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* $\Rightarrow ('S, 'n)$ *dfs-sws* $\Rightarrow ('S, 'n)$ *dfs-sws set*
where *dfs-step dfs* $s \equiv$ *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) $\neq \{\}$
$\Longrightarrow$ *dfs-step dfs* $s = \{$*dfs-sws.make* (*start* s) $st'\ w'\ d'$ (*finish* s) $c'\ s' \mid e\ st'\ w'\ d'\ c'\ s'.\ e \in hd$ (*wl* s) $\wedge$
(*if* $e \in$ *dfs-restrict dfs then* $st' = x\#xs \wedge w' = (hd$ (*wl* s) $- \{e\})\#tl$ (*wl* s) $\wedge\ d' =$ *discover* $s \wedge c' =$ *counter* $s \wedge s' =$ (*state* s)
else if $e \in$ *discovered* s *then* $st' = x\#xs \wedge w' = (hd$ (*wl* s) $- \{e\})\#tl$ (*wl* s) $\wedge\ d' =$ *discover* $s \wedge c' =$ *counter* $s \wedge s' =$ *dfs-remove dfs* (*state* s) $s\ e$
else $st' = e\#x\#xs \wedge w' =$ *succs* $e\#(hd$ (*wl* s) $- \{e\})\#tl$ (*wl* s) $\wedge\ d' =$ (*discover* s)$(e \mapsto$ *counter* s) $\wedge\ c' =$ *Suc* (*counter* s) $\wedge\ s' =$ *dfs-action dfs* (*state* s) $s\ e)\}$
⟨*proof*⟩

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) $\neq \{\} \Longrightarrow e \in$ *hd* (*wl* s) $\Longrightarrow e \in$ *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) $\neq \{\} \Longrightarrow e \in$ *hd* (*wl* s) $\Longrightarrow e \notin$ *dfs-restrict dfs* $\Longrightarrow e \in$ *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) $\neq \{\} \Longrightarrow e \in$ *hd* (*wl* s) $\Longrightarrow e \notin$ *dfs-restrict dfs* $\Longrightarrow e \notin$ *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)$(e \mapsto$ *counter* s), *counter* $:=$ *Suc* (*counter* s)$|\!) \Longrightarrow s' \in$ *dfs-step dfs* s
⟨*proof*⟩

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
⟨*proof*⟩

lemma *dfs-step-cases-elem* [*case-names empty restrict remove visit*, *cases pred*: *dfs-step*, *consumes 2*]:
assumes *inv*: $s' \in$ *dfs-step dfs s wl s* $\neq []$
and *empty*: $\bigwedge x\ xs\ w\ ws.\ [\![\ stack\ s = x\#xs;\ wl\ s = w\#ws;\ w = \{\}\ ]\!] \Longrightarrow$
$s' = s(\!|stack := xs,\ wl := ws,\ finish := (finish\ s)(x \mapsto counter\ s),\ counter := Suc\ (counter\ s),\ state := dfs\text{-}post\ dfs\ (state\ s)\ s\ x\ |\!) \Longrightarrow P$
and *restrict*: $\bigwedge e\ x\ xs\ w\ ws.\ [\![\ stack\ s = x\#xs;\ wl\ s = w\#ws;\ w \neq \{\};\ e \in w;\ e \in dfs\text{-}restrict\ dfs\ ]\!] \Longrightarrow$
$s' = s(\!|\ wl := (w - \{e\})\#ws\ |\!) \Longrightarrow P$
and *remove*: $\bigwedge e\ x\ xs\ w\ ws.\ [\![\ stack\ s = x\#xs;\ wl\ s = w\#ws;\ w \neq \{\};\ e \in w;\ e \notin dfs\text{-}restrict\ dfs;\ e \in discovered\ s\ ]\!] \Longrightarrow$
$s' = s(\!|\ wl := (w - \{e\})\#ws,\ state := dfs\text{-}remove\ dfs\ (state\ s)\ s\ e|\!) \Longrightarrow P$
and *visit*: $\bigwedge e\ x\ xs\ w\ ws.\ [\![\ stack\ s = x\#xs;\ wl\ s = w\#ws;\ w \neq \{\};\ e \in w;\ e \notin dfs\text{-}restrict\ dfs;\ e \notin discovered\ s\ ]\!] \Longrightarrow$
$s' = s(\!|\ state := dfs\text{-}action\ dfs\ (state\ s)\ s\ e,\ stack := e\#x\#xs,\ wl := succs\ e\#(w - \{e\})\#ws,\ discover := (discover\ s)(e \mapsto counter\ s),\ counter := Suc\ (counter\ s)|\!) \Longrightarrow P$
shows P
⟨*proof*⟩

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*: $\forall\ n < length\ (stack\ s).\ P\ (stack\ s\ !\ n)\ (wl\ s\ !\ n)\ (discover\ s)\ (finish\ s)\ (start\ s)$
and *case-0*: $P\ x\ w\ (discover\ s)\ (finish\ s)\ (start\ s) \Longrightarrow P\ x\ (w - \{e\})\ (discover\ s')\ (finish\ s')\ (start\ s')$
and *step*: $\bigwedge n.\ [\![\ n < length\ (stack\ s);\ n > 0;\ P\ (stack\ s\ !\ n)\ (wl\ s\ !\ n)\ (discover\ s)\ (finish\ s)\ (start\ s)\ ]\!] \Longrightarrow P\ (stack\ s\ !\ n)\ (wl\ s\ !\ n)\ (discover\ s')\ (finish\ s')\ (start\ s')$
shows $\forall n < length\ (stack\ s').\ P\ (stack\ s'\ !\ n)\ (wl\ s'\ !\ n)\ (discover\ s')\ (finish\ s')\ (start\ s')$
⟨*proof*⟩

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*: $\forall\ n < length\ (stack\ s).\ P\ (stack\ s\ !\ n)\ (wl\ s\ !\ n)\ (discover\ s)\ (finish\ s)\ (start\ s)$
and *case-0*: $P\ e\ (succs\ e)\ (discover\ s')\ (finish\ s')\ (start\ s')$
and *case-1*: $P\ x\ w\ (discover\ s)\ (finish\ s)\ (start\ s) \Longrightarrow P\ x\ (w - \{e\})\ (discover\ s')\ (finish\ s')\ (start\ s')$
and *step*: $\bigwedge n.\ [\![\ n < length\ (stack\ s);\ n > 0;\ P\ (stack\ s\ !\ n)\ (wl\ s\ !\ n)\ (discover$

$s)$ $(finish\ s)$ $(start\ s)$ $]\!] \Longrightarrow P$ $(stack\ s\ !\ n)$ $(wl\ s\ !\ n)$ $(discover\ s')$ $(finish\ s')$ $(start\ s')$
shows $\forall n < length\ (stack\ s').\ P\ (stack\ s'\ !\ n)\ (wl\ s'\ !\ n)\ (discover\ s')\ (finish\ s')\ (start\ s')$
⟨*proof*⟩

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

lemma *dfs-step-preserves-start*:
$s' \in dfs\text{-}step\ dfs\ s \Longrightarrow wl\ s \neq [] \Longrightarrow start\ s' = start\ s$
⟨*proof*⟩

lemma *dfs-step-discovered-subset*:
$s' \in dfs\text{-}step\ dfs\ s \Longrightarrow wl\ s \neq [] \Longrightarrow discovered\ s' \supseteq discovered\ s$
⟨*proof*⟩

lemma *dfs-step-exists*:
assumes *inv*: $stack\ s \neq []$ $wl\ s \neq []$
shows $dfs\text{-}step\ dfs\ s \neq \{\}$
⟨*proof*⟩

lemma *dfs-step-stack-not-discovered*:
assumes $s' \in dfs\text{-}step\ dfs\ s\ wl\ s \neq []$
and $length\ (stack\ s') > length\ (stack\ s)$
shows $hd\ (stack\ s') \notin discovered\ s$
⟨*proof*⟩

lemma *dfs-step-stack-not-restricted*:
assumes $s' \in dfs\text{-}step\ dfs\ s\ wl\ s \neq []$
and $length\ (stack\ s') > length\ (stack\ s)$
shows $hd\ (stack\ s') \notin dfs\text{-}restrict\ dfs$
⟨*proof*⟩

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 $dfs\text{-}cond\text{-}compl :: ('S, 'n, 'X)\ dfs\text{-}algorithm\text{-}scheme \Rightarrow ('S, 'n)\ dfs\text{-}sws \Rightarrow bool$
where
$dfs\text{-}cond\text{-}compl\ dfs\ s \equiv stack\ s \neq [] \wedge wl\ s \neq [] \wedge dfs\text{-}cond\ dfs\ (state\ s)$

lemma *dfs-cond-compl-cond* [*dest*]:
$dfs\text{-}cond\text{-}compl\ dfs\ s \Longrightarrow dfs\text{-}cond\ dfs\ (state\ s)$
⟨*proof*⟩

lemma *dfs-cond-compl-step-exists* [*dest*]:
$dfs\text{-}cond\text{-}compl\ dfs\ s \Longrightarrow dfs\text{-}step\ dfs\ s \neq \{\}$
⟨*proof*⟩

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'* ∈ *dfs-step dfs s*

lemma *dfs-nextE*:
⟦ *dfs-next dfs s s'*; ⟦*dfs-cond-compl dfs s*; *s'* ∈ *dfs-step dfs s*⟧ ⟹ *P* ⟧ ⟹ *P*
⟨*proof*⟩

lemma *dfs-next-dfs-step*[*dest*]:
dfs-next dfs s s' ⟹ *s'* ∈ *dfs-step dfs s*
⟨*proof*⟩

lemma *dfs-next-stack-notempty*[*simp*, *intro*]:
dfs-next dfs s s' ⟹ *stack s* ≠ []
⟨*proof*⟩

lemma *dfs-next-wl-notempty*[*simp*, *intro*]:
dfs-next dfs s s' ⟹ *wl s* ≠ []
⟨*proof*⟩

lemma *lift-to-dfs-next*:
assumes *s'* ∈ *dfs-step dfs s* ⟹ *wl s* ≠ [] ⟹ *PROP S*
shows *dfs-next dfs s s'* ⟹ *PROP S*
⟨*proof*⟩

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'*
⟨*proof*⟩

lemma *dfs-next-cond-compl*:
dfs-next dfs s s' ⟹ *dfs-cond-compl dfs s*
⟨*proof*⟩

