mem-def predicate1D set-ConsD set-subset-Cons subset-trans*)
thus ⋀*w*. *w* ∈ *set* (*iwll s′*) ⟹ *distinct w* ∧ *set w* ⊆ *V* **using** *g2 membFalse* **by** *auto*

```
    qed (simp-all add: map.correct)
   qed
  qed simp-all
qed

definition dfs-fun-impl :: 'n ⇒ (('S,'n,'m) impl-sws) nres where
  dfs-fun-impl x ≡ WHILE_T dfs-cond-impl (λs. RETURN (dfs-step-impl s)) (dfs-start-impl
x)

definition dfs-impl-build-rel where
  dfs-impl-build-rel ≡ br impl-sws-α (λs. impl-sws-invar s ∧ dfs-impl-invar impl-dfs
s)

lemma dfs-fun-impl-refine:
  assumes x ∈ V x ∉ dfs-restrict dfs
  shows dfs-fun-impl x ≤ ⇓ dfs-impl-build-rel (dfs-fun dfs x)
unfolding dfs-fun-impl-def dfs-fun-def dfs-impl-build-rel-def
apply (refine-rcg)
  apply (simp add: dfs-start-impl-correct[OF assms(1)] restr-correct assms)
  apply (rule context-conjI)
   apply (intro impl-sws-invarI)
    apply (simp add: dfs-start-impl-constructable restr-correct assms)
   apply (simp-all add: dfs-start-impl-def correct succs-distinct assms succs-correct
succs-in-V)[3]
    apply (simp add: impl-preserves-invar-start[OF impl-preserves-invar] assms
restr-correct)
  apply (simp add: dfs-cond-impl-correct)
  apply (rule SPEC-refine-sv)
   apply (rule br-single-valued)
   apply (simp add: dfs-step-impl-dfs-next)
   apply (rule conjI)
    apply (blast dest: dfs-step-impl-invars)
    apply (blast dest: impl-preserves-invar-step[OF impl-preserves-invar])
done

theorem dfs-fun-impl-correct:
  assumes dfs-preserves-invar dfs x ∈ V x ∉ dfs-restrict dfs
  shows dfs-fun-impl x ≤ ⇓ dfs-impl-build-rel (SPEC (λs. s ∈ dfs-constr-from dfs
x ∧ dfs-finished dfs s))
proof −
  note dfs-fun-impl-refine[OF assms(2−)]
  also note dfs-fun-correct[OF assms]
  finally show ?thesis .
qed

lemma dfs-fun-impl-nofail[refine-pw-simps]:
  assumes dfs-preserves-invar dfs x ∈ V x ∉ dfs-restrict dfs
  shows nofail (dfs-fun-impl x)
proof −
```

note *dfs-fun-nofail*[*OF assms*]
also note *pw-conc-nofail*[*symmetric*]
finally show *?thesis* **by** (*blast intro*: *pwD dfs-fun-impl-refine*[*OF assms*(2−)])
qed

schematic-lemma *dfs-code-aux*:
RETURN ?dfs-code ≤ *dfs-fun-impl x*
unfolding *dfs-fun-impl-def*
by *refine-transfer*

definition *dfs-code* :: *'n* ⇒ (*'S*,*'n*,*'m*) *impl-sws* **where**
dfs-code x ≡ *while dfs-cond-impl dfs-step-impl* (*dfs-start-impl x*)

lemmas *dfs-code-refine* = *dfs-code-aux*[*folded dfs-code-def*]

theorem *dfs-code-correct*:
⟦ *dfs-preserves-invar dfs*; *x* ∈ *V*; *x* ∉ *dfs-restrict dfs*; *s* = *dfs-code x* ⟧ ⟹ *impl-sws-α s* ∈ *dfs-constr-from dfs x* ∧ *dfs-finished dfs* (*impl-sws-α s*) ∧ *impl-sws-invar s* ∧ *dfs-impl-invar impl-dfs s*
using *order-trans*[*OF dfs-code-refine dfs-fun-impl-correct*, *of x*]
unfolding *dfs-impl-build-rel-def*
by (*auto elim*!: *RETURN-ref-SPECD*)

corollary *dfs-code-constructable*:
⟦ *dfs-preserves-invar dfs*; *x* ∈ *V*; *x* ∉ *dfs-restrict dfs* ⟧ ⟹ *impl-sws-α* (*dfs-code x*) ∈ *dfs-constr-from dfs x*
by (*metis dfs-code-correct*)

corollary *dfs-code-preserves-invar*:
⟦ *dfs-preserves-invar dfs*; *x* ∈ *V*; *x* ∉ *dfs-restrict dfs* ⟧ ⟹ *impl-sws-invar* (*dfs-code x*)
by (*metis dfs-code-correct*)

corollary *dfs-code-preserves-dfs-invar*:
⟦ *dfs-preserves-invar dfs*; *x* ∈ *V*; *x* ∉ *dfs-restrict dfs* ⟧ ⟹ *dfs-impl-invar impl-dfs* (*dfs-code x*)
by (*metis dfs-code-correct*)

corollary *dfs-code-finished*:
⟦ *dfs-preserves-invar dfs*; *x* ∈ *V*; *x* ∉ *dfs-restrict dfs* ⟧ ⟹ *dfs-finished dfs* (*impl-sws-α* (*dfs-code x*))
by (*metis dfs-code-correct*)

— Helper function to return the list of visited nodes.
definition *to-visited* :: (*'S*,*'n*,*'m*) *impl-sws* ⇒ *'n list* **where**
to-visited s = *map fst* (*to-list* (*idisc s*))

lemma *to-visited-correct*:
assumes *invar*: *invar* (*idisc s*)

shows *set* (*to-visited s*) = *discovered* (*impl-sws-α s*)
proof −
have *map-of* (*to-list* (*idisc s*)) = *discover* (*impl-sws-α s*) **using** *invar* **by** (*simp add*: *correct*)
hence *dom* (*map-of* (*to-list* (*idisc s*))) = *discovered* (*impl-sws-α s*) **by** *simp*
with *dom-map-of-conv-image-fst*[*of to-list* (*idisc s*)] **show** *?thesis* **unfolding** *to-visited-def* **by** *simp*
qed

