A Framework for Verifying Depth-First Search Algorithms

Peter Lammich and René Neumann

September 13, 2023

Abstract

This entry presents a framework for the modular verification of DFS-based algorithms, which is described in our [CPP-2015] paper. It provides a generic DFS algorithm framework, that can be parameterized with user-defined actions on certain events (e.g. discovery of new node).

It comes with an extensible library of invariants, which can be used to derive invariants of a specific parameterization.

Using refinement techniques, efficient implementations of the algorithms can easily be derived. Here, the framework comes with templates for a recursive and a tail-recursive implementation, and also with several templates for implementing the data structures required by the DFS algorithm.

Finally, this entry contains a set of re-usable DFS-based algorithms, which illustrate the application of the framework.

[CPP-2015] Peter Lammich, René Neumann: A Framework for Verifying Depth-First Search Algorithms. CPP 2015: 137-146

Contents

Chapter 1

The DFS Framework

This chapter contains the basic DFS Framework

1.1 General DFS with Hooks

theory *Param-DFS*
imports
  CAVA-Base.CAVA-Base
  CAVA-Automata.Digraph
  Misc/DFS-Framework-Refine-Aux
begin

We define a general DFS algorithm, which is parameterized over hook functions at certain events during the DFS.

1.1.1 State and Parameterization

The state of the general DFS. Users may inherit from this state using the record package's inheritance support.

record *'v state* =
  counter :: *nat* — Node counter (timer)
  discovered :: *'v* ⇀ *nat* — Discovered times of nodes
  finished :: *'v* ⇀ *nat* — Finished times of nodes
  pending :: (*'v* × *'v*) *set* — Edges to be processed next
  stack :: *'v list* — Current DFS stack
  tree-edges :: *'v rel* — Tree edges
  back-edges :: *'v rel* — Back edges
  cross-edges :: *'v rel* — Cross edges

abbreviation *NOOP s* ≡ *RETURN* (*state.more s*)

Record holding the parameterization.

record (*'v*,*'s*,*'es*) *gen-parameterization* =

on-init :: *'es nres*
on-new-root :: *'v* ⇒ *'s* ⇒ *'es nres*
on-discover :: *'v* ⇒ *'v* ⇒ *'s* ⇒ *'es nres*
on-finish :: *'v* ⇒ *'s* ⇒ *'es nres*
on-back-edge :: *'v* ⇒ *'v* ⇒ *'s* ⇒ *'es nres*
on-cross-edge :: *'v* ⇒ *'v* ⇒ *'s* ⇒ *'es nres*
is-break :: *'s* ⇒ *bool*

Default type restriction for parameterizations. The event handler functions go from a complete state to the user-defined part of the state (i.e. the fields added by inheritance).

type-synonym (*'v*,*'es*) *parameterization*
= (*'v*,(*'v*,*'es*) *state-scheme*,*'es*) *gen-parameterization*

Default parameterization, the functions do nothing. This can be used as the basis for specialized parameterizations, which may be derived by updating some fields.

definition ⋀*more init. dflt-parametrization more init* ≡ ⦇
on-init = *init*,
on-new-root = λ-. *RETURN o more*,
on-discover = λ- -. *RETURN o more*,
on-finish = λ-. *RETURN o more*,
on-back-edge = λ- -. *RETURN o more*,
on-cross-edge = λ- -. *RETURN o more*,
is-break = λ-. *False* ⦈
lemmas *dflt-parametrization-simp*[*simp*] =
gen-parameterization.simps[*mk-record-simp*, *OF dflt-parametrization-def*]

This locale builds a DFS algorithm from a graph and a parameterization.

locale *param-DFS-defs* =
graph-defs G
for *G* :: (*'v*, *'more*) *graph-rec-scheme*
\+
fixes *param* :: (*'v*,*'es*) *parameterization*
begin

1.1.2 DFS operations

Node predicates

First, we define some predicates to check whether nodes are in certain sets

definition *is-discovered* :: *'v* ⇒ (*'v*,*'es*) *state-scheme* ⇒ *bool*
where *is-discovered u s* ≡ *u* ∈ *dom* (*discovered s*)

definition *is-finished* :: *'v* ⇒ (*'v*,*'es*) *state-scheme* ⇒ *bool*
where *is-finished u s* ≡ *u* ∈ *dom* (*finished s*)

definition *is-empty-stack* :: (*'v*,*'es*) *state-scheme* ⇒ *bool*
where *is-empty-stack s* ≡ *stack s* = []