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* $\Rightarrow$ $'n \Rightarrow ('S, 'n)$ *dfs-sws*
where *dfs-constr-start dfs x* $\equiv$ $(\!|$ *start* $= x$, *stack* $= [x]$, *wl* $= [succs\ x]$, *discover* $= [x \mapsto 0]$, *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*
⟨*proof*⟩

inductive *dfs-constructable* :: $('S, 'n, 'X)$ *dfs-algorithm-scheme* $\Rightarrow ('S, 'n)$ *dfs-sws* $\Rightarrow$ *bool*
for *dfs* :: $('S, 'n, 'X)$ *dfs-algorithm-scheme* **where**
start: $\bigwedge x.\ x \in V \Longrightarrow x \notin$ *dfs-restrict dfs* $\Longrightarrow$ *dfs-constructable dfs* (*dfs-constr-start dfs x*)
| *step*: *dfs-next dfs s s'* $\Longrightarrow$ *dfs-constructable dfs s* $\Longrightarrow$ *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'*: $\bigwedge x.\ x \in V \Longrightarrow x \notin$ *dfs-restrict dfs* $\Longrightarrow$ *P* (*dfs-constr-start dfs x*)
and *empty'*: $\bigwedge s\ s'\ x\ xs\ w\ ws.$ ⟦ *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*)$(x \mapsto$ *counter s*), *counter* $:=$ *Suc* (*counter s*), *state* $:=$ *dfs-post dfs* (*state s*) *s x*$|\!)$⟧
$\Longrightarrow$ *P s'*
and *restrict'*: $\bigwedge s\ s'\ e\ x\ xs\ w\ ws.$ ⟦ *dfs-next dfs s s'*; *dfs-constructable dfs s*; *dfs-constructable dfs s'*; *P s*; *stack s* $= x\#xs$; *wl s* $= w\#ws$; $w \neq \{\}$;
$e \in w$; $e \in$ *dfs-restrict dfs*; $s' = s(\!|$ *wl* $:= (w - \{e\})\#ws|\!)$ ⟧
$\Longrightarrow$ *P s'*
and *remove'*: $\bigwedge s\ s'\ e\ x\ xs\ w\ ws.$ ⟦ *dfs-next dfs s s'*; *dfs-constructable dfs s*; *dfs-constructable dfs s'*; *P s*; *stack s* $= x\#xs$; *wl s* $= w\#ws$; $w \neq \{\}$;
$e \in w$; $e \notin$ *dfs-restrict dfs*; $e \in$ *discovered s*; $s' = s(\!|$ *wl* $:= (w - \{e\})\#ws$, *state* $:=$ *dfs-remove dfs* (*state s*) *s e*$|\!)$ ⟧
$\Longrightarrow$ *P s'*

and *visit'*: $\bigwedge s\ s'\ e\ x\ xs\ w\ ws.\ [\![\ dfs\text{-}next\ dfs\ s\ s';\ dfs\text{-}constructable\ dfs\ s;\ dfs\text{-}constructable\ dfs\ s';\ P\ s;\ stack\ s = x\#xs;\ wl\ s = w\#ws;\ w \neq \{\};$
$e \in w;\ e \notin dfs\text{-}restrict\ dfs;\ e \notin discovered\ s;$
$s' = s(\!|\ state := dfs\text{-}action\ dfs\ (state\ s)\ s\ e,\ stack := e\#x\#xs,\ wl := succs\ e\#(w - \{e\})\#ws,\ discover := (discover\ s)(e \mapsto counter\ s),\ counter := Suc\ (counter\ s)\ |\!)\ ]\!]$
$\Longrightarrow P\ s'$
shows $P\ s$
⟨*proof*⟩

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*]: $x \in V \Longrightarrow x \notin dfs\text{-}restrict\ dfs \Longrightarrow dfs\text{-}constr\text{-}start\ dfs\ x \in dfs\text{-}constr\text{-}from\ dfs\ x$
| *step*: $s \in dfs\text{-}constr\text{-}from\ dfs\ x \Longrightarrow dfs\text{-}next\ dfs\ s\ s' \Longrightarrow s' \in dfs\text{-}constr\text{-}from\ dfs\ x$

lemma *dfs-constr-from-rtranclp-dfs-next*:
assumes $x \in V\ x \notin dfs\text{-}restrict\ dfs$
shows $s \in dfs\text{-}constr\text{-}from\ dfs\ x \longleftrightarrow (dfs\text{-}next\ dfs)^{**}\ (dfs\text{-}constr\text{-}start\ dfs\ x)\ s$
⟨*proof*⟩

lemma *dfs-constr-from-constructable*[*dest*]:
$s \in dfs\text{-}constr\text{-}from\ dfs\ x \Longrightarrow dfs\text{-}constructable\ dfs\ s$
⟨*proof*⟩

lemma *dfs-constructable-constr-from-start*:
$dfs\text{-}constructable\ dfs\ s \Longrightarrow s \in dfs\text{-}constr\text{-}from\ dfs\ (start\ s)$
⟨*proof*⟩

lemma *constr-from-implies-start*:
$s \in dfs\text{-}constr\text{-}from\ dfs\ x \Longrightarrow start\ s = x$
⟨*proof*⟩

lemma *counter-larger-one*:
$dfs\text{-}constructable\ dfs\ s \Longrightarrow counter\ s \geq 1$
⟨*proof*⟩

lemma *length-wl-eq-stack*:
$dfs\text{-}constructable\ dfs\ s \Longrightarrow length\ (wl\ s) = length\ (stack\ s)$
⟨*proof*⟩

lemma *succs-discovered-subset-wl-all*:
$dfs\text{-}constructable\ dfs\ s \Longrightarrow \forall\ n < length\ (stack\ s).\ succs\ (stack\ s\ !\ n) - discovered$

$s - \mathit{dfs\text{-}restrict}\ \mathit{dfs} \subseteq \mathit{wl}\ s\ !\ n$
⟨*proof*⟩

lemma *succs-discovered-subset-wl*:
$\mathit{dfs\text{-}constructable}\ \mathit{dfs}\ s \Longrightarrow n < \mathit{length}\ (\mathit{stack}\ s) \Longrightarrow \mathit{succs}\ (\mathit{stack}\ s\ !\ n) - \mathit{discovered}\ s - \mathit{dfs\text{-}restrict}\ \mathit{dfs} \subseteq \mathit{wl}\ s\ !\ n$
⟨*proof*⟩

lemma *empty-wl-succs-discovered*:
$\mathit{dfs\text{-}constructable}\ \mathit{dfs}\ s \Longrightarrow n < \mathit{length}\ (\mathit{stack}\ s) \Longrightarrow \mathit{wl}\ s\ !\ n = \{\} \Longrightarrow \mathit{succs}\ (\mathit{stack}\ s\ !\ n) - \mathit{dfs\text{-}restrict}\ \mathit{dfs} \subseteq \mathit{discovered}\ s$
⟨*proof*⟩

lemma *wl-subset-succs-all*:
$\mathit{dfs\text{-}constructable}\ \mathit{dfs}\ s \Longrightarrow \forall\, n < \mathit{length}\ (\mathit{stack}\ s).\ \mathit{wl}\ s\ !\ n \subseteq \mathit{succs}\ (\mathit{stack}\ s\ !\ n)$
⟨*proof*⟩

lemma *wl-subset-succs*:
$\mathit{dfs\text{-}constructable}\ \mathit{dfs}\ s \Longrightarrow n < \mathit{length}\ (\mathit{stack}\ s) \Longrightarrow \mathit{wl}\ s\ !\ n \subseteq \mathit{succs}\ (\mathit{stack}\ s\ !\ n)$
⟨*proof*⟩

lemma *wl-subset-verts*:
$\mathit{dfs\text{-}constructable}\ \mathit{dfs}\ s \Longrightarrow x \in \mathit{set}\ (\mathit{wl}\ s) \Longrightarrow x \subseteq V$
⟨*proof*⟩

lemma *wl-finite*:
$\mathit{dfs\text{-}constructable}\ \mathit{dfs}\ s \Longrightarrow x \in \mathit{set}\ (\mathit{wl}\ s) \Longrightarrow \mathit{finite}\ x$
⟨*proof*⟩

lemma *discovered-subset-verts*:
$\mathit{dfs\text{-}constructable}\ \mathit{dfs}\ s \Longrightarrow \mathit{discovered}\ s \subseteq V$
⟨*proof*⟩

lemma *discovered-finite*:
$\mathit{dfs\text{-}constructable}\ \mathit{dfs}\ s \Longrightarrow \mathit{finite}\ (\mathit{discovered}\ s)$
⟨*proof*⟩

lemma *ran-discover-finite*:
$\mathit{dfs\text{-}constructable}\ \mathit{dfs}\ s \Longrightarrow \mathit{finite}\ (\mathit{ran}\ (\mathit{discover}\ s))$
⟨*proof*⟩

lemma *stack-subset-discovered*:
$\mathit{dfs\text{-}constructable}\ \mathit{dfs}\ s \Longrightarrow \mathit{set}\ (\mathit{stack}\ s) \subseteq \mathit{discovered}\ s$
⟨*proof*⟩

lemma *stack-subset-verts*:
$\mathit{dfs\text{-}constructable}\ \mathit{dfs}\ s \Longrightarrow \mathit{set}\ (\mathit{stack}\ s) \subseteq V$
⟨*proof*⟩

lemma *stack-distinct*:
dfs-constructable dfs s $\Longrightarrow$ *distinct* (*stack s*)
⟨*proof*⟩

lemma *last-stack-is-start*:
dfs-constructable dfs s $\Longrightarrow$ *stack s* $\neq$ [] $\Longrightarrow$ *last* (*stack s*) = *start s*
⟨*proof*⟩

lemma *start-discovered*:
dfs-constructable dfs s $\Longrightarrow$ *start s* $\in$ *discovered s*
⟨*proof*⟩

lemma *start-not-restr*:
dfs-constructable dfs s $\Longrightarrow$ *start s* $\notin$ *dfs-restrict dfs*
⟨*proof*⟩

lemma *dfs-start-in-verts*:
dfs-constructable dfs s $\Longrightarrow$ *start s* $\in$ *V*
⟨*proof*⟩

lemma *finished-stack-eq-discovered*:
dfs-constructable dfs s $\Longrightarrow$ *finished s* $\cup$ *set* (*stack s*) = *discovered s*
⟨*proof*⟩

lemma *finished-subset-discovered*:
dfs-constructable dfs s $\Longrightarrow$ *finished s* $\subseteq$ *discovered s*
⟨*proof*⟩

lemma *finished-subset-verts*:
dfs-constructable dfs s $\Longrightarrow$ *finished s* $\subseteq$ *V*
⟨*proof*⟩

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

lemma *ran-finish-finite*:
dfs-constructable dfs s $\Longrightarrow$ *finite* (*ran* (*finish s*))
⟨*proof*⟩

lemma *finished-implies-discovered*:
dfs-constructable dfs s $\Longrightarrow$ *v* $\in$ *finished s* $\Longrightarrow$ *v* $\in$ *discovered s*
⟨*proof*⟩

lemma *not-discovered-implies-not-finished*:
dfs-constructable dfs s $\Longrightarrow$ *v* $\notin$ *discovered s* $\Longrightarrow$ *v* $\notin$ *finished s*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *discovered-not-restricted*:
dfs-constructable dfs s $\Longrightarrow$ *v* $\in$ *discovered s* $\Longrightarrow$ *v* $\notin$ *dfs-restrict dfs*
⟨*proof*⟩

lemma *finished-not-restricted*:
dfs-constructable dfs s $\Longrightarrow$ *v* $\in$ *finished s* $\Longrightarrow$ *v* $\notin$ *dfs-restrict dfs*
⟨*proof*⟩

lemma *stack-not-restricted*:
dfs-constructable dfs s $\Longrightarrow$ *v* $\in$ *set* (*stack s*) $\Longrightarrow$ *v* $\notin$ *dfs-restrict dfs*
⟨*proof*⟩

lemma *restricted-not-finished*:
dfs-constructable dfs s $\Longrightarrow$ *v* $\in$ *dfs-restrict dfs* $\Longrightarrow$ *v* $\notin$ *finished s*
⟨*proof*⟩

lemma *restricted-not-discovered*:
dfs-constructable dfs s $\Longrightarrow$ *v* $\in$ *dfs-restrict dfs* $\Longrightarrow$ *v* $\notin$ *discovered s*
⟨*proof*⟩

lemma *restricted-not-stack*:
dfs-constructable dfs s $\Longrightarrow$ *v* $\in$ *dfs-restrict dfs* $\Longrightarrow$ *v* $\notin$ *set* (*stack s*)
⟨*proof*⟩

lemma *stack-implies-not-finished*:
dfs-constructable dfs s $\Longrightarrow$ *v* $\in$ *set* (*stack s*) $\Longrightarrow$ *v* $\notin$ *finished s*
⟨*proof*⟩