— Helper function returning a tuple (node, δ node, φ node)
definition *to-fd* :: ('*S*, '*n*, '*m*) *impl-sws* ⇒ ('*n* × *nat* × *nat*) *list* **where**
to-fd s = *map* (λ *n*. *let d* = *the* (*lookup n* (*idisc s*)) *in*
let f = *the* (*lookup n* (*ifinish s*)) *in*
(*n*,*d*,*f*)) (*to-visited s*)

lemma *to-fd-correct*:
assumes *invars*: *invar* (*ifinish s*) *invar* (*idisc s*)
and *elem*: (*n*, *d*, *f*) ∈ *set* (*to-fd s*)
shows φ (*impl-sws-α s*) *n* = *f* ∧ δ (*impl-sws-α s*) *n* = *d*
using *assms* **unfolding** *to-fd-def*
by (*auto simp add*: *to-visited-correct correct*)
end

definition *simple-impl-dfs* :: (*unit*, '*n*, '*m*, '*n es*) *dfs-algorithm-impl* **where**
simple-impl-dfs = (| *dfs-impl-cond* = λ-. *True*,
dfs-impl-action = λ- - -. (),
dfs-impl-post = λ- - -. (),
dfs-impl-remove = λ- - -. (),
dfs-impl-start = λ-. (),
dfs-impl-restrict = *es-empty* (),
dfs-impl-invar = λ-. *True*|)

lemma *simple-impl-dfs-simps* [*simp*]:
dfs-impl-cond simple-impl-dfs S
dfs-impl-action simple-impl-dfs S s x = ()
dfs-impl-post simple-impl-dfs S s x = ()
dfs-impl-remove simple-impl-dfs S s x = ()
dfs-impl-start simple-impl-dfs x = ()
dfs-impl-restrict simple-impl-dfs = ()
dfs-impl-invar simple-impl-dfs s
by (*simp-all add*: *simple-impl-dfs-def*)

locale *SimpleDFS-Impl* = *DFS-Impl simple-dfs simple-impl-dfs - es-α es-invar es-memb - - id*
begin

lemma *simple-preserves-invar*:
impl-preserves-invar
by (*intro impl-preserves-invarI*) *simp-all*

lemma *dfs-cond-impl-conv-simple*:
dfs-cond-impl $(S,\ n,\ d,\ f,\ c,\ st,\ wll) \equiv st \neq [] \wedge wll \neq []$
unfolding *dfs-cond-impl-def*
by *simp*

lemmas *dfs-start-impl-conv-simple* = *dfs-start-impl-def*[*unfolded simple-impl-dfs-simps*]
lemmas *dfs-step-impl-conv-simple* = *dfs-step-impl'.simps*[*unfolded simple-impl-dfs-simps*]
end

sublocale *SimpleDFS-Impl* $\subseteq$ *DFS-Impl-correct simple-dfs simple-impl-dfs - es-*α *es-invar es-memb - - id*
by *unfold-locales* (*simp-all add*: *simple-preserves-invar*)

end

theory *Nested-DFS-Impl*
imports *Nested-DFS DFS-Impl*
begin

locale *SubDFS-Impl-def* = *set*: *StdSet sops* +
map: *StdMap mops* +
finite-digraph V E
for *sops* :: $('n,\ 's,\ 'Y)$ *set-ops-scheme*
and *mops* :: $('n,\ nat,\ 'm,\ 'X)$ *map-ops-scheme*
and $V :: 'n$ *set* **and** $E :: ('n \times 'n)$ *set* +
fixes *sup-st* :: $'n \Rightarrow$ *bool* **and** *restr* :: $'s$
assumes *rs-invar* [*simp*]: *set.invar restr*
begin

fun *check-cycle-impl* **where**
check-cycle-impl True - - = True
| *check-cycle-impl False - e = sup-st e* — if e is on the parent (=super) stack, we have a cycle

definition *sub-impl-dfs-invar* **where**
sub-impl-dfs-invar s $\equiv$ *set.invar* (*snd* (*istate s*)) $\wedge$ *set.*α (*snd* (*istate s*)) = *set.*α *restr* $\cup$ *dom* (*map.*α (*ifinish s*))

definition *sub-impl-dfs* :: $(bool \times 's,\ 'n,\ 'm,\ 's)$ *dfs-algorithm-impl* **where**
sub-impl-dfs = (| *dfs-impl-cond* = *Not* $\circ$ *fst*,
dfs-impl-action = $\lambda(f,S)\ s\ x.\ (check\text{-}cycle\text{-}impl\ f\ s\ x,\ S)$,
dfs-impl-post = $\lambda(f,S)$ - $x.\ (f,\ set.ins\text{-}dj\ x\ S)$,
dfs-impl-remove = $\lambda(f,S)\ s\ x.\ (x = istart\ s \wedge sup\text{-}st\ x,\ S)$,
dfs-impl-start = λ-. (*False*, *restr*),
dfs-impl-restrict = *restr*,
dfs-impl-invar = *sub-impl-dfs-invar* |)

lemma *sub-impl-dfs-simps*[*simp*]:

dfs-impl-cond sub-impl-dfs S $\longleftrightarrow \neg$ (*fst S*)
dfs-impl-action sub-impl-dfs S s x = (*check-cycle-impl* (*fst S*) *s x*, *snd S*)
dfs-impl-post sub-impl-dfs S s x = (*fst S*, *set.ins-dj x* (*snd S*))
dfs-impl-remove sub-impl-dfs S s x = (*x* = *istart s* $\wedge$ *sup-st x*, *snd S*)
dfs-impl-start sub-impl-dfs x = (*False*, *restr*)
dfs-impl-restrict sub-impl-dfs = *restr*
dfs-impl-invar sub-impl-dfs = *sub-impl-dfs-invar*
unfolding *sub-impl-dfs-def*
by (*simp-all split*: *prod.split*)

end

locale *SubDFS-Impl* = *SubDFS-Impl-def sops mops V E ss rs* + *DFS-Impl sub-dfs* {*x*. *ss x*} (*set.α rs*) *sub-impl-dfs mops set.α set.invar set.memb V E fst*
for *sops* :: ('*n*, '*s*, '*Y*) *set-ops-scheme*
and *mops* :: ('*n*, *nat*, '*m*, '*X*) *map-ops-scheme*
and *V* :: '*n set* **and** *E* :: ('*n* × '*n*) *set*
and *ss* :: '*n* ⇒ *bool* **and** *rs* :: '*s*
begin