Effects on Basic State

We define the effect of the operations on the basic part of the state

definition *discover*
  $:: 'v \Rightarrow 'v \Rightarrow ('v,'es)\ state\text{-}scheme \Rightarrow ('v,'es)\ state\text{-}scheme$
where
$discover\ u\ v\ s \equiv let$
  $d = (discovered\ s)(v \mapsto counter\ s);\ c = counter\ s + 1;$
  $st = v\#stack\ s;$
  $p = pending\ s \cup \{v\} \times E``\{v\};$
  $t = insert\ (u,v)\ (tree\text{-}edges\ s)$
$in\ s(\!|\ discovered := d,\ counter := c,\ stack := st,\ pending := p,\ tree\text{-}edges := t|\!)$

lemma *discover-simps*[*simp*]:
  $counter\ (discover\ u\ v\ s) = Suc\ (counter\ s)$
  $discovered\ (discover\ u\ v\ s) = (discovered\ s)(v \mapsto counter\ s)$
  $finished\ (discover\ u\ v\ s) = finished\ s$
  $stack\ (discover\ u\ v\ s) = v\#stack\ s$
  $pending\ (discover\ u\ v\ s) = pending\ s \cup \{v\} \times E``\{v\}$
  $tree\text{-}edges\ (discover\ u\ v\ s) = insert\ (u,v)\ (tree\text{-}edges\ s)$
  $cross\text{-}edges\ (discover\ u\ v\ s) = cross\text{-}edges\ s$
  $back\text{-}edges\ (discover\ u\ v\ s) = back\text{-}edges\ s$
  $state.more\ (discover\ u\ v\ s) = state.more\ s$
  by (*simp-all add*: *discover-def*)

definition *finish*
  $:: 'v \Rightarrow ('v,'es)\ state\text{-}scheme \Rightarrow ('v,'es)\ state\text{-}scheme$
where
$finish\ u\ s \equiv let$
  $f = (finished\ s)(u \mapsto counter\ s);\ c = counter\ s + 1;$
  $st = tl\ (stack\ s)$
$in\ s(\!|\ finished := f,\ counter := c,\ stack := st|\!)$

lemma *finish-simps*[*simp*]:
  $counter\ (finish\ u\ s) = Suc\ (counter\ s)$
  $discovered\ (finish\ u\ s) = discovered\ s$
  $finished\ (finish\ u\ s) = (finished\ s)(u \mapsto counter\ s)$
  $stack\ (finish\ u\ s) = tl\ (stack\ s)$
  $pending\ (finish\ u\ s) = pending\ s$
  $tree\text{-}edges\ (finish\ u\ s) = tree\text{-}edges\ s$
  $cross\text{-}edges\ (finish\ u\ s) = cross\text{-}edges\ s$
  $back\text{-}edges\ (finish\ u\ s) = back\text{-}edges\ s$
  $state.more\ (finish\ u\ s) = state.more\ s$
  by (*simp-all add*: *finish-def*)

definition *back-edge*
  $:: 'v \Rightarrow 'v \Rightarrow ('v,'es)\ state\text{-}scheme \Rightarrow ('v,'es)\ state\text{-}scheme$
where
$back\text{-}edge\ u\ v\ s \equiv let$

  $b = insert\ (u,v)\ (back\text{-}edges\ s)$
 $in\ s(\!|\ back\text{-}edges := b\ |\!)$

lemma *back-edge-simps*[*simp*]:
  $counter\ (back\text{-}edge\ u\ v\ s) = counter\ s$
  $discovered\ (back\text{-}edge\ u\ v\ s) = discovered\ s$
  $finished\ (back\text{-}edge\ u\ v\ s) = finished\ s$
  $stack\ (back\text{-}edge\ u\ v\ s) = stack\ s$
  $pending\ (back\text{-}edge\ u\ v\ s) = pending\ s$
  $tree\text{-}edges\ (back\text{-}edge\ u\ v\ s) = tree\text{-}edges\ s$
  $cross\text{-}edges\ (back\text{-}edge\ u\ v\ s) = cross\text{-}edges\ s$
  $back\text{-}edges\ (back\text{-}edge\ u\ v\ s) = insert\ (u,v)\ (back\text{-}edges\ s)$
  $state.more\ (back\text{-}edge\ u\ v\ s) = state.more\ s$
  by (*simp-all add*: *back-edge-def*)

definition *cross-edge*
  $:: 'v \Rightarrow 'v \Rightarrow ('v,'es)\ state\text{-}scheme \Rightarrow ('v,'es)\ state\text{-}scheme$
where
$cross\text{-}edge\ u\ v\ s \equiv let$
  $c = insert\ (u,v)\ (cross\text{-}edges\ s)$
 $in\ s(\!|\ cross\text{-}edges := c\ |\!)$