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

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*⟩

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*
⟨*proof*⟩

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*))
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

lemma *start-reachable-stack*:
assumes *constr*: *dfs-constructable dfs s*
and $x \in$ *set* (*stack s*)
shows *start s* $\rightarrow\star$ *x*
⟨*proof*⟩

lemma *ran-discover-lt-counter*:
dfs-constructable dfs s $\Longrightarrow$ $v \in$ *ran* (*discover s*) $\Longrightarrow$ $v <$ *counter s*
⟨*proof*⟩

lemma *discover-lt-counter*:
assumes *constr*: *dfs-constructable dfs s*
and *nn*: $v \in$ *discovered s*
shows δ *s v* $<$ *counter s*
⟨*proof*⟩

lemma *ran-finish-lt-counter*:
dfs-constructable dfs s $\Longrightarrow$ $v \in$ *ran* (*finish s*) $\Longrightarrow$ $v <$ *counter s*
⟨*proof*⟩

lemma *finish-lt-counter*:
assumes *constr*: *dfs-constructable dfs s*
and *nn*: $v \in$ *finished s*
shows $\varphi\ s\ v <$ *counter s*
⟨*proof*⟩

lemma *finish-gt-discover*:
dfs-constructable dfs s $\Longrightarrow v \in$ *finished s* $\Longrightarrow \varphi\ s\ v > \delta\ s\ v$
⟨*proof*⟩

lemma *discover-finish-distinct*:
dfs-constructable dfs s $\Longrightarrow$ *ran* (*finish s*) $\cap$ *ran* (*discover s*) $= \{\}$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *discover-card-ran-dom*:
dfs-constructable dfs s $\Longrightarrow$ *card* (*ran* (*discover s*)) $=$ *card* (*dom* (*discover s*))
⟨*proof*⟩

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*⟩

lemma *finish-card-ran-dom*:
dfs-constructable dfs s $\Longrightarrow$ *card* (*ran* (*finish s*)) $=$ *card* (*dom* (*finish s*))
⟨*proof*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

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*⟩

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*⟩

lemma *obtain-discovered-predecessor*:
assumes *dfs-constructable tdfs s*
and *v* ∈ *discovered s*
and *v* ≠ *start s*
obtains *y* **where** *v* ∈ *succs y* **and** *y* ∈ *discovered s* **and** *δ s v* > *δ s y*
⟨*proof*⟩

lemma *stack-finish-discover*:
dfs-constructable dfs s ⟹ *v* ∈ *set* (*stack s*) ⟹ *w* ∈ *finished s* ⟹ *φ s w* < *δ s v* ∨ *δ s v* < *δ s w*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*⟩

lemma *no-reach-discover-finish*:
assumes *constr*: *dfs-constructable dfs s*
and *no-reach*: $\neg v \rightarrow\star w$
shows *v* $\notin$ *discovered s* $\vee$ *w* $\notin$ *discovered s* $\vee$ *w* $\notin$ *finished s* $\vee\ \delta\ s\ v > \delta\ s\ w \vee$
(*v* $\in$ *finished s* $\wedge\ \varphi\ s\ v < \varphi\ s\ w$)
⟨*proof*⟩

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*⟩

lemma *rev-stack-vwalk-or-empty*:
assumes *dfs-constructable dfs s*
shows *vwalk* (*rev* (*stack s*)) $\mathcal{G} \vee$ *stack s* $= []$
⟨*proof*⟩

lemma *rev-stack-vwalk*:
dfs-constructable dfs s $\Longrightarrow$ *stack s* $\neq [] \Longrightarrow$ *vwalk* (*rev* (*stack s*)) $\mathcal{G}$
⟨*proof*⟩

lemma *discover-start-eq-0*:
dfs-constructable dfs s $\Longrightarrow \delta\ s$ (*start s*) $= 0$
⟨*proof*⟩

lemma *discovered-neq-0*:
assumes *constr*: *dfs-constructable dfs s*
and *discovered*: *v* $\in$ *discovered s*
and *ne*: *v* $\neq$ *start s*
shows $\delta\ s\ v > 0$
⟨*proof*⟩

lemma *discover-gt-start*:
dfs-constructable dfs s $\Longrightarrow$ *v* $\in$ *discovered s* $\Longrightarrow$ *v* $\neq$ *start s* $\Longrightarrow \delta\ s\ v > \delta\ s$
(*start s*)
⟨*proof*⟩

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 $start\ s \rightarrow \backslash dfs\text{-}restrict\ dfs+\ v$
⟨*proof*⟩

lemma *start-reach-discovered*:
$dfs\text{-}constructable\ dfs\ s \Longrightarrow stack\ s \neq [] \Longrightarrow v \in discovered\ s \Longrightarrow start\ s \rightarrow\star\ v$
⟨*proof*⟩

lemma *constr-from-restr-reach-stack*:
assumes *constr*: $s \in dfs\text{-}constr\text{-}from\ dfs\ x$
and *stack*: $v \in set\ (stack\ s)$
and *ne*: $v \neq x$
shows $x \rightarrow \backslash dfs\text{-}restrict\ dfs+\ v$
⟨*proof*⟩

lemma *constr-from-reach-stack*:
assumes *constr*: $s \in dfs\text{-}constr\text{-}from\ dfs\ x$
and *stack*: $v \in set\ (stack\ s)$
shows $x \rightarrow\star\ v$
⟨*proof*⟩

lemma *finished-implies-succs-discovered*:
$dfs\text{-}constructable\ dfs\ s \Longrightarrow v \in finished\ s \Longrightarrow succs\ v - dfs\text{-}restrict\ dfs \subseteq discovered\ s$
⟨*proof*⟩

lemma *finished-implies-succs-discoveredI*:
$[\![\ dfs\text{-}constructable\ dfs\ s;\ v \in finished\ s;\ w \in succs\ v;\ w \notin dfs\text{-}restrict\ dfs\ ]\!] \Longrightarrow w \in discovered\ s$
⟨*proof*⟩

lemma *finished-no-reach-implies-succs-finished*:
$dfs\text{-}constructable\ dfs\ s \Longrightarrow v \in finished\ s \Longrightarrow w \in succs\ v \Longrightarrow w \notin dfs\text{-}restrict\ dfs \Longrightarrow \neg\ w \rightarrow\star\ v \Longrightarrow w \in finished\ s$
⟨*proof*⟩

lemma *not-finished-succs-impl-not-finished*:
$dfs\text{-}constructable\ dfs\ s \Longrightarrow v \in discovered\ s \Longrightarrow w \in discovered\ s \Longrightarrow \delta\ s\ v < \delta\ s\ w \Longrightarrow w \notin finished\ s \Longrightarrow w \in succs\ v \Longrightarrow v \notin finished\ s$
⟨*proof*⟩

lemma *finished-succs-finish*:
$dfs\text{-}constructable\ dfs\ s \Longrightarrow v \in discovered\ s \Longrightarrow w \in finished\ s \Longrightarrow w \in succs\ v \Longrightarrow \delta\ s\ w > \delta\ s\ v \Longrightarrow v \notin finished\ s \vee \varphi\ s\ w < \varphi\ s\ v$
⟨*proof*⟩

lemma *finished-no-reach-nondiscovered*:
assumes *constr*: *dfs-constructable dfs s*
and *noreach*: $\forall\, x \in finished\ s.\ \forall\ \ y \in set(stack\ s).\ \neg\ x \rightarrow \backslash dfs\text{-}restrict\ dfs+\ y$

and *nondisc*: $z \notin$ *discovered s*
shows $\forall x \in$ *finished s*. $\neg$ *x* $\rightarrow\backslash$*dfs-restrict dfs+ z*
⟨*proof*⟩

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*⟩

lemma *reach-generate-sws* [*case-names step finished*, *consumes 1*]:
assumes *v* $\rightarrow\backslash$*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*⟩

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*
⟨*proof*⟩

lemma *dfs-finishedI*:
dfs-constructable dfs s ⟹ ¬(∃ *s*′. *dfs-next dfs s s*′) ⟹ *dfs-finished dfs s*
⟨*proof*⟩

lemma *dfs-finished-stackI*:
dfs-constructable dfs s ⟹ *stack s* = [] ⟹ *dfs-finished dfs s*
⟨*proof*⟩

lemma *dfs-finished-wlI*:
dfs-constructable dfs s ⟹ *wl s* = [] ⟹ *dfs-finished dfs s*
⟨*proof*⟩

lemma *dfs-finished-constructable*[*intro*]:

dfs-finished dfs s $\Longrightarrow$ *dfs-constructable dfs s*
⟨*proof*⟩

lemma *not-dfs-cond-implies-finished*:
dfs-constructable dfs s $\Longrightarrow$ $\neg$ *dfs-cond dfs* (*state s*) $\Longrightarrow$ *dfs-finished dfs s*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *finished-is-completed-or-not-cond*:
dfs-finished dfs s $\Longrightarrow$ *dfs-completed dfs s* $\vee$ $\neg$ *dfs-cond dfs* (*state s*)
⟨*proof*⟩

lemma *dfs-completed-constructable*[*dest*,*simp*]:
dfs-completed dfs s $\Longrightarrow$ *dfs-constructable dfs s*
⟨*proof*⟩

lemma *dfs-completedE*:
⟦ *dfs-completed dfs s*; ⟦ *dfs-finished dfs s*; *dfs-cond dfs* (*state s*) ⟧ $\Longrightarrow$ *P* ⟧ $\Longrightarrow$ *P*
⟨*proof*⟩

lemma *dfs-completed-empty*:
dfs-completed dfs s $\Longrightarrow$ *stack s* = [] $\vee$ *wl s* = []
⟨*proof*⟩

lemma *dfs-completed-empty-stack*:
dfs-completed dfs s $\Longrightarrow$ *stack s* = []
⟨*proof*⟩

lemma *dfs-completed-empty-wl*:
dfs-completed dfs s $\Longrightarrow$ *wl s* = []
⟨*proof*⟩

lemma *dfs-completed-stackI*:
dfs-constructable dfs s $\Longrightarrow$ *stack s* = [] $\Longrightarrow$ *dfs-cond dfs* (*state s*) $\Longrightarrow$ *dfs-completed dfs s*
⟨*proof*⟩

lemma *dfs-completed-wlI*:
dfs-constructable dfs s $\Longrightarrow$ *wl s* = [] $\Longrightarrow$ *dfs-cond dfs* (*state s*) $\Longrightarrow$ *dfs-completed dfs s*
⟨*proof*⟩

lemma *dfs-completed-revE* [*case-names prev*, *induct pred*: *dfs-completed*]:
assumes *finished*: *dfs-completed dfs s*

and *prev*: $\bigwedge s\ r.\ [\![\ \mathit{dfs\text{-}constructable}\ \mathit{dfs}\ r;\ \mathit{dfs\text{-}next}\ \mathit{dfs}\ r\ s;\ \mathit{dfs\text{-}completed}\ \mathit{dfs}\ s;\ s = r(\!|\ \mathit{stack} := [], \mathit{wl} := [], \mathit{finish} := (\mathit{finish}\ r)(\mathit{hd}\ (\mathit{stack}\ r) \mapsto \mathit{counter}\ r), \mathit{counter} := \mathit{Suc}\ (\mathit{counter}\ r), \mathit{state} := \mathit{dfs\text{-}post}\ \mathit{dfs}\ (\mathit{state}\ r)\ r\ (\mathit{hd}\ (\mathit{stack}\ r))|)\ ]\!] \Longrightarrow P\ s$
shows $P\ s$
⟨*proof*⟩