lemma *check-cycle-impl-correct*:
check-cycle-impl S s e $\longleftrightarrow$ *check-cycle* {*x*. *ss x*} *S s' e*
by (*cases S*) *simp-all*

lemma *sub-impl-preserves-invar*:
impl-preserves-invar
proof (*rule impl-preserves-invarI*)
case *goal1* **thus** *?case* **by** (*simp add*: *sub-impl-dfs-invar-def dfs-start-impl-def map.correct*)
next
case *goal2* **hence** *length* (*istack s*) = *length* (*iwll s*) **using** *length-wl-eq-stack*[*OF impl-sws-invar-constructable*] **by** *simp-all*
with *goal2* **show** *?case*
proof (*induction s rule*: *dfs-step-impl'.induct*)
case (*goal1 s'*) **let** *?s* = (*S*, *n*, *d*, *f*, *c*, *x* # *xs*, [] # *ys*) **let** *?αs* = *impl-sws-α ?s*

from *goal1* **have** *constr*: *dfs-constructable* (*sub-dfs* {*x*. *ss x*} (*set.α rs*)) *?αs* **using** *impl-sws-invar-constructable* **by** *simp*

from *stack-implies-not-finished*[*OF this*] *stack-not-restricted*[*OF this*] **have** *x-not*: *x* ∉ *dom* (*map.α f*) *x* ∉ *set.α rs* **by** (*simp-all add*: *set.correct*)

from *goal1* **have** *ifinish s'* = *map.update x c f* **and** *map.invar f* **using** *impl-sws-invar-mapinvars*[**where** *s*=*?s*] **by** *simp-all*
hence *map.α* (*ifinish s'*) = (*map.α f*)(*x* ↦ *c*) **by** (*simp add*: *map.correct*)
hence *dom* (*map.α* (*ifinish s'*)) = *insert x* (*dom* (*map.α f*)) **by** *simp*

moreover from *goal1* **have** *set.α (snd S) = set.α rs ∪ dom (map.α f)* **and** *S-inv*: *set.invar (snd S)* **by** (*simp-all add*: *sub-impl-dfs-invar-def*)
moreover with *x-not* **have** *x-not-S*: *x ∉ set.α (snd S)* **by** *simp*

from *goal1* **have** *snd (istate s') = set.ins-dj x (snd S)* **by** *simp*
with *S-inv x-not-S* **have** *set.invar (snd (istate s'))* **and** *set.α (snd (istate s'))* *= insert x (set.α (snd S))* **by** (*simp-all add*: *set.correct*)

ultimately show *?case* **by** (*simp add*: *sub-impl-dfs-invar-def*)
qed (*simp-all add*: *sub-impl-dfs-invar-def*)
qed

end

sublocale *SubDFS-Impl ⊆ DFS-Impl-correct sub-dfs {x. ss x} (set.α rs) sub-impl-dfs mops set.α set.invar set.memb V E fst*
by *unfold-locales* (*simp-all add*: *check-cycle-impl-correct sub-impl-preserves-invar*)

context *SubDFS-Impl*
begin

abbreviation *sub-dfs-fun-impl ≡ dfs-fun-impl*
abbreviation *sub-dfs-code ≡ dfs-code*
abbreviation *sub-build-rel ≡ dfs-impl-build-rel*

theorem *sub-dfs-fun-impl-correct*:
assumes *x ∈ V* **and** *x ∉ set.α rs*
shows *sub-dfs-fun-impl x ≤ ⇓ sub-build-rel (SPEC (λs. state s ⟷ sub-cycle {x. ss x} (set.α rs) x))*
proof −
from *assms(2)* **have** *x ∉ dfs-restrict (sub-dfs {x. ss x} (set.α rs))* **by** *simp*
note *dfs-fun-impl-refine*[*OF assms(1) this*]
also note *dfs-fun-sub-dfs-correct*[*OF assms*, *of* {*x. ss x*}]
finally show *?thesis* **.**
qed

lemmas *sub-dfs-fun-impl-correct-unfolded = sub-dfs-fun-impl-correct*[*unfolded sub-cycle-iff-in-sup-st*]

theorem *sub-dfs-code-correct*:
⟦ *x ∈ V*; *x ∉ set.α rs*; *s = sub-dfs-code x* ⟧ ⟹ *fst (istate s) ⟷ sub-cycle {x. ss x} (set.α rs) x*
using *order-trans*[*OF dfs-code-refine sub-dfs-fun-impl-correct*, *of x*]
unfolding *dfs-impl-build-rel-def*
by (*auto elim*!: *RETURN-ref-SPECD*)

lemmas *sub-dfs-code-correct-unfolded = sub-dfs-code-correct*[*unfolded sub-cycle-iff-in-sup-st*]

lemmas *sub-dfs-code-preserves-invar = dfs-code-preserves-invar*[*OF sub-dfs-preserves-invar*]
lemmas *sub-dfs-code-preserves-dfs-invar = dfs-code-preserves-dfs-invar*[*OF sub-dfs-preserves-invar*]

lemma *sub-dfs-code-finished-unfolded*:
assumes x: $x \in V\ x \notin set.\alpha\ rs$
and *s-def*: $s = sub\text{-}dfs\text{-}code\ x$
and $\neg\ fst\ (istate\ s)$
shows *finished* $(impl\text{-}sws\text{-}\alpha\ s) = \{v.\ x = v \vee x \rightarrow\backslash set.\alpha\ rs+\ v\}$
proof −
let *?DFS* $= sub\text{-}dfs\ \{x.\ ss\ x\}\ (set.\alpha\ rs)$
from ‹$\neg\ fst\ (istate\ s)$› **have** $\neg\ state\ (impl\text{-}sws\text{-}\alpha\ s)$ **by** *simp*
moreover from *dfs-code-finished*[*OF sub-dfs-preserves-invar*] *x s-def* **have** *dfs-finished ?DFS* $(impl\text{-}sws\text{-}\alpha\ s)$ **by** *auto*
ultimately have *dfs-completed ?DFS* $(impl\text{-}sws\text{-}\alpha\ s)$ **unfolding** *dfs-completed-def* **by** *simp*
note *completed-finished-eq-reachable*[*OF this*]
moreover from *dfs-code-constructable*[*OF sub-dfs-preserves-invar*] *x s-def* **have** $impl\text{-}sws\text{-}\alpha\ s \in dfs\text{-}constr\text{-}from\ ?DFS\ x$ **by** *simp*
with *constr-from-implies-start* **have** $start\ (impl\text{-}sws\text{-}\alpha\ s) = x$ **by** *blast*
ultimately
show *finished* $(impl\text{-}sws\text{-}\alpha\ s) = \{v.\ x = v \vee x \rightarrow\backslash set.\alpha\ rs+\ v\}$ **by** *auto*
qed
end