lemma *cross-edge-simps*[*simp*]:
  $counter\ (cross\text{-}edge\ u\ v\ s) = counter\ s$
  $discovered\ (cross\text{-}edge\ u\ v\ s) = discovered\ s$
  $finished\ (cross\text{-}edge\ u\ v\ s) = finished\ s$
  $stack\ (cross\text{-}edge\ u\ v\ s) = stack\ s$
  $pending\ (cross\text{-}edge\ u\ v\ s) = pending\ s$
  $tree\text{-}edges\ (cross\text{-}edge\ u\ v\ s) = tree\text{-}edges\ s$
  $cross\text{-}edges\ (cross\text{-}edge\ u\ v\ s) = insert\ (u,v)\ (cross\text{-}edges\ s)$
  $back\text{-}edges\ (cross\text{-}edge\ u\ v\ s) = back\text{-}edges\ s$
  $state.more\ (cross\text{-}edge\ u\ v\ s) = state.more\ s$
  by (*simp-all add*: *cross-edge-def*)

definition *new-root*
  $:: 'v \Rightarrow ('v,'es)\ state\text{-}scheme \Rightarrow ('v,'es)\ state\text{-}scheme$
where
  $new\text{-}root\ v0\ s \equiv let$
    $c = Suc\ (counter\ s);$
    $d = (discovered\ s)(v0 \mapsto counter\ s);$
    $p = \{v0\} \times E``\{v0\};$
    $st = [v0]$
   $in\ s(\!|counter := c,\ discovered := d,\ pending := p,\ stack := st|\!)$

lemma *new-root-simps*[*simp*]:
  $counter\ (new\text{-}root\ v0\ s) = Suc\ (counter\ s)$
  $discovered\ (new\text{-}root\ v0\ s) = (discovered\ s)(v0 \mapsto counter\ s)$
  $finished\ (new\text{-}root\ v0\ s) = finished\ s$

stack (*new-root v0 s*) = [*v0*]
pending (*new-root v0 s*) = ({*v0*}×*E*``{*v0*})
tree-edges (*new-root v0 s*) = *tree-edges s*
cross-edges (*new-root v0 s*) = *cross-edges s*
back-edges (*new-root v0 s*) = *back-edges s*
state.more (*new-root v0 s*) = *state.more s*
by (*simp-all add*: *new-root-def*)

definition *empty-state e*
≡ ⦇ *counter* = *0*,
discovered = *Map.empty*,
finished = *Map.empty*,
pending = {},
stack = [],
tree-edges = {},
back-edges = {},
cross-edges = {},
. . . = *e* ⦈

lemma *empty-state-simps*[*simp*]:
counter (*empty-state e*) = *0*
discovered (*empty-state e*) = *Map.empty*
finished (*empty-state e*) = *Map.empty*
pending (*empty-state e*) = {}
stack (*empty-state e*) = []
tree-edges (*empty-state e*) = {}
back-edges (*empty-state e*) = {}
cross-edges (*empty-state e*) = {}
state.more (*empty-state e*) = *e*
by (*simp-all add*: *empty-state-def*)

Effects on Whole State

The effects of the operations on the whole state are defined by combining the effects of the basic state with the parameterization.

definition *do-cross-edge*
:: *'v* ⇒ *'v* ⇒ (*'v*,*'es*) *state-scheme* ⇒ (*'v*,*'es*) *state-scheme nres*
where
do-cross-edge u v s ≡ *do* {
let s = *cross-edge u v s*;
e ← *on-cross-edge param u v s*;
RETURN (*s*⦇*state.more* := *e*⦈)
}

definition *do-back-edge*
:: *'v* ⇒ *'v* ⇒ (*'v*,*'es*) *state-scheme* ⇒ (*'v*,*'es*) *state-scheme nres*
where
do-back-edge u v s ≡ *do* {
let s = *back-edge u v s*;