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 $\mathit{stack}\ r = [\mathit{start}\ s]$
⟨*proof*⟩

lemma *completed-start-finished*:
$\mathit{dfs\text{-}completed}\ \mathit{dfs}\ s \Longrightarrow \mathit{start}\ s \in \mathit{finished}\ s$
⟨*proof*⟩

lemma *completed-start-finish-eq-counter*:
$\mathit{dfs\text{-}completed}\ \mathit{dfs}\ s \Longrightarrow \varphi\ s\ (\mathit{start}\ s) = \mathit{counter}\ s - 1$
⟨*proof*⟩

lemma *completed-finish-lt-start*:
$\mathit{dfs\text{-}completed}\ \mathit{dfs}\ s \Longrightarrow v \in \mathit{finished}\ s \Longrightarrow v \neq \mathit{start}\ s \Longrightarrow \varphi\ s\ v < \varphi\ s\ (\mathit{start}\ s)$
⟨*proof*⟩

lemma *completed-finished-eq-discovered*:
$\mathit{dfs\text{-}completed}\ \mathit{dfs}\ s \Longrightarrow \mathit{finished}\ s = \mathit{discovered}\ s$
⟨*proof*⟩

lemma *completed-start-finished-restr-reach*:
assumes *compl*: *dfs-completed dfs s*
and *finished*: $v \in \mathit{finished}\ s$
and *ne*: $v \neq \mathit{start}\ s$
shows $\mathit{start}\ s \rightarrow \backslash \mathit{dfs\text{-}restrict}\ \mathit{dfs}+\ v$
⟨*proof*⟩

lemma *completed-start-finished-reach*:
assumes *compl*: *dfs-completed dfs s*
and *finished*: $v \in \mathit{finished}\ s$
shows $\mathit{start}\ s \rightarrow\star\ v$
⟨*proof*⟩

lemma *completed-reach-implies-finished*:
assumes *compl*: *dfs-completed dfs s*
and *fin*: $v \in \mathit{finished}\ s$
and *reach*: $v \rightarrow \backslash \mathit{dfs\text{-}restrict}\ \mathit{dfs}+\ w$
shows $w \in \mathit{finished}\ s$
⟨*proof*⟩

lemma *completed-start-reach-finished*:
dfs-completed dfs s $\Longrightarrow$ *start s* $\rightarrow\backslash$*dfs-restrict dfs*+ *v* $\Longrightarrow$ *v* $\in$ *finished s*
⟨*proof*⟩

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*⟩

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*⟩

lemma *completed-finished-succs-finish*:
assumes *compl*: *dfs-completed dfs s*
and *finished*: *v* $\in$ *finished s*
and *d-lt*: δ *s v* < δ *s w*
and *succs*: *w* $\in$ *succs v*
and *w* $\notin$ *dfs-restrict dfs*
shows φ *s w* < φ *s v*
⟨*proof*⟩

lemma *dfs-finished-cases* [*case-names start step*, *consumes 1*]:
assumes *fin*: *dfs-finished dfs s*
and *start*: ¬ *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*⟩
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*
⟨*proof*⟩

lemma *dfs-invar-compl-constr-from*[*dest*]:
dfs-invar-compl dfs x s ⟹ *s* ∈ *dfs-constr-from dfs x*
⟨*proof*⟩

lemma *dfs-invar-compl-constr*[*dest*]:
dfs-invar-compl dfs x s ⟹ *dfs-constructable dfs s*
⟨*proof*⟩

lemma *dfs-invar-complE*:
⟦ *dfs-invar-compl dfs x s*; ⟦ *s* ∈ *dfs-constr-from dfs x*; *dfs-invar dfs s* ⟧ ⟹ *P* ⟧ ⟹ *P*
⟨*proof*⟩

lemma *dfs-invar-complI*:
⟦ *s* ∈ *dfs-constr-from dfs x*; *dfs-invar dfs s* ⟧ ⟹ *dfs-invar-compl dfs x s*
⟨*proof*⟩

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* ⇒ '*n* ⇒ ('*S*,'*n*) *dfs-sws nres* **where**
dfs-fun dfs x ≡
$WHILE_T^{(dfs\text{-}invar\text{-}compl\ dfs\ x)}$ (*dfs-cond-compl dfs*) (λ *s*. *SPEC* (λ*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* ⇒ *bool*
where
dfs-preserves-invar dfs ≡
(∀ *x* ∈ *V*. *dfs-invar dfs* (*dfs-constr-start dfs x*)) ∧
(∀ *x s s*'. *dfs-invar-compl dfs x s* ∧ *dfs-next dfs s s*' ⟶
dfs-invar dfs s')

lemma *dfs-preserves-invarI*:
⟦⋀*x s s'*. ⟦ *dfs-invar-compl dfs x s*; *dfs-next dfs s s'* ⟧ ⟹ *dfs-invar dfs s'*; ⋀*x*. *x* ∈ *V* ⟹ *dfs-invar dfs* (*dfs-constr-start dfs x*) ⟧
⟹ *dfs-preserves-invar dfs*
⟨*proof*⟩

lemma *dfs-preserves-invarD*[*intro*]:
⟦ *dfs-preserves-invar dfs*; *dfs-invar-compl dfs x s*; *dfs-next dfs s s'* ⟧ ⟹ *dfs-invar dfs s'*
⟨*proof*⟩

lemma *dfs-preserves-invarE*[*elim*]:
assumes *dfs-preserves-invar dfs*
and ⟦⋀*x s s'*. ⟦ *dfs-invar-compl dfs x s*; *dfs-next dfs s s'* ⟧ ⟹ *dfs-invar dfs s'* ⟧ ⟹ *P*
shows *P*
⟨*proof*⟩

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'*
⟨*proof*⟩

lemma *constr-from-invar-compl*:
s ∈ *dfs-constr-from dfs x* ⟹ *dfs-preserves-invar dfs* ⟹ *dfs-invar-compl dfs x s*
⟨*proof*⟩

lemma *constr-from-invarI*[*intro*!]:
s ∈ *dfs-constr-from dfs x* ⟹ *dfs-preserves-invar dfs* ⟹ *dfs-invar dfs s*
⟨*proof*⟩

lemma *dfs-constructable-invarI*[*intro*!]:
dfs-constructable dfs s ⟹ *dfs-preserves-invar dfs* ⟹ *dfs-invar dfs s*
⟨*proof*⟩

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 ≡ *less-than* <∗*lex*∗> *less-than* <∗*lex*∗> *less-than*

lemma *wf-dfs-step-measure*:
wf dfs-step-measure
⟨*proof*⟩

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*
⟨*proof*⟩

theorem *wf-ws-rel*:
wf ws-rel
⟨*proof*⟩

lemma *ws-rel-intro*:
(*ws-to-measure ws*', *ws-to-measure ws*) ∈ *dfs-step-measure* ⟹ (*ws*', *ws*) ∈ *ws-rel*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

theorem *dfs-fun-correct*:
assumes *dfs-preserves-invar dfs*
and *x* ∈ *V x* ∉ *dfs-restrict dfs*
shows *dfs-fun dfs x* ≤ *SPEC* (λ *s*. *s* ∈ *dfs-constr-from dfs x* ∧ *dfs-finished dfs s*)
⟨*proof*⟩

lemma *dfs-fun-nofail*[*refine-pw-simps*]:
assumes *dfs-preserves-invar dfs*
and *x* ∈ *V x* ∉ *dfs-restrict dfs*

shows *nofail* (*dfs-fun dfs x*)
⟨*proof*⟩

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*⟩

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* ∈ *dfs-constr-from dfs x*
⟨*proof*⟩

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*)
⟨*proof*⟩

schematic-lemma *dfs-fun-code*:
RETURN ?f ≤ *dfs-fun dfs x*
⟨*proof*⟩

lemma *dfs-fun-nonempty*:
∃ *s*. *inres* (*dfs-fun dfs x*) *s*
⟨*proof*⟩

lemma *dfs-fun-pred*:
assumes *x*: $x \in V\ x \notin$ *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*)
⟨*proof*⟩

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 = {}
⟨*proof*⟩

lemma (**in** *finite-digraph*) *simple-dfs-preserves-invar*:
dfs-preserves-invar simple-dfs
⟨*proof*⟩
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
⟨*proof*⟩

lemma *dfs-next-edges-subset*:
dfs-next tdfs s s' ⟹ *edges s* ⊆ *edges s'*
⟨*proof*⟩

lemma *no-self-edges*:
dfs-constructable tdfs s ⟹ (*v*, *w*) ∈ *edges s* ⟹ *v* ≠ *w*
⟨*proof*⟩

lemma *stack-tl-subset-edges*:
dfs-constructable tdfs s ⟹ *stack s* ≠ [] ⟹ *set* (*tl*(*stack s*)) ⊆ *Field* (*edges s*)
⟨*proof*⟩

lemma *stack-hd-in-edges*:
dfs-constructable tdfs s ⟹ *edges s* ≠ {} ⟹ *stack s* ≠ [] ⟹ *hd* (*stack s*) ∈ *Field* (*edges s*)
⟨*proof*⟩

lemma *stack-subset-edges*:
assumes *constr*: *dfs-constructable tdfs s*
and *ne*: *edges s* ≠ {}
shows *set* (*stack s*) ⊆ *Field* (*edges s*)
⟨*proof*⟩

lemma *stack-gt-1-implies-edges*:
assumes *dfs-constructable tdfs s*
and *length* (*stack s*) > *1*
shows *edges s* ≠ {}
⟨*proof*⟩

lemma *edges-stack-is-discovered*:
assumes *dfs-constructable tdfs s*
and *stack s* ≠ []
shows *Field* (*edges s*) ∪ *set* (*stack s*) = *discovered s*
⟨*proof*⟩

lemma *completed-edges-eq-discovered*:
assumes *dfs-completed tdfs s*

and *edges s* $\neq$ {}
shows *Field* (*edges s*) = *discovered s*
⟨*proof*⟩

lemma *edges-subset-discovered*:
assumes *dfs-constructable tdfs s*
shows *Field* (*edges s*) $\subseteq$ *discovered s*
⟨*proof*⟩

lemma *etrancl-is-discovered*:
assumes *constr*: *dfs-constructable tdfs s*
and *etrancl*: (*v*,*w*) $\in$ (*edges s*)$^+$
shows *v* $\in$ *discovered s* **and** *w* $\in$ *discovered s*
⟨*proof*⟩

lemma *edges-subset-E*:
dfs-constructable tdfs s $\Longrightarrow$ *edges s* $\subseteq$ *E*
⟨*proof*⟩

lemma *edge-is-succs*:
assumes *dfs-constructable tdfs s*
and (*v*, *w*) $\in$ *edges s*
shows *w* $\in$ *succs v*
⟨*proof*⟩

lemma *discover-edge*:
dfs-constructable tdfs s $\Longrightarrow$ (*v*,*w*) $\in$ *edges s* $\Longrightarrow$ δ *s v* < δ *s w*
⟨*proof*⟩

lemma *discover-etrancl*:
assumes *constr*: *dfs-constructable tdfs s*
and *reach*: (*v*,*w*) $\in$ (*edges s*)$^+$
shows δ *s v* < δ *s w*
⟨*proof*⟩

lemma *no-cycle*:
dfs-constructable tdfs s $\Longrightarrow$ (*v*,*w*) $\in$ (*edges s*)$^+$ $\Longrightarrow$ *v* $\neq$ *w*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*)$^+$
⟨*proof*⟩

lemma *hd-stack-no-cycle*:
assumes *constr*: *dfs-constructable tdfs s*
and *edge*: (*hd* (*stack s*), *w*) $\in$ (*edges s*)$^+$
and *ne*: *stack s* $\neq$ []
shows *w* $\notin$ *set* (*stack s*)
⟨*proof*⟩

lemma *no-edge-to-stack*:
dfs-constructable tdfs s $\Longrightarrow$ (*v*, *w*) $\in$ *edges s* $\Longrightarrow$ *v* $\notin$ *set* (*stack s*) $\Longrightarrow$ *w* $\notin$ *set* (*stack s*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

lemma *finish-edge*:
assumes *constr*: *dfs-constructable tdfs s*
and *edge*: (*v*,*w*) $\in$ *edges s*
and *fin*: *v* $\in$ *finished s*
shows φ *s v* > φ *s w*
⟨*proof*⟩

lemma *finish-etrancl*:
assumes *constr*: *dfs-constructable tdfs s*
and *reach*: (*v*,*w*) $\in$ (*edges s*)$^+$
and *fin*: *v* $\in$ *finished s*
shows φ *s v* > φ *s w*
⟨*proof*⟩