locale *NestedDFS-pre* = *set*: *StdSet sops* +
setA: *set-memb setAα setAinvar setAmemb* +
sws-impl mops V E σα +
finite-accepting-digraph V E setAα $\mathcal{A}$
for *sops* :: $('n,\ 's,\ 'Y)$ *set-ops-scheme*
and *setAα* :: $'sA \Rightarrow 'n\ set$ **and** *setAinvar* :: $'sA \Rightarrow bool$ **and** *setAmemb* :: $'n \Rightarrow 'sA \Rightarrow bool$
and *mops* :: $('n,\ nat,\ 'm,\ 'X)$ *map-ops-scheme*
and $\sigma\alpha :: bool \times 's \Rightarrow bool \times 'n\ set$
and $V::'n\ set$ **and** $E :: ('n \times 'n)\ set$ **and** $\mathcal{A} :: 'sA+$
fixes $\gamma succs :: 'n \Rightarrow 'n\ list$
defines $\sigma\alpha\text{-}def$: $\sigma\alpha \equiv \lambda(b,s).\ (b,\ set.\alpha\ s)$
assumes *ssuccs-correct*: $set\ (\gamma succs\ y) = succs\ y$
and *ssuccs-distinct*: $distinct\ (\gamma succs\ y)$
and *A-invar*: *setAinvar* $\mathcal{A}$
begin

lemma $\sigma\alpha\text{-}simps[simp]$:
$fst\ (\sigma\alpha\ S) \longleftrightarrow fst\ S$
$snd\ (\sigma\alpha\ S) = set.\alpha\ (snd\ S)$
$\sigma\alpha\ (b,\ F) = (b,\ set.\alpha\ F)$
unfolding $\sigma\alpha\text{-}def$
by $(cases\ S,\ simp\text{-}all)+$

end

locale *NestedDFS-Impl-def* = *NestedDFS-pre sops - - - mops*

for *sops* :: (*'n*, *'s*, *'Y*) *set-ops-scheme*
and *mops* :: (*'n*, *nat*, *'m*, *'X*) *map-ops-scheme* +
fixes *M* :: (*bool* × *'n set*, *'n*) *dfs-sws* ⇒ *'n set*
and *γM* :: (*bool* × *'s*, *'n*, *'m*) *impl-sws* ⇒ *'n* ⇒ *bool*
assumes *M-correct*: *γM s* = *M* (*impl-sws-α s*)
and *M-defs*: ⋀*s*. *stack s* ≠ [] ⟹ *hd* (*stack s*) ∈ *M s* ⋀*s*. *stack s* ≠ [] ⟹ *M s* ⊆ *set* (*stack s*)
begin

declare *A-invar*[*simp*]

definition *sub-impl* :: (*'n* ⇒ *bool*) ⇒ *'s* ⇒ *'n* ⇒ *'s option*
where *sub-impl ss R x* ≡ *let s* = *SubDFS-Impl.sub-dfs-code sops mops ss R γsuccs x*
in if fst (*istate s*) *then None else Some* (*snd* (*istate s*))

fun *run-sub-dfs-impl* **where**
run-sub-dfs-impl (*True*, *F*) - - = (*True*, *F*)
| *run-sub-dfs-impl* (*False*, *F*) *s e* = (*if setAmemb e* 𝒜 *then*
case sub-impl (*γM s*) *F e of*
None ⇒ (*True*, *F*)
| *Some F'* ⇒ (*False*, *F'*)
else (*False*, *F*))

abbreviation *nested-impl-invar'* **where**
nested-impl-invar' ≡ *set.invar* ∘ *snd*

definition *nested-impl-invar* **where**
nested-impl-invar ≡ *dfs-impl-state-invar nested-impl-invar'*

lemma *nested-impl-invar-conv*[*simp*]:
nested-impl-invar s ⟷ *set.invar* (*snd* (*istate s*))
unfolding *nested-impl-invar-def dfs-impl-state-invar-def*
by *simp*

definition *nested-impl-dfs* :: (*bool* × *'s*, *'n*, *'m*, *'n es*) *dfs-algorithm-impl* **where**
nested-impl-dfs = (| *dfs-impl-cond* = λ*S*. ¬ *fst S*,
dfs-impl-action = λ*S* - -. *S*,
dfs-impl-post = *run-sub-dfs-impl*,
dfs-impl-remove = λ*S* - -. *S*,
dfs-impl-start = λ*x*. (*False*, *set.empty* ()),
dfs-impl-restrict = *es-empty* (),
dfs-impl-invar = *nested-impl-invar* |)

lemma *nested-impl-dfs-simps*[*simp*]:
dfs-impl-cond nested-impl-dfs S ⟷ ¬ *fst S*
dfs-impl-post nested-impl-dfs = *run-sub-dfs-impl*
dfs-impl-action nested-impl-dfs S s x = *S*
dfs-impl-remove nested-impl-dfs S s x = *S*

dfs-impl-start nested-impl-dfs x = (*False*, *set.empty* ())
dfs-impl-restrict nested-impl-dfs = *es-empty* ()
dfs-impl-invar nested-impl-dfs = *nested-impl-invar*
unfolding *nested-impl-dfs-def*
by *simp-all*