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*⟩

lemma *discover-finish-implies-in-etrancl*:
assumes *dfs-constructable tdfs s*
and $v \in$ *discovered s* **and** $w \in$ *discovered s*
and $\delta\ s\ v < \delta\ s\ w$ **and** $v \notin$ *finished s* $\vee$ $(w \in$ *finished s* $\wedge\ \varphi\ s\ w < \varphi\ s\ v)$
shows $(v,w) \in (edges\ s)^+$
⟨*proof*⟩

lemma *not-in-etrancl-discover-finish*:
assumes *constr*: *dfs-constructable tdfs s*
and *ne*: $v \neq w$
and *not-in-etrancl*: $(v,w) \notin (edges\ s)^+$
shows $v \notin$ *discovered s* $\vee\ w \notin$ *discovered s* $\vee\ w \notin$ *finished s* $\vee\ \delta\ s\ v > \delta\ s\ w \vee$
$(v \in$ *finished s* $\wedge\ \varphi\ s\ v < \varphi\ s\ w)$
⟨*proof*⟩

lemma *both-not-in-etrancl-discover-finish*:
assumes *constr*: *dfs-constructable tdfs s*
and *ne*: $v \neq w$
and *not-in-etrancl*: $(v,w) \notin (edges\ s)^+\ (w,v) \notin (edges\ s)^+$
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*⟩

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** $E\ E_G :: ('n \times 'n)\ set$ +
fixes $s :: ('n\ tree, 'n)\ dfs\text{-}sws$
assumes *E-edges*: $E = edges\ s$
and *E-ne*: $E \neq \{\}$
and *V-discovered*: $V =$ *discovered s*
and *completed*: *dfs-completed tdfs s*
begin

abbreviation *G-fd-to-graph* $(\mathcal{G}_G)$ **where**
G-fd-to-graph ≡ *G.fd-to-graph*
abbreviation *G-fd-reachable* (- $\rightarrow_{G\star}$ - [*100*,*100*] *40*) **where**
G-fd-reachable ≡ *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*:
  $v \rightarrow d+ \; w \longleftrightarrow (v,w) \in (edges \; s)^+$
⟨*proof*⟩

lemma *constructable*:
  dfs-constructable tdfs s
⟨*proof*⟩

lemma *dfs-edges-sub*:
  $E \subseteq E_G$
⟨*proof*⟩

lemma *dfs-vertices-sub*:
  $V \subseteq V_G$
⟨*proof*⟩

lemma *verts-eq-edges*:
  $V = Field \; E$
⟨*proof*⟩

lemma *s-discovered*[*simp*, *intro*]:
  $x \in V \Longrightarrow x \in discovered \; s$
⟨*proof*⟩

lemma *s-finished*[*simp*,*intro*]:
  $x \in V \Longrightarrow x \in finished \; s$
⟨*proof*⟩

lemma *vert-reach-implies-vert*:
  assumes $x \in V$
  and *reach*: $x \rightarrow G+ \; y$
  shows $y \in V$
⟨*proof*⟩

lemma *vert-ewalk-ewalk-verts*:
  assumes *v*: $x \in V$
  and *walk*: *ewalk x p y*
  shows *set* (*ewalk-verts x p*) $\subseteq V$
⟨*proof*⟩

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*⟩
end

sublocale *dfs-run* ⊆ *finite-subgraph* V_G V E_G E
⟨*proof*⟩

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*:
$v \in V \Longrightarrow \delta\ v < \varphi\ v$
⟨*proof*⟩

lemma *no-self-reach*:
$v \rightarrow d+\ w \Longrightarrow v \neq w$
⟨*proof*⟩

lemma *correct-order*:
$v \in V \Longrightarrow w \in V \Longrightarrow \delta\ v < \delta\ w \Longrightarrow\ \varphi\ v < \delta\ w \vee \varphi\ v > \varphi\ w$
⟨*proof*⟩

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*⟩

lemma *discover-neq-finish*:
$v \in V \Longrightarrow w \in V \Longrightarrow \delta\ v \neq \varphi\ w$
⟨*proof*⟩

lemma *discover-neq-discover*:
$v \in V \Longrightarrow w \in V \Longrightarrow v \neq w \Longrightarrow \delta\ v \neq \delta\ w$
⟨*proof*⟩

lemma *finish-neq-finish*:
$v \in V \Longrightarrow w \in V \Longrightarrow v \neq w \Longrightarrow \varphi\ v \neq \varphi\ w$
⟨*proof*⟩

lemma *reach-discover*:
$v \rightarrow d{+}\ w \Longrightarrow \delta\ v < \delta\ w$
⟨*proof*⟩

lemma *reach-finish*:
$v \rightarrow d{+}\ w \Longrightarrow \varphi\ v > \varphi\ w$
⟨*proof*⟩

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*⟩

lemma *discover-finish-implies-reach*:
$v \in V \Longrightarrow w \in V \Longrightarrow \delta\ v < \delta\ w \Longrightarrow \varphi\ v > \varphi\ w \Longrightarrow v \rightarrow d{+}\ w$
⟨*proof*⟩

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*⟩

lemma *both-no-reach-discover-finish*:
$v \in V \Longrightarrow w \in V \Longrightarrow \neg\ v \rightarrow d\star\ w \Longrightarrow \neg\ w \rightarrow d\star\ v \Longrightarrow \delta\ w > \varphi\ v \vee \varphi\ w < \delta\ v$
⟨*proof*⟩

lemma *treeish*:
assumes *reach*: $v \rightarrow d{+}\ w$
shows $\neg\ w \rightarrow d\star\ v$
⟨*proof*⟩

lemma *discover-finish-impl-disj*:
assumes $v \in V\ w \in V$
and $\delta\ v < \delta\ w\ \varphi\ v < \varphi\ w$
shows $\varphi\ v < \delta\ w$
⟨*proof*⟩

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*⟩

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

definition *WHITE u x* ≡ δ *x* ≥ *u*
definition *GREY u x* ≡ δ *x* < *u* ∧ φ *x* ≥ *u*
definition *BLACK u x* ≡ φ *x* < *u*

lemma *single-color*:
  assumes *x* ∈ *V*
  shows *WHITE u x* ≠ (*GREY u x* ∨ *BLACK u x*)
  and *GREY u x* ≠ (*WHITE u x* ∨ *BLACK u x*)
  and *BLACK u x* ≠ (*WHITE u x* ∨ *GREY u x*)
⟨*proof*⟩

lemma *grey-impl-reach*:
  assumes *verts*: *v* ∈ *V w* ∈ *V*
  and *grey*: *GREY u v GREY u w*
  shows *v* →*d*⋆ *w* ∨ *w* →*d*⋆ *v*
⟨*proof*⟩

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

definition *DISJ v w* ≡ (φ *v* < δ *w* ∨ φ *w* < δ *v*) ∧ ¬ *v* →*d*⋆ *w* ∧ ¬ *w* →*d*⋆ *v*
definition *IN v w* ≡ (δ *v* < δ *w* ∧ φ *v* > φ *w*) ∧ *v* →*d*+ *w*

lemma *reach-eq-IN*:
  v →*d*+ *w* ⟷ *IN v w*
⟨*proof*⟩

lemma *not-reach-eq-DISJ*:
  v ∈ *V* ⟹ *w* ∈ *V* ⟹ ¬ *v* →*d*⋆ *w* ∧ ¬ *w* →*d*⋆ *v* ⟷ *DISJ v w*
⟨*proof*⟩

theorem *parenthesis*:
  assumes *verts*: *v* ∈ *V w* ∈ *V*
  and *v* ≠ *w*
  shows *DISJ v w* ≠ (*IN v w* ∨ *IN w v*)
  and *IN v w* ≠ (*DISJ v w* ∨ *IN w v*)
⟨*proof*⟩

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 $\longleftrightarrow$ ($\exists$ *p*. *G.ewalk x p y* $\wedge$ ($\forall$ *v* $\in$ *set* (*ewalk-verts x p*). *WHITE* (*δ x*) *v*))
⟨*proof*⟩

lemma *reach-impl-white-path*:
assumes *x* $\rightarrow$*d*$\star$ *y*
shows *white-path x y*
⟨*proof*⟩

lemma *white-path-impl-reach*:
assumes *wp*: *white-path x y*
and *x* $\in$ *V*
shows *x* $\rightarrow$*d*$\star$ *y*
⟨*proof*⟩

theorem *white-path*:
x $\in$ *V* $\Longrightarrow$ *white-path x y* $\longleftrightarrow$ *x* $\rightarrow$*d*$\star$ *y*
⟨*proof*⟩

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

context *finite-digraph*
begin

fun *check-cycle* :: *'n set* $\Rightarrow$ *bool* $\Rightarrow$ (*bool*, *'n*) *dfs-sws* $\Rightarrow$ *'n* $\Rightarrow$ *bool* **where**
check-cycle - True - - = True
| *check-cycle sup-st False - e* = (*e* $\in$ *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 $\equiv$ $\neg$ *state s* $\longrightarrow$ (*finished s* $\cap$ *sup-st* $\subseteq$ {*start s*} $\wedge$
set (*stack s*) $\cap$ *sup-st* $\subseteq$ {*start s*} $\wedge$
(*start s* $\in$ *sup-st* $\longrightarrow$ (($\forall$ *x* $\in$ *finished s*. *start s* $\notin$ *succs x*) $\wedge$
($\forall$ *n* < *length* (*stack s*). *start s* $\notin$ *wl s* ! *n* $\longrightarrow$ *start s* $\notin$ *succs* (*stack s* ! *n*)))))

lemma *sub-dfs-invarI*:
⟦ $\neg$ *state s*; *finished s* $\cap$ *sup-st* $\subseteq$ {*start s*}; *set* (*stack s*) $\cap$ *sup-st* $\subseteq$ {*start s*}; *start s* $\in$ *sup-st* $\Longrightarrow$ $\forall$ *x* $\in$ *finished s*. *start s* $\notin$ *succs x*; *start s* $\in$ *sup-st* $\Longrightarrow$ $\forall$ *n* < *length* (*stack s*). *start s* $\notin$ *wl s* ! *n* $\longrightarrow$ *start s* $\notin$ *succs* (*stack s* ! *n*) ⟧ $\Longrightarrow$ *sub-dfs-invar sup-st s*
⟨*proof*⟩

definition *sub-dfs* :: *'n set* $\Rightarrow$ *'n set* $\Rightarrow$ (*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*
⟨*proof*⟩

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

lemma *sub-dfs-preserves-invar*[*intro*!]:
dfs-preserves-invar (*sub-dfs ss R*)
⟨*proof*⟩

definition *sub-cyclic* **where** *sub-cyclic ss s* ≡
stack s ≠ [] ∧
((*hd* (*stack s*) ∈ *ss* ∧ *start s* ≠ *hd* (*stack s*)) ∨ (*start s* ∈ *succs* (*hd* (*stack s*)) ∧ *start s* ∈ *ss*)) ∧
state s
definition *sub-cycle* **where** *sub-cycle ss R x* ≡ ∃ *s* ∈ *dfs-constr-from* (*sub-dfs ss R*) *x*. *sub-cyclic ss s*

lemma *state-implies-sub-cyclic*:
dfs-constructable (*sub-dfs ss R*) *s* ⟹ *state s* ⟹ *sub-cyclic ss s*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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

lemma *dfs-fun-sub-dfs-correct*:
assumes $x \in V\ x \notin R$
shows $dfs\text{-}fun\ (sub\text{-}dfs\ ss\ R)\ x \leq SPEC\ (\lambda s.\ state\ s \longleftrightarrow sub\text{-}cycle\ ss\ R\ x)$
⟨*proof*⟩

theorem *sub-dfs-correct*:
assumes $x \in V\ x \notin R$
and $inres\ (dfs\text{-}fun\ (sub\text{-}dfs\ ss\ R)\ x)\ s$
shows $state\ s \longleftrightarrow sub\text{-}cycle\ ss\ R\ x$
⟨*proof*⟩