lemma *sub-impl-correct*:
assumes *inv*: *set.invar R*
and *x*: *x* ∈ *V x* ∉ *set.α R*
shows *Option.map* (*set.α*) (*sub-impl ss R x*) = *Option.map* (λ*s*. *s* ∪ *set.α R*) (*run-sub-dfs'* {*x*. *ss x*} (*set.α R*) *x*)
proof −
from *inv* **interpret** *sub*!: *SubDFS-Impl γsuccs sops mops V E ss R* **by** *unfold-locales* (*simp-all add*: *ssuccs-distinct ssuccs-correct*)
note *sub* = *this*
show *?thesis*
proof (*cases run-sub-dfs'* {*x*. *ss x*} (*set.α R*) *x*)
case *None* **with** *run-sub-dfs'-correct*[*OF x*] **have** ∃ *v*. *x* →*set.α R*+ *v* ∧ *ss v* **by** *simp*
hence *fst* (*istate* (*sub.sub-dfs-code x*)) **using** *sub sub.sub-dfs-code-correct-unfolded*[*OF x*] **by** *blast*
with *None* **show** *?thesis* **unfolding** *sub-impl-def* **by** *simp*
next
case (*Some F*) **def** *s* ≡ *sub.sub-dfs-code x*
hence *inv*: *sub.impl-sws-invar s* **using** *sub.sub-dfs-code-preserves-invar x* **by** *simp*

from *s-def* **have** ∗: *set.α* (*snd* (*istate s*)) = *dom* (*map.α* (*ifinish s*)) ∪ *set.α R*
using *sub.sub-dfs-code-preserves-dfs-invar x* **by** (*auto simp add*: *sub.sub-impl-dfs-invar-def*)

from *Some* **have** *run-sub-dfs'* {*x*. *ss x*} (*set.α R*) *x* ≠ *None* **by** *simp*
with *run-sub-dfs'-correct*[*OF x*] **have** *nis*: ¬ *fst* (*istate s*) **using** *sub.sub-dfs-code-correct-unfolded*[*OF x*] *s-def* **by** *auto*
with *sub.sub-dfs-code-finished-unfolded x s-def* **have** *finished* (*sub.impl-sws-α s*) = {*v*. *x* = *v* ∨ *x* →*set.α R*+ *v*} **by** *simp*
with *sub.impl-sws-invar-mapinvars*[*OF inv*] **have** *dom* (*map.α* (*ifinish s*)) = {*v*. *x* = *v* ∨ *x* →*set.α R*+ *v*} **by** (*simp add*: *map.correct*)
with *nis Some* **show** *?thesis* **using** *run-sub-dfs'-finished*[*OF x Some*] ∗ **unfolding** *sub-impl-def s-def* **by** *auto*
qed
qed

lemma *sub-impl-set-invar*:
assumes *inv*: *set.invar R*
and *x*: *x* ∈ *V x* ∉ *set.α R*
and *some*: *sub-impl ss R x* = *Some F'*
shows *set.invar F'*
proof −

from *inv* **interpret** *sub*!: *SubDFS-Impl* γ*succs sops mops V E ss R* **by** *unfold-locales* (*simp-all add*: *ssuccs-distinct ssuccs-correct*)

def $s \equiv$ *sub.sub-dfs-code x*
hence *set.invar* (*snd* (*istate s*)) **using** *sub.sub-dfs-code-preserves-dfs-invar x* **by** (*auto simp add*: *sub.sub-impl-dfs-invar-def*)
with *s-def some* **show** *?thesis* **unfolding** *sub-impl-def* **by** (*cases fst* (*istate s*)) *simp-all*
qed

lemma *run-sub-dfs-impl-correct'*:
assumes $x \in$ *set* (*istack s*) *nested-impl-invar s nested-dfs-invar* (*impl-sws-*α *s*)
and *constr*: *dfs-constructable* (*nested-dfs M*) (*impl-sws-*α *s*)
shows $\sigma\alpha$ (*run-sub-dfs-impl* (*istate s*) *s x*) = *run-sub-dfs M* ($\sigma\alpha$ (*istate s*)) (*impl-sws-*α *s*) *x*
proof (*cases istate s*)
case (*Pair b F*) **hence** *run-sub-dfs M* (*b*, *set.*α *F*) (*impl-sws-*α *s*) *x* = $\sigma\alpha$ (*run-sub-dfs-impl* (*b*, *F*) *s x*) (**is** *?L* = *?R*)
proof (*cases b*)
case *True* **hence** *?L* = (*b*, *set.*α *F*) **by** *simp*
also from *True* **have** ... = *?R* **by** *simp*
finally show *?thesis* **.**
next
case *False* **with** *Pair* **have** $\neg$ *has-cycle* (*impl-sws-*α *s*) **by** *simp*
with *assms* **have** *set* (*istack s*) $\cap$ *set.*α *F* = {} **unfolding** *nested-dfs-invar-def* **by** (*simp add*: *Pair*)
with *assms* **have** $x \notin$ *set.*α *F* **by** *auto*
hence *xnF*:$x \notin$ *snd* (*b*, *set.*α *F*) **by** *simp*

from *False* **have** *nb*: $\neg$ *fst* (*b*, *set.*α *F*) **by** *simp*

from *constr stack-subset-verts* **have** *set* (*istack s*) $\subseteq$ *V* **by** *fastforce*
with *assms* **have** *xV*: $x \in$ *V* **by** *auto*

from *assms Pair* **have** *F-inv*: *set.invar F* **by** *simp*
note *sub-impl-correct* = *sub-impl-correct*[*OF F-inv xV* ‹$x \notin$ *set.*α *F*›, **where** *ss* = γ*M s*]

show *?thesis* **using** *nb xV xnF*
proof (*cases rule*: *run-sub-dfs-casesE*[**where** *s*=*impl-sws-*α *s* **and** *M*=*M*])
case *same* **hence** $\neg$ *setAmemb x* $\mathcal{A}$ **by** (*simp add*: *setA.memb-correct*)
hence *?R* = (*b*, *set.*α *F*) **by** (*simp add*: *False*)
with *same* **show** *?thesis* **by** *simp*
next
case (*cycle v*) **hence** *setAmemb x* $\mathcal{A}$ **by** (*simp add*: *setA.memb-correct*)

moreover
from *cycle* **have** *run-sub-dfs'* {*x*. γ*M s x*} (*set.*α *F*) *x* = *None* **by** (*auto simp add*: *set.correct M-correct Collect-def*)

```
    hence sub-impl (γM s) F x = None using sub-impl-correct by simp

    ultimately show ?thesis using cycle False by simp
   next
    case (no-cycle F') hence setAmemb x A by (simp add: setA.memb-correct)

    moreover
     from no-cycle have run-sub-dfs' {x. γM s x} (set.α F) x = Some F' by
(simp add: set.correct M-correct Collect-def)
     hence Option.map set.α (sub-impl (γM s) F x) = Some (F' ∪ set.α F)
using sub-impl-correct by simp
    ultimately show ?thesis using no-cycle False by auto
   qed
  qed
  with Pair show ?thesis by simp
qed

lemma run-sub-dfs-preserves-invar:
  assumes x ∈ set (istack s) nested-impl-invar' (istate s)
  and constr: dfs-constructable (nested-dfs M) (impl-sws-α s)
  shows nested-impl-invar' (run-sub-dfs-impl (istate s) s x)
proof −
  obtain b F where bF: istate s = (b, F) by (cases istate s) auto
  with assms have F-inv: set.invar F by simp
  with bF show ?thesis
  proof (cases b)
    case False with F-inv bF show ?thesis
    proof (cases setAmemb x A)
      case True with F-inv False bF show ?thesis
      proof (cases sub-impl (γM s) F x)
        case (Some F')
        note F-inv
          moreover from constr stack-subset-verts have set (istack s) ⊆ V by
fastforce
        with assms have x ∈ V by auto
       moreover from constr have nested-dfs-invar (impl-sws-α s) using dfs-constructable-invarI[OF
- nested-dfs-preserves-invar] M-defs by simp
      with False bF have set (istack s) ∩ set.α F = {} unfolding nested-dfs-invar-def
by simp
        with assms have x ∉ set.α F by auto
        ultimately have set.invar F' using sub-impl-set-invar [OF - - - Some] by
metis+
        with Some False True bF show ?thesis by simp
      qed simp
    qed simp
  qed simp
qed

end
```

locale *NestedDFS-Impl* = *NestedDFS-Impl-def* - - - $\sigma\alpha$ *V E* - γ*succs sops mops* +
DFS-Impl (*nested-dfs M*) *nested-impl-dfs mops es-*α *es-invar*
es-memb V E $\sigma\alpha$ γ*succs*
for *sops* :: ('*n*, '*s*, '*Y*) *set-ops-scheme*
and *mops* :: ('*n*, *nat*, '*m*, '*X*) *map-ops-scheme*
and *V*::'*n set* **and** *E* :: ('*n* × '*n*) *set*
and γ*succs* :: '*n* ⇒ '*n list*
and $\sigma\alpha$ :: *bool* × '*s* ⇒ *bool* × '*n set*
begin

lemma *run-sub-dfs-impl-correct*:
assumes *x* ∈ *set* (*istack s*) *nested-impl-invar s impl-sws-invar s*
shows $\sigma\alpha$ (*run-sub-dfs-impl* (*istate s*) *s x*) = *run-sub-dfs M* ($\sigma\alpha$ (*istate s*))
(*impl-sws-*α *s*) *x*
proof −
from *assms* **have** *dfs-constructable* (*nested-dfs M*) (*impl-sws-*α *s*) **and** *nested-dfs-invar*
(*impl-sws-*α *s*)
using *impl-sws-invar-dfs-invar*[*OF nested-dfs-preserves-invar*] *M-defs* **by** *auto*
with *run-sub-dfs-impl-correct'* *assms* **show** *?thesis* **by** *blast*
qed

lemma *nested-impl-preserves*:
impl-preserves-invar
using *nested-impl-dfs-simps*(*7*)[*unfolded nested-impl-invar-def*]
proof (*rule state-impl-preserves-invarI*)
case *goal2* **hence** *dfs-constructable* (*nested-dfs M*) (*impl-sws-*α *s*) **by** *auto*
note *run-sub-dfs-preserves-invar*[*OF goal2*(*1*,*2*) *this*]
thus *?case* **by** *simp*
qed (*simp-all add*: *set.correct*)
end

sublocale *NestedDFS-Impl* ⊆ *DFS-Impl-correct nested-dfs M nested-impl-dfs mops*
es-α *es-invar es-memb V E* $\sigma\alpha$
by *unfold-locales* (*simp-all add*: *run-sub-dfs-impl-correct set.correct nested-impl-preserves*)

locale *BasicDFS-Impl-def* = *NestedDFS-pre*
begin

definition *basicM-impl s* ≡ λ*x*. *x* = *hd* (*istack s*)

lemma *basicM-impl-correct*:
basicM-impl s = *basicM* (*impl-sws-*α *s*)
unfolding *basicM-impl-def basicM-def*
by *simp* (*metis Collect-def singleton-conv*)
end

sublocale *BasicDFS-Impl-def* ⊆ *NestedDFS-Impl-def* - - - - - - - - - - *basicM*
basicM-impl

by *unfold-locales* (*simp-all add*: *basicM-impl-correct* $\sigma\alpha$*-def*)

locale *BasicDFS-Impl* = *BasicDFS-Impl-def*

sublocale *BasicDFS-Impl* $\subseteq$ *NestedDFS-Impl* - - - - *basicM basicM-impl*
by *unfold-locales* (*simp-all add*: *ssuccs-correct ssuccs-distinct*)

context *BasicDFS-Impl*
begin

abbreviation *basic-impl-dfs* $\equiv$ *nested-impl-dfs*
abbreviation *basic-dfs-fun-impl* $\equiv$ *dfs-fun-impl*
abbreviation *basic-dfs-code* $\equiv$ *dfs-code*
abbreviation *basic-build-rel* $\equiv$ *dfs-impl-build-rel*

theorem *basic-dfs-fun-impl-correct*:
assumes $x \in V$
shows *basic-dfs-fun-impl* $x \leq \Downarrow$ *basic-build-rel* (*SPEC* (λs. *has-cycle* $s \longleftrightarrow$ *cycle basicM* x))
proof −
note *dfs-fun-impl-refine*[*OF assms*, *folded basic-dfs-def*]
also note *dfs-fun-basic-dfs-correct*[*OF assms*]
finally show *?thesis* **by** (*simp add*: *basic-dfs-def*)
qed