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 $inres\ (dfs\text{-}fun\ (sub\text{-}dfs\ ss\ R)\ x)\ s$
shows $dfs\text{-}completed\ (sub\text{-}dfs\ ss\ R)\ s \longleftrightarrow \neg\ sub\text{-}cycle\ ss\ R\ x$
⟨*proof*⟩

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 $RETURN\ (sub\text{-}cycle\ ss\ R\ x) = do\ \{\ s \leftarrow dfs\text{-}fun\ (sub\text{-}dfs\ ss\ R)\ x;$
$RETURN\ (state\ s)\ \}$
⟨*proof*⟩

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 $RETURN\ (if\ sub\text{-}cycle\ ss\ R\ x\ then\ None\ else\ Some\ \{v.\ v = x \vee x \rightarrow \backslash R+$
$v\}) =$
$do\ \{\ s \leftarrow dfs\text{-}fun\ (sub\text{-}dfs\ ss\ R)\ x;\ RETURN\ (if\ state\ s\ then\ None\ else$
$Some\ (finished\ s))\ \}$ (**is** $?L = ?R$)
⟨*proof*⟩

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* ⟷ (∃ *v*. *e* →\ *R*+ *v* ∧ *v* ∈ *ss*)
⟨*proof*⟩

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*⟩

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* ∈ $\mathcal{A}$ *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*:
  $e \notin \mathcal{A} \Longrightarrow$ *run-sub-dfs M S s e* = *S*
⟨*proof*⟩

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 $e \in V$ $e \notin$ *snd S*
  and *same*: ⟦ $e \notin \mathcal{A}$; *run-sub-dfs M S s e* = *S*⟧ ⟹ *P*
  and *cycle*: ⋀*v*. ⟦ $e \in \mathcal{A}$; *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'*. ⟦ $e \in \mathcal{A}$; *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*⟩

definition *nested-dfs-invar* **where**

nested-dfs-invar s ≡ (¬ *has-cycle s* ⟶
(∀ *x* ∈ $\mathcal{A}$ ∩ *finished s*. ¬ *x* →+ *x*) ∧
set (*stack s*) ∩ *snd* (*state s*) = {} ∧
(∀ *x*. *x* ∉ *discovered s* ⟶ *x* ∉ *snd* (*state s*)) ∧
(∀ *x y*. *x* →+ *y* ⟶ *x* ∈ *snd* (*state s*) ⟶ *y* ∈ *snd* (*state s*)))

lemma *nested-dfs-invarI*:
⟦ ¬ *has-cycle s*;
∀ *x* ∈ $\mathcal{A}$ ∩ *finished s*. ¬ *x* →+ *x*;
set (*stack s*) ∩ *snd* (*state s*) = {};
⋀*x*. *x* ∉ *discovered s* ⟹ *x* ∉ *snd* (*state s*);
⋀*x y*. *x* →+ *y* ⟹ *x* ∈ *snd* (*state s*) ⟹ *y* ∈ *snd* (*state s*)⟧ ⟹ *nested-dfs-invar s*
⟨*proof*⟩

definition *nested-dfs* :: ((*bool* × 'n *set*, 'n) *dfs-sws* ⇒ 'n *set*) ⇒ (*bool* × 'n *set*, 'n) *dfs-algorithm-invar* **where**
nested-dfs M = (| *dfs-cond* = *Not* ∘ *fst*, *dfs-action* = λ *S* - -. *S*, *dfs-post* = *run-sub-dfs M*, *dfs-remove* = λ *S* - -. *S*, *dfs-start* = λ*x*. (*False*, {}), *dfs-restrict* = {}, *dfs-invar* = *nested-dfs-invar* |)

lemma *nested-dfs-simps*[*simp*]:
dfs-cond (*nested-dfs M*) *S* ⟷ ¬ *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*) = {}
⟨*proof*⟩

lemma *nested-dfs-has-cycle-not-next*:
dfs-constructable (*nested-dfs M*) *s* ⟹ *has-cycle s* ⟹ ¬ *dfs-next* (*nested-dfs M*) *s s'*
⟨*proof*⟩

lemma *ewalk-verts-vwalk-to-ewalk*:
xs ≠ [] ⟹ *vwalk xs* $\mathcal{G}$ ⟹ *hd xs* = *u* ⟹ *ewalk-verts u* (*vwalk-to-ewalk xs*) = *xs*
⟨*proof*⟩

lemma *vwalk-restr-reachable1-intro*:
assumes *vwalk xs* $\mathcal{G}$
and *y* ∈ *set xs*
and *set xs* ∩ *R* = {}
and *y* ≠ *last xs*
shows *y* →*R*+ *last xs*
⟨*proof*⟩

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*⟩

definition *cyclic* **where** *cyclic x s* ≡ *has-cycle s* ∧ (∃ *v* ∈ *finished s* ∩ $\mathcal{A}$. *v* →+ *v* ∧ *x* →⋆ *v*)
definition *cycle* **where** *cycle M x* ≡ ∃ *s* ∈ *dfs-constr-from* (*nested-dfs M*) *x*. *cyclic x s*

lemma *has-cycle-implies-cyclic*:
dfs-constructable (*nested-dfs M*) *s* ⟹ (⋀*s*. *stack s* ≠ [] ⟹ *hd* (*stack s*) ∈ *M s*) ⟹ (⋀*s*. *stack s* ≠ [] ⟹ *M s* ⊆ *set* (*stack s*)) ⟹ *has-cycle s* ⟹ *cyclic* (*start s*) *s*
⟨*proof*⟩

lemma *cycle-not-completed*:
assumes *M-hd*: ⋀*s*. *stack s* ≠ [] ⟹ *hd* (*stack s*) ∈ *M s*
and *reach*: *x* →⋆ *v*
and *cycle*: *v* →+ *v*
and *fin*: *v* ∈ $\mathcal{A}$
and *constr*: *s* ∈ *dfs-constr-from* (*nested-dfs M*) *x*
shows ¬ *dfs-completed* (*nested-dfs M*) *s*
⟨*proof*⟩

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* ∈ $\mathcal{A}$
obtains *s* **where** *s* ∈ *dfs-constr-from* (*nested-dfs M*) *x cyclic x s*
⟨*proof*⟩

lemma *cycle-get-cycle*:
assumes *cycle M x*
obtains *v* **where** *x* →⋆ *v v* ∈ $\mathcal{A}$ *v* →+ *v*
⟨*proof*⟩

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* ⟷ (∃ *v*. *x* →⋆ *v* ∧ *v* ∈ $\mathcal{A}$ ∧ *v* →+ *v*)
⟨*proof*⟩

lemma *dfs-fun-nested-dfs-correct*:
assumes *x* ∈ *V*
and *M-correct*: ⋀*s*. *stack s* ≠ [] ⟹ *hd* (*stack s*) ∈ *M s* ⋀*s*. *stack s* ≠ [] ⟹ *M s* ⊆ *set* (*stack s*)

shows *dfs-fun* (*nested-dfs M*) *x* ≤ *SPEC* (λ*s*. *has-cycle s* ⟷ *cycle M x*)
⟨*proof*⟩

theorem *nested-dfs-correct*:
assumes *x* ∈ *V*
and ⋀*s*. *stack s* ≠ [] ⟹ *hd* (*stack s*) ∈ *M s* ⋀*s*. *stack s* ≠ [] ⟹ *M s* ⊆ *set* (*stack s*)
and *inres* (*dfs-fun* (*nested-dfs M*) *x*) *s*
shows *has-cycle s* ⟷ *cycle M x*
⟨*proof*⟩

lemma *RETURN-nested-cycle-correct*:
assumes *x* ∈ *V*
and ⋀*s*. *stack s* ≠ [] ⟹ *hd* (*stack s*) ∈ *M s* ⋀*s*. *stack s* ≠ [] ⟹ *M s* ⊆ *set* (*stack s*)
shows *RETURN* (*cycle M x*) = *do* { *s* ← *dfs-fun* (*nested-dfs M*) *x*; *RETURN* (*has-cycle s*) }
⟨*proof*⟩

end

2.3 Basic Nested DFS

context *finite-accepting-digraph*
begin

definition *basicM* ≡ λ*s*. { *hd* (*stack s*) }
definition *basic-dfs* ≡ *nested-dfs basicM*

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

lemma *basicM-correct*[*simp*]:
stack s ≠ [] ⟹ *hd* (*stack s*) ∈ *basicM s*
stack s ≠ [] ⟹ *basicM s* ⊆ *set* (*stack s*)
⟨*proof*⟩

lemma *basic-dfs-preserves-invar*:
dfs-preserves-invar basic-dfs
⟨*proof*⟩

lemma *basic-cycle-iff-cycle*:
cycle basicM x ⟷ (∃ *v*. *x* →⋆ *v* ∧ *v* ∈ $\mathcal{A}$ ∧ *v* →+ *v*)
⟨*proof*⟩

lemma *dfs-fun-basic-dfs-correct*:
x ∈ *V* ⟹ *dfs-fun basic-dfs x* ≤ *SPEC* (λ*s*. *has-cycle s* ⟷ *cycle basicM x*)
⟨*proof*⟩

theorem *basic-dfs-correct*:

$x \in V \Longrightarrow$ *inres* (*dfs-fun basic-dfs x*) *s* $\Longrightarrow$ *has-cycle s* $\longleftrightarrow$ *cycle basicM x*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *hpy-dfs-preserves-invar*:
dfs-preserves-invar hpy-dfs
⟨*proof*⟩

lemma *hpy-cycle-iff-cycle*:
cycle hpyM x $\longleftrightarrow$ ($\exists\, v.\ x \rightarrow\star\ v \wedge v \in \mathcal{A} \wedge v \rightarrow+\ v$)
⟨*proof*⟩

lemma *dfs-fun-hpy-dfs-correct*:
$x \in V \Longrightarrow$ *dfs-fun hpy-dfs x* $\leq$ *SPEC* ($\lambda s.$ *has-cycle s* $\longleftrightarrow$ *cycle hpyM x*)
⟨*proof*⟩

theorem *hpy-dfs-correct*:
$x \in V \Longrightarrow$ *inres* (*dfs-fun hpy-dfs x*) *s* $\Longrightarrow$ *has-cycle s* $\longleftrightarrow$ *cycle hpyM x*
⟨*proof*⟩

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* × *'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 ⟷ ¬ *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 = {}
⟨*proof*⟩

lemma *SE-dfs-has-cycle-not-next*:
dfs-constructable SE-dfs s ⟹ *has-cycle s* ⟹ ¬ *dfs-next SE-dfs s s'*
⟨*proof*⟩

lemma *SE-to-nested*:
s ∈ *dfs-constr-from SE-dfs x* ⟹ ¬ *has-cycle s* ⟹ *s* ∈ *dfs-constr-from hpy-dfs x*
⟨*proof*⟩

corollary *SE-to-nested'*:
dfs-constructable SE-dfs s ⟹ ¬ *has-cycle s* ⟹ *dfs-constructable hpy-dfs s*
⟨*proof*⟩

lemma *SE-dfs-preserves-invar*[*intro*]:
dfs-preserves-invar SE-dfs
⟨*proof*⟩

definition *SE-cyclic* **where** *SE-cyclic s* ≡ *has-cycle s* ∧ (*cyclic* (*start s*) *s* ∨ (∃ *x* ∈ *set* (*stack s*). *x* ∈ 𝒜 ∧ *x* →+ *x*))
definition *SE-cycle* **where** *SE-cycle x* ≡ ∃ *s*. *s* ∈ *dfs-constr-from SE-dfs x* ∧ *SE-cyclic s*

lemma *has-cycle-implies-SE-cyclic*:
dfs-constructable SE-dfs s ⟹ *has-cycle s* ⟹ *SE-cyclic s*
⟨*proof*⟩

lemma *SE-cycle-not-completed*:
assumes *reach*: *x* →⋆ *v*
and *cycle*: *v* →+ *v*
and *fin*: *v* ∈ 𝒜
and *constr*: *s* ∈ *dfs-constr-from SE-dfs x*
shows ¬ *dfs-completed SE-dfs s*
⟨*proof*⟩

lemma *cycle-generates-SE-cyclic*:
assumes *reach*: *x* →⋆ *v*
and *cycle*: *v* →+ *v*

and *fin*: $v \in \mathcal{A}$
obtains s **where** $s \in$ *dfs-constr-from SE-dfs x SE-cyclic s*
⟨*proof*⟩

lemma *SE-cycle-get-cycle*:
assumes *SE-cycle x*
obtains v **where** $x \rightarrow\star\ v\ v \in \mathcal{A}\ v \rightarrow+\ v$
⟨*proof*⟩

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*⟩

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*$)$
⟨*proof*⟩

theorem *SE-dfs-correct*:
assumes $x \in V$
and *inres* (*dfs-fun SE-dfs x*) *s*
shows *has-cycle s* $\longleftrightarrow$ *SE-cycle x*
⟨*proof*⟩

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*) }
⟨*proof*⟩

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* ⟨*proof*⟩
interpretation *es*: *set-memb es-α es-invar es-memb* ⟨*proof*⟩
interpretation *es*: *set-isEmpty es-α es-invar es-isEmpty* ⟨*proof*⟩
interpretation *es*: *set-to-list es-α es-invar es-to-list* ⟨*proof*⟩
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*
⟨*proof*⟩

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

definition *dfs-impl-state-invar* **where**
dfs-impl-state-invar P ≡ λ*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* × ′*n*) *set* +
fixes α_σ :: ′*S* ⇒ ′*Sa*
begin
definition *impl-sws-α* :: (′*S*, ′*n*, ′*m*) *impl-sws* ⇒ (′*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*
⟨*proof*⟩

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*)
⟨*proof*⟩
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 $\gamma succs :: 'n \Rightarrow 'n\ list$
assumes *succs-correct*: $set\ (\gamma succs\ y) = succs\ y$
and *succs-distinct*: $distinct\ (\gamma succs\ y)$
begin

fun $dfs\text{-}step\text{-}impl' :: ('S,'n,'m)\ impl\text{-}sws \Rightarrow ('S,'n,'m)\ impl\text{-}sws \Rightarrow ('S,'n,'m)\ impl\text{-}sws$
where
$dfs\text{-}step\text{-}impl'\ s\ (S,\ n,\ d,\ f,\ c,\ (x\#xs),\ ([]\#ys)) = (dfs\text{-}impl\text{-}post\ impl\text{-}dfs\ S\ s\ x,$
$n,\ d,\ map.update\ x\ c\ f,\ c + 1,\ xs,\ ys)$
$|\ dfs\text{-}step\text{-}impl'\ s\ (S,\ n,\ d,\ f,\ c,\ (x\#xs),\ ((z\#zs)\#ys)) = (if\ setmemb\ z\ (dfs\text{-}impl\text{-}restrict$
$impl\text{-}dfs)\ then\ (S,\ n,\ d,\ f,\ c,\ (x\#xs),\ (zs\#ys))$
$else\ if\ map.lookup\ z\ d \neq None\ then$
$(dfs\text{-}impl\text{-}remove\ impl\text{-}dfs\ S\ s\ z,\ n,\ d,\ f,\ c,\ x\ \#\ xs,\ zs\ \#\ ys)$
$else\ (dfs\text{-}impl\text{-}action\ impl\text{-}dfs\ S\ s\ z,$
$n,\ map.update\ z\ c\ d,\ f,\ c + 1,\ z\ \#\ x\ \#\ xs,\ (\gamma succs\ z)\ \#\ zs\ \#\ ys))$
$|\ dfs\text{-}step\text{-}impl'\ s\ (S,\ n,\ d,\ f,\ c,\ [],\ []) = s$

abbreviation $dfs\text{-}step\text{-}impl\ s \equiv dfs\text{-}step\text{-}impl'\ s\ s$

definition $dfs\text{-}cond\text{-}impl :: ('S,\ 'n,\ 'm)\ impl\text{-}sws \Rightarrow bool$ **where**
$dfs\text{-}cond\text{-}impl\ s \equiv istack\ s \neq [] \wedge iwll\ s \neq [] \wedge dfs\text{-}impl\text{-}cond\ impl\text{-}dfs\ (istate\ s)$

definition $dfs\text{-}start\text{-}impl :: 'n \Rightarrow ('S,\ 'n,\ 'm)\ impl\text{-}sws$ **where**
$dfs\text{-}start\text{-}impl\ x = (dfs\text{-}impl\text{-}start\ impl\text{-}dfs\ x,\ x,\ map.sng\ x\ 0,\ map.empty\ (),$
$(1::nat),\ [x],\ [\gamma succs\ x])$

definition $impl\text{-}sws\text{-}invar :: ('S,\ 'n,\ 'm)\ impl\text{-}sws \Rightarrow bool$ **where**
$impl\text{-}sws\text{-}invar\ s \equiv$
$dfs\text{-}constructable\ dfs\ (impl\text{-}sws\text{-}\alpha\ s) \wedge$
$map.invar\ (ifinish\ s) \wedge map.invar\ (idisc\ s) \wedge$
$(\forall\ w \in set\ (iwll\ s).\ distinct\ w \wedge set\ w \subseteq V)$

lemma *impl-sws-invarI*:
$[\![\ dfs\text{-}constructable\ dfs\ (impl\text{-}sws\text{-}\alpha\ s);$
$map.invar\ (ifinish\ s);\ map.invar\ (idisc\ s);$
$\bigwedge w.\ w \in set\ (iwll\ s) \Longrightarrow distinct\ w \wedge set\ w \subseteq V\ ]\!] \Longrightarrow impl\text{-}sws\text{-}invar\ s$
⟨*proof*⟩

lemma *impl-sws-invarE*:
$[\![\ impl\text{-}sws\text{-}invar\ s;$
$[\![\ dfs\text{-}constructable\ dfs\ (impl\text{-}sws\text{-}\alpha\ s);$
$map.invar\ (ifinish\ s);\ map.invar\ (idisc\ s);$
$\bigwedge w.\ w \in set\ (iwll\ s) \Longrightarrow distinct\ w \wedge set\ w \subseteq V\ ]\!] \Longrightarrow P\ ]\!]$
$\Longrightarrow P$
⟨*proof*⟩

lemma *impl-sws-invar-constructable*:
$impl\text{-}sws\text{-}invar\ s \Longrightarrow dfs\text{-}constructable\ dfs\ (impl\text{-}sws\text{-}\alpha\ s)$
⟨*proof*⟩

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

lemma *impl-sws-invar-mapinvars*:
impl-sws-invar s $\Longrightarrow$ *map.invar* (*idisc s*)
impl-sws-invar s $\Longrightarrow$ *map.invar* (*ifinish s*)
⟨*proof*⟩

lemma *impl-sws-invar-dfs-invar*:
dfs-preserves-invar dfs $\Longrightarrow$ *impl-sws-invar s* $\Longrightarrow$ *dfs-invar dfs* (*impl-sws-α s*)
⟨*proof*⟩

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 $\Longrightarrow$ *x* ∈ *V* $\Longrightarrow$ *x* ∉ *setα* (*dfs-impl-restrict impl-dfs*) $\Longrightarrow$ *impl-sws-invar* (*dfs-start-impl x*) $\Longrightarrow$ *dfs-impl-invar impl-dfs* (*dfs-start-impl x*)
⟨*proof*⟩

lemma *impl-preserves-invar-step*:
impl-preserves-invar $\Longrightarrow$ *impl-sws-invar s* $\Longrightarrow$ *dfs-cond-impl s* $\Longrightarrow$ *dfs-impl-invar impl-dfs s* $\Longrightarrow$ *s'* = *dfs-step-impl s* $\Longrightarrow$ *dfs-impl-invar impl-dfs s'*
⟨*proof*⟩

lemma *impl-preserves-invarI*:
⟦ ⋀*v*. ⟦*v* ∈ *V*; *v* ∉ *setα* (*dfs-impl-restrict impl-dfs*); *impl-sws-invar* (*dfs-start-impl v*)⟧ $\Longrightarrow$ *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* ⟧ $\Longrightarrow$ *dfs-impl-invar impl-dfs s'*
⟧ $\Longrightarrow$ *impl-preserves-invar*
⟨*proof*⟩

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*) ⟧ $\Longrightarrow$ *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*⟧ $\Longrightarrow$ *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*⟧ $\Longrightarrow$ *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*⟧ $\Longrightarrow$ *P* (*dfs-impl-action impl-dfs* (*istate s*) *s*

x)
shows *impl-preserves-invar*
⟨*proof*⟩
end

locale *DFS-Impl-correct* = *DFS-Impl dfs impl-dfs mops setα setinvar setmemb V E* α_σ *γsuccs*
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*
and *γsuccs* :: *'n* ⇒ *'n list+*
assumes *action-correct*: ⟦*x* ∈ *set* (*hd* (*iwll s*)); *map.lookup x* (*idisc s*) = *None*; *dfs-impl-invar impl-dfs s*; *impl-sws-invar s*; *dfs-cond-impl s* ⟧ ⟹ α_σ (*dfs-impl-action impl-dfs* (*istate s*) *s x*) = *dfs-action dfs* (α_σ (*istate s*)) (*impl-sws-α s*) *x*
and *post-correct*: ⟦*x* ∈ *set* (*istack s*); *dfs-impl-invar impl-dfs s*; *impl-sws-invar s*; *dfs-cond-impl s* ⟧ ⟹ α_σ (*dfs-impl-post impl-dfs* (*istate s*) *s x*) = *dfs-post dfs* (α_σ (*istate s*)) (*impl-sws-α s*) *x*
and *remove-correct*: ⟦*x* ∈ *set* (*hd* (*iwll s*)); *map.lookup x* (*idisc s*) ≠ *None*; *dfs-impl-invar impl-dfs s*; *impl-sws-invar s*; *dfs-cond-impl s* ⟧ ⟹ α_σ (*dfs-impl-remove impl-dfs* (*istate s*) *s x*) = *dfs-remove dfs* (α_σ (*istate s*)) (*impl-sws-α s*) *x*
and *start-correct*: ⟦*x* ∈ *V*; *x* ∉ *setα* (*dfs-impl-restrict impl-dfs*)⟧ ⟹ α_σ (*dfs-impl-start impl-dfs x*) = *dfs-start dfs x*
and *cond-correct*: *dfs-impl-cond impl-dfs S* = *dfs-cond dfs* (α_σ *S*)
and *restr-correct*: *setα* (*dfs-impl-restrict impl-dfs*) = *dfs-restrict dfs*
and *restr-invar*: *setinvar* (*dfs-impl-restrict impl-dfs*)
and *impl-preserves-invar*: *impl-preserves-invar*
begin

lemma *dfs-cond-impl-correct*:
dfs-cond-impl s = *dfs-cond-compl dfs* (*impl-sws-α s*)
⟨*proof*⟩

lemma *dfs-cond-impl-conv*:
dfs-cond-impl (*S*, *n*, *d* ,*f* ,*c*, *st*, *wll*) ≡ *st* ≠ [] ∧ *wll* ≠ [] ∧ *dfs-impl-cond impl-dfs S*
⟨*proof*⟩