lemmas *basic-dfs-fun-impl-correct-unfolded* = *basic-dfs-fun-impl-correct*[*unfolded basic-cycle-iff-cycle*]

lemmas *basic-dfs-code-preserves-invar* = *dfs-code-preserves-invar*[*folded basic-dfs-def*, *OF basic-dfs-preserves-invar*]
lemmas *basic-dfs-code-preserves-dfs-invar* = *dfs-code-preserves-dfs-invar*[*folded basic-dfs-def*, *OF basic-dfs-preserves-invar*]

theorem *basic-dfs-code-correct*:
$[\![x \in V; s = \textit{basic-dfs-code } x]\!] \Longrightarrow \textit{fst } (\textit{istate } s) \longleftrightarrow \textit{cycle basicM } x$
using *order-trans*[*OF dfs-code-refine basic-dfs-fun-impl-correct*, *of x*]
unfolding *dfs-impl-build-rel-def*
by (*auto elim*!: *RETURN-ref-SPECD*)

lemmas *basic-dfs-code-correct-unfolded* = *basic-dfs-code-correct*[*unfolded basic-cycle-iff-cycle*]
end

locale *HPYDFS-Impl-def* = *NestedDFS-pre*
begin

definition *hpyM-impl* $s \equiv \lambda x$. *set.memb* x (*set.from-list* (*istack* s))

lemma *hpyM-impl-correct*:
hpyM-impl s = *hpyM* (*impl-sws-*α s)

unfolding *hpyM-impl-def hpyM-def*
by (*simp add*: *set.correct*) (*metis* (*full-types*) *Collect-conj-eq Collect-def Int-absorb Int-def*)
end

sublocale *HPYDFS-Impl-def* ⊆ *NestedDFS-Impl-def* - - - - - - - - - - *hpyM hpyM-impl*
by *unfold-locales* (*simp-all add*: *hpyM-impl-correct* σα*-def*)

locale *HPYDFS-Impl* = *HPYDFS-Impl-def*

sublocale *HPYDFS-Impl* ⊆ *NestedDFS-Impl* - - - - *hpyM hpyM-impl*
by *unfold-locales* (*simp-all add*: *ssuccs-correct ssuccs-distinct*)

context *HPYDFS-Impl*
begin

abbreviation *hpy-impl-dfs* ≡ *nested-impl-dfs*
abbreviation *hpy-dfs-fun-impl* ≡ *dfs-fun-impl*
abbreviation *hpy-dfs-code* ≡ *dfs-code*
abbreviation *hpy-build-rel* ≡ *dfs-impl-build-rel*

theorem *hpy-dfs-fun-impl-correct*:
assumes $x \in V$
shows *hpy-dfs-fun-impl* $x \leq \Downarrow$ *hpy-build-rel* (*SPEC* (λ*s*. *has-cycle s* ⟷ *cycle hpyM x*))
proof −
note *dfs-fun-impl-refine*[*OF assms*, *folded hpy-dfs-def*]
also note *dfs-fun-hpy-dfs-correct*[*OF assms*]
finally show *?thesis* **by** (*simp add*: *hpy-dfs-def*)
qed

lemmas *hpy-dfs-fun-impl-correct-unfolded* = *hpy-dfs-fun-impl-correct*[*unfolded hpy-cycle-iff-cycle*]

lemmas *hpy-dfs-code-preserves-invar* = *dfs-code-preserves-invar*[*folded hpy-dfs-def*, *OF hpy-dfs-preserves-invar*]
lemmas *hpy-dfs-code-preserves-dfs-invar* = *dfs-code-preserves-dfs-invar*[*folded hpy-dfs-def*, *OF hpy-dfs-preserves-invar*]

theorem *hpy-dfs-code-correct*:
⟦ $x \in V$; s = *hpy-dfs-code x* ⟧ ⟹ *fst* (*istate s*) ⟷ *cycle hpyM x*
using *order-trans*[*OF dfs-code-refine hpy-dfs-fun-impl-correct*, *of x*]
unfolding *dfs-impl-build-rel-def*
by (*auto elim*!: *RETURN-ref-SPECD*)

lemmas *hpy-dfs-code-correct-unfolded* = *hpy-dfs-code-correct*[*unfolded hpy-cycle-iff-cycle*]
end

locale *SEDFS-Impl-def* = *HPYDFS-Impl-def*

begin

fun *SE-remove-impl* :: (*bool* × *'b*) ⇒ (*bool* × *'b*, *'a*, *'e*) *impl-sws* ⇒ *'a* ⇒ (*bool* × *'b*) **where**
  SE-remove-impl (*True*, *F*) - - = (*True*, *F*)
| *SE-remove-impl* (*False*, *F*) *s x* = ((*setAmemb x* $\mathcal{A}$ ∨ *setAmemb* (*hd* (*istack s*)) $\mathcal{A}$) ∧ *map.lookup x* (*ifinish s*) = *None*, *F*)

definition *SE-impl-dfs* :: (*bool* × *'b*, *'a*, *'e*, *'a es*) *dfs-algorithm-impl* **where**
  SE-impl-dfs = *nested-impl-dfs* (| *dfs-impl-remove* := *SE-remove-impl* |)

lemma *SE-impl-dfs-simps*[*simp*]:
  dfs-impl-cond SE-impl-dfs S ⟷ ¬ *fst S*
  dfs-impl-post SE-impl-dfs = *run-sub-dfs-impl*
  dfs-impl-action SE-impl-dfs S s x = *S*
  dfs-impl-remove SE-impl-dfs = *SE-remove-impl*
  dfs-impl-start SE-impl-dfs x = (*False*, *set.empty* ())
  dfs-impl-restrict SE-impl-dfs = *es-empty* ()
  dfs-impl-invar SE-impl-dfs = *nested-impl-invar*
unfolding *SE-impl-dfs-def*
by *simp-all*
end

locale *SEDFS-Impl* = *SEDFS-Impl-def sops* - - - *mops* σα *V E* - γ*succs* +
                DFS-Impl SE-dfs SE-impl-dfs mops es-α es-invar es-memb V E σα γ*succs*
  for *sops* :: (*'n*, *'s*, *'Y*) *set-ops-scheme*
  and *mops* :: (*'n*, *nat*, *'m*, *'X*) *map-ops-scheme*
  and *V*::*'n set* **and** *E* :: (*'n* × *'n*) *set*
  and γ*succs* :: *'n* ⇒ *'n list*
  and σα :: *bool* × *'s* ⇒ *bool* × *'n set*
begin

lemma *SE-remove-correct*:
  assumes *x-in-d*: *map.lookup x* (*idisc s*) ≠ *None*
  and *inv*: *impl-sws-invar s*
 shows σα (*SE-remove-impl* (*istate s*) *s x*) = *SE-remove* (σα (*istate s*)) (*impl-sws-α s*) *x*
proof (*cases istate s*)
 case (*Pair b F*) **hence** *SE-remove* (*b*, *set.α F*) (*impl-sws-α s*) *x* = σα (*SE-remove-impl* (*b*, *F*) *s x*) (**is** *?L* = *?R*)
  proof (*cases b*)
    case *True* **hence** *?L* = (*b*, *set.α F*) **by** *simp*
    also from *True* **have** ... = *?R* **by** *simp*
    finally show *?thesis* **.**
  next
    case *False*
    from *inv* **have** *constr*: *dfs-constructable SE-dfs* (*impl-sws-α s*) **by** *auto*

from *inv* **have** *map.lookup x (ifinish s) = None ⟷ x ∉ finished (impl-sws-α s)* **by** (*auto simp add*: *map.correct impl-sws-invar-mapinvars*)
also have *... ⟷ x ∈ set (istack s)*
proof −
from *inv x-in-d* **have** *d*: *x ∈ discovered (impl-sws-α s)* **by** (*auto simp add*: *map.correct impl-sws-invar-mapinvars*)
thus *?thesis* **by** (*bestsimp dest*!: *discovered-not-finished-implies-stack*[*OF constr d*] *intro*!: *stack-implies-not-finished*[*OF constr*])
qed
finally show *?thesis* **by** (*simp add*: *setA.memb-correct map.correct False*)
qed
thus *?thesis* **by** (*simp add*: *Pair*)
qed

lemma *SE-impl-preserves*:
impl-preserves-invar
using *SE-impl-dfs-simps*(*7*)[*unfolded nested-impl-invar-def*]
proof (*rule state-impl-preserves-invarI*)
case *goal2* **hence** *constr*: *dfs-constructable SE-dfs (impl-sws-α s)* **by** *auto*
show *?case*
proof (*cases has-cycle (impl-sws-α s)*)
case *False* **with** *SE-to-nested' constr* **have** *dfs-constructable hpy-dfs (impl-sws-α s)* **by** *blast*
with *goal2 run-sub-dfs-preserves-invar* **show** *?thesis* **by** (*simp add*: *hpy-dfs-def*)
next
case *True* **with** *goal2* **show** *?thesis* **by** (*cases istate s*) *simp*
qed
next
case *goal3* **thus** *?case* **by** (*cases istate s*, *cases fst (istate s)*) *simp-all*
qed (*simp-all add*: *set.correct*)

lemma *run-sub-dfs-impl-correct*:
assumes *x ∈ set (istack s) nested-impl-invar s impl-sws-invar s*
shows *σα (run-sub-dfs-impl (istate s) s x) = run-sub-dfs hpyM (σα (istate s)) (impl-sws-α s) x*
proof −
from *assms* **have** *constr*: *dfs-constructable SE-dfs (impl-sws-α s)* **and** *inv*: *nested-dfs-invar (impl-sws-α s)*
using *impl-sws-invar-dfs-invar*[*OF SE-dfs-preserves-invar*] **by** *auto*
show *?thesis*
proof (*cases has-cycle (impl-sws-α s)*)
case *False* **with** *constr SE-to-nested'* **have** *dfs-constructable hpy-dfs (impl-sws-α s)* **by** *blast*
with *inv run-sub-dfs-impl-correct' assms* **show** *?thesis* **by** (*simp add*: *hpy-dfs-def*)
next
case *True* **thus** *?thesis* **by** (*cases istate s*) *simp*
qed
qed
end

sublocale *SEDFS-Impl* $\subseteq$ *DFS-Impl-correct SE-dfs SE-impl-dfs mops es-*α *es-invar es-memb V E* $\sigma\alpha$
by *unfold-locales* (*simp-all add*: *run-sub-dfs-impl-correct SE-remove-correct set.correct SE-impl-preserves*)

context *SEDFS-Impl*
begin

abbreviation *SE-dfs-fun-impl* $\equiv$ *dfs-fun-impl*
abbreviation *SE-dfs-code* $\equiv$ *dfs-code*
abbreviation *SE-build-rel* $\equiv$ *dfs-impl-build-rel*

theorem *SE-dfs-fun-impl-correct*:
  assumes $x \in V$
  shows *SE-dfs-fun-impl x* $\leq \Downarrow$ *SE-build-rel* (*SPEC* (λs. *has-cycle s* $\longleftrightarrow$ *SE-cycle x*))
proof −
  note *dfs-fun-impl-refine*[*OF assms*]
  also note *dfs-fun-SE-dfs-correct*[*OF assms*]
  finally show *?thesis* **by** *simp*
qed

lemmas *SE-dfs-fun-impl-correct-unfolded* = *SE-dfs-fun-impl-correct*[*unfolded SE-cycle-iff-cycle*]

lemmas *SE-dfs-code-preserves-invar* = *dfs-code-preserves-invar*[*OF SE-dfs-preserves-invar*]
lemmas *SE-dfs-code-preserves-dfs-invar* = *dfs-code-preserves-dfs-invar*[*OF SE-dfs-preserves-invar*]

theorem *SE-dfs-code-correct*:
  ⟦ $x \in V$; s = *SE-dfs-code x* ⟧ $\Longrightarrow$ *fst* (*istate s*) $\longleftrightarrow$ *SE-cycle x*
using *order-trans*[*OF dfs-code-refine SE-dfs-fun-impl-correct*, *of x*]
unfolding *dfs-impl-build-rel-def*
by (*auto elim*!: *RETURN-ref-SPECD*)

lemmas *SE-dfs-code-correct-unfolded* = *SE-dfs-code-correct*[*unfolded SE-cycle-iff-cycle*]
end
end