lemma *dfs-start-impl-correct*:
x ∈ *V* ⟹ *x* ∉ *setα* (*dfs-impl-restrict impl-dfs*) ⟹ *impl-sws-α* (*dfs-start-impl x*) = *dfs-constr-start dfs x*
⟨*proof*⟩

lemma *dfs-start-impl-constructable*:
assumes *x* ∈ *V*
and *x* ∉ *setα* (*dfs-impl-restrict impl-dfs*)

shows *dfs-constructable dfs* (*impl-sws-α* (*dfs-start-impl x*))
⟨*proof*⟩

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-sws-α* $s' \in dfs\text{-}step\ dfs$ (*impl-sws-α s*)
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

definition *dfs-fun-impl* :: $'n \Rightarrow (('S,'n,'m)\ impl\text{-}sws)\ nres$ **where**
$dfs\text{-}fun\text{-}impl\ x \equiv WHILE_T\ dfs\text{-}cond\text{-}impl\ (\lambda s.\ RETURN\ (dfs\text{-}step\text{-}impl\ s))\ (dfs\text{-}start\text{-}impl\ x)$

definition *dfs-impl-build-rel* **where**
$dfs\text{-}impl\text{-}build\text{-}rel \equiv br\ impl\text{-}sws\text{-}\alpha\ (\lambda s.\ impl\text{-}sws\text{-}invar\ s \wedge dfs\text{-}impl\text{-}invar\ impl\text{-}dfs\ s)$

lemma *dfs-fun-impl-refine*:
assumes $x \in V\ x \notin dfs\text{-}restrict\ dfs$
shows $dfs\text{-}fun\text{-}impl\ x \leq \Downarrow\ dfs\text{-}impl\text{-}build\text{-}rel\ (dfs\text{-}fun\ dfs\ x)$
⟨*proof*⟩

theorem *dfs-fun-impl-correct*:
assumes *dfs-preserves-invar dfs* $x \in V\ x \notin dfs\text{-}restrict\ dfs$
shows $dfs\text{-}fun\text{-}impl\ x \leq \Downarrow\ dfs\text{-}impl\text{-}build\text{-}rel\ (SPEC\ (\lambda s.\ s \in dfs\text{-}constr\text{-}from\ dfs\ x \wedge dfs\text{-}finished\ dfs\ s))$
⟨*proof*⟩

lemma *dfs-fun-impl-nofail*[*refine-pw-simps*]:
assumes *dfs-preserves-invar dfs* $x \in V\ x \notin dfs\text{-}restrict\ dfs$

shows *nofail* (*dfs-fun-impl x*)
⟨*proof*⟩

schematic-lemma *dfs-code-aux*:
RETURN ?dfs-code ≤ *dfs-fun-impl x*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

corollary *dfs-code-preserves-invar*:
⟦ *dfs-preserves-invar dfs*; *x* ∈ *V*; *x* ∉ *dfs-restrict dfs* ⟧ ⟹ *impl-sws-invar* (*dfs-code x*)
⟨*proof*⟩

corollary *dfs-code-preserves-dfs-invar*:
⟦ *dfs-preserves-invar dfs*; *x* ∈ *V*; *x* ∉ *dfs-restrict dfs* ⟧ ⟹ *dfs-impl-invar impl-dfs* (*dfs-code x*)
⟨*proof*⟩

corollary *dfs-code-finished*:
⟦ *dfs-preserves-invar dfs*; *x* ∈ *V*; *x* ∉ *dfs-restrict dfs* ⟧ ⟹ *dfs-finished dfs* (*impl-sws-α* (*dfs-code x*))
⟨*proof*⟩
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*⟩
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*
⟨*proof*⟩
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
⟨*proof*⟩

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
⟨*proof*⟩

lemma *dfs-cond-impl-conv-simple*:
  dfs-cond-impl (*S*, *n*, *d*, *f*, *c*, *st*, *wll*) ≡ *st* ≠ [] ∧ *wll* ≠ []
⟨*proof*⟩

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* ⊆ *DFS-Impl-correct simple-dfs simple-impl-dfs - es-α es-invar es-memb - - id*
⟨*proof*⟩

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* × ′*n*) *set* +
fixes *sup-st* :: ′*n* ⇒ *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 ≡ *set.invar* (*snd* (*istate s*)) ∧ *set.α* (*snd* (*istate s*)) = *set.α restr* ∪ *dom* (*map.α* (*ifinish s*))

definition *sub-impl-dfs* :: (*bool* × ′*s*, ′*n*, ′*m*, ′*s*) *dfs-algorithm-impl* **where**
sub-impl-dfs = (‖ *dfs-impl-cond* = *Not* ∘ *fst*,
dfs-impl-action = λ(*f*,*S*) *s x*. (*check-cycle-impl f s x*, *S*),
dfs-impl-post = λ(*f*,*S*) - *x*. (*f*, *set.ins-dj x S*),
dfs-impl-remove = λ(*f*,*S*) *s x*. (*x* = *istart s* ∧ *sup-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 ⟷ ¬ (*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* ∧ *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*
⟨*proof*⟩

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 $ss :: 'n \Rightarrow bool$ **and** $rs :: 's$
begin

lemma *check-cycle-impl-correct*:
$check\text{-}cycle\text{-}impl\ S\ s\ e \longleftrightarrow check\text{-}cycle\ \{x.\ ss\ x\}\ S\ s'\ e$
⟨*proof*⟩

lemma *sub-impl-preserves-invar*:
impl-preserves-invar
⟨*proof*⟩

end

sublocale *SubDFS-Impl* ⊆ *DFS-Impl-correct sub-dfs* $\{x.\ ss\ x\}$ $(set.\alpha\ rs)$ *sub-impl-dfs mops set.α set.invar set.memb V E fst*
⟨*proof*⟩

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 $x \in V$ **and** $x \notin set.\alpha\ rs$
shows $sub\text{-}dfs\text{-}fun\text{-}impl\ x \leq \Downarrow\ sub\text{-}build\text{-}rel\ (SPEC\ (\lambda s.\ state\ s \longleftrightarrow sub\text{-}cycle\ \{x.\ ss\ x\}\ (set.\alpha\ rs)\ x))$
⟨*proof*⟩

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

theorem *sub-dfs-code-correct*:
$[\![\ x \in V;\ x \notin set.\alpha\ rs;\ s = sub\text{-}dfs\text{-}code\ x\ ]\!] \Longrightarrow fst\ (istate\ s) \longleftrightarrow sub\text{-}cycle\ \{x.\ ss\ x\}\ (set.\alpha\ rs)\ x$
⟨*proof*⟩

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*⟩
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* ⇒ *'n set* **and** *setAinvar* :: *'sA* ⇒ *bool* **and** *setAmemb* :: *'n* ⇒ *'sA* ⇒ *bool*
and *mops* :: (*'n*, *nat*, *'m*, *'X*) *map-ops-scheme*
and *σα* :: *bool* × *'s* ⇒ *bool* × *'n set*
and *V*::*'n set* **and** *E* :: (*'n* × *'n*) *set* **and** $\mathcal{A}$ :: *'sA*+
fixes *γsuccs* :: *'n* ⇒ *'n list*
defines *σα-def*: *σα* ≡ *λ*(*b*,*s*). (*b*, *set.α s*)
assumes *ssuccs-correct*: *set* (*γsuccs y*) = *succs y*
and *ssuccs-distinct*: *distinct* (*γsuccs y*)
and *A-invar*: *setAinvar* $\mathcal{A}$
begin

lemma *σα-simps*[*simp*]:
fst (*σα S*) ⟷ *fst S*
snd (*σα S*) = *set.α* (*snd S*)
σα (*b*, *F*) = (*b*, *set.α F*)
⟨*proof*⟩

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* $\mathcal{A}$ *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*))
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*⟩

lemma *run-sub-dfs-impl-correct'*:
assumes *x* ∈ *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 *σα* (*run-sub-dfs-impl* (*istate s*) *s x*) = *run-sub-dfs M* (*σα* (*istate s*))

(*impl-sws-α s*) *x*
⟨*proof*⟩

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*⟩

end

locale *NestedDFS-Impl* = *NestedDFS-Impl-def* - - - *σα V E* - *γsuccs sops mops* + *DFS-Impl* (*nested-dfs M*) *nested-impl-dfs mops es-α es-invar es-memb V E σα γsuccs*
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 *γsuccs* :: *′n* ⇒ *′n list*
and *σα* :: *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 *σα* (*run-sub-dfs-impl* (*istate s*) *s x*) = *run-sub-dfs M* (*σα* (*istate s*)) (*impl-sws-α s*) *x*
⟨*proof*⟩

lemma *nested-impl-preserves*:
impl-preserves-invar
⟨*proof*⟩
end

sublocale *NestedDFS-Impl* ⊆ *DFS-Impl-correct nested-dfs M nested-impl-dfs mops es-α es-invar es-memb V E σα*
⟨*proof*⟩

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*)
⟨*proof*⟩
end

sublocale *BasicDFS-Impl-def* ⊆ *NestedDFS-Impl-def* - - - - - - - - - - *basicM basicM-impl*
⟨*proof*⟩

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

sublocale *BasicDFS-Impl* ⊆ *NestedDFS-Impl* - - - - *basicM basicM-impl*
⟨*proof*⟩

context *BasicDFS-Impl*
begin

abbreviation *basic-impl-dfs* ≡ *nested-impl-dfs*
abbreviation *basic-dfs-fun-impl* ≡ *dfs-fun-impl*
abbreviation *basic-dfs-code* ≡ *dfs-code*
abbreviation *basic-build-rel* ≡ *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*⟩

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 = *basic-dfs-code* x ⟧ ⟹ *fst* (*istate* s) $\longleftrightarrow$ *cycle basicM* x
⟨*proof*⟩

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)
⟨*proof*⟩
end

sublocale *HPYDFS-Impl-def* ⊆ *NestedDFS-Impl-def* - - - - - - - - - - *hpyM hpyM-impl*
⟨*proof*⟩

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

sublocale *HPYDFS-Impl* ⊆ *NestedDFS-Impl* - - - - *hpyM hpyM-impl*
⟨*proof*⟩

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* ∈ *V*
shows *hpy-dfs-fun-impl x* ≤ ⇓ *hpy-build-rel* (*SPEC* (λ*s*. *has-cycle s* ⟷ *cycle hpyM x*))
⟨*proof*⟩

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* ∈ *V*; *s* = *hpy-dfs-code x* ⟧ ⟹ *fst* (*istate s*) ⟷ *cycle hpyM x*
⟨*proof*⟩

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* 𝒜 ∨ *setAmemb* (*hd* (*istack s*)) 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
⟨*proof*⟩
end

locale *SEDFS-Impl = SEDFS-Impl-def sops - - - mops σα V E - γsuccs + DFS-Impl SE-dfs SE-impl-dfs mops es-α es-invar es-memb V E σα γsuccs*
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 *γsuccs* :: *'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*⟩

lemma *SE-impl-preserves*:
impl-preserves-invar
⟨*proof*⟩

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*⟩
end

sublocale *SEDFS-Impl ⊆ DFS-Impl-correct SE-dfs SE-impl-dfs mops es-α es-invar es-memb V E σα*
⟨*proof*⟩

context *SEDFS-Impl*
begin

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

theorem *SE-dfs-fun-impl-correct*:
assumes *x ∈ V*
shows *SE-dfs-fun-impl x ≤ ⇓ SE-build-rel (SPEC (λs. has-cycle s ⟷ SE-cycle*

x))
⟨*proof*⟩

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 ⟧ ⟹ *fst* (*istate* s) ⟷ *SE-cycle* x
⟨*proof*⟩

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