$e \leftarrow$ *on-back-edge param u v s*;
$RETURN$ (s(|*state.more* := e|))
}

definition *do-known-edge*
:: $'v \Rightarrow 'v \Rightarrow ('v,'es)$ *state-scheme* $\Rightarrow ('v,'es)$ *state-scheme nres*
where
do-known-edge u v s $\equiv$
if is-finished v s then
do-cross-edge u v s
else
do-back-edge u v s

definition *do-discover*
:: $'v \Rightarrow 'v \Rightarrow ('v,'es)$ *state-scheme* $\Rightarrow ('v,'es)$ *state-scheme nres*
where
do-discover u v s $\equiv$ *do* {
let s = *discover u v s*;
$e \leftarrow$ *on-discover param u v s*;
$RETURN$ (s(|*state.more* := e|))
}

definition *do-finish*
:: $'v \Rightarrow ('v,'es)$ *state-scheme* $\Rightarrow ('v,'es)$ *state-scheme nres*
where
do-finish u s $\equiv$ *do* {
let s = *finish u s*;
$e \leftarrow$ *on-finish param u s*;
$RETURN$ (s(|*state.more* := e|))
}

definition *get-new-root* **where**
get-new-root s $\equiv$ *SPEC* ($\lambda v.\ v \in V0 \wedge \neg$*is-discovered v s*)

definition *do-new-root* **where**
do-new-root v0 s $\equiv$ *do* {
let s = *new-root v0 s*;
$e \leftarrow$ *on-new-root param v0 s*;
$RETURN$ (s(|*state.more* := e|))
}

lemmas *op-defs* = *discover-def finish-def back-edge-def cross-edge-def new-root-def*
lemmas *do-defs* = *do-discover-def do-finish-def do-known-edge-def*
do-cross-edge-def do-back-edge-def do-new-root-def
lemmas *pred-defs* = *is-discovered-def is-finished-def is-empty-stack-def*

definition *init* $\equiv$ *do* {
$e \leftarrow$ *on-init param*;
$RETURN$ (*empty-state e*)

}

1.1.3 DFS Algorithm

We phrase the DFS algorithm iteratively: While there are undiscovered root nodes or the stack is not empty, inspect the topmost node on the stack: Follow any pending edge, or finish the node if there are no pending edges left.

definition *cond* :: $('v,'es)$ *state-scheme* $\Rightarrow$ *bool* **where**
cond s $\longleftrightarrow$ $(V0 \subseteq \{v.\ \textit{is-discovered}\ v\ s\} \longrightarrow \neg\textit{is-empty-stack}\ s)$
$\wedge\ \neg\textit{is-break param}\ s$

lemma *cond-alt*:
$\textit{cond} = (\lambda s.\ (V0 \subseteq \textit{dom}\ (\textit{discovered}\ s) \longrightarrow \textit{stack}\ s \neq []) \wedge \neg\textit{is-break param}\ s)$
apply (*rule ext*)
unfolding *cond-def is-discovered-def is-empty-stack-def*
by *auto*

definition *get-pending* ::
$('v,\ 'es)$ *state-scheme* $\Rightarrow$ $('v \times 'v\ \textit{option} \times ('v,\ 'es)\ \textit{state-scheme})$ *nres*
— Get topmost stack node and a pending edge if any. The pending edge is removed.
where *get-pending s* $\equiv$ *do* {
let u = *hd* (*stack s*);
let Vs = *pending s* `` {*u*};

if Vs = {} *then*
RETURN (*u*,*None*,*s*)
else do {
v $\leftarrow$ *RES Vs*;
let s = *s*(| *pending* := *pending s* − {(*u*,*v*)}|);
RETURN (*u*,*Some v*,*s*)
}
}

definition *step* :: $('v,'es)$ *state-scheme* $\Rightarrow$ $('v,'es)$ *state-scheme nres*
where
step s $\equiv$
if is-empty-stack s then do {
v0 $\leftarrow$ *get-new-root s*;
do-new-root v0 s
} *else do* {
(*u*, *Vs*,*s*) $\leftarrow$ *get-pending s*;
case Vs of
None $\Rightarrow$ *do-finish u s*
| *Some v* $\Rightarrow$ *do* {
if is-discovered v s then

do-known-edge u v s
else
do-discover u v s
}
}

definition *it-dfs* ≡ *init* ≫= *WHILE cond step*
definition *it-dfsT* ≡ *init* ≫= *WHILET cond step*

end

1.1.4 Invariants

We now build the infrastructure for establishing invariants of DFS algorithms. The infrastructure is modular and extensible, i.e., we can define re-usable libraries of invariants.

For technical reasons, invariants are established in a two-step process:

1. First, we prove the invariant wrt. the parameterization in the *param-DFS* locale.

2. Next, we transfer the invariant to the *DFS-invar*-locale.

locale *param-DFS* =
fb-graph G + *param-DFS-defs G param*
for *G* :: ('*v*, '*more*) *graph-rec-scheme*
and *param* :: ('*v*,'*es*) *parameterization*
begin

definition *is-invar* :: (('*v*, '*es*) *state-scheme* ⇒ *bool*) ⇒ *bool*
— Predicate that states that *I* is an invariant.
where *is-invar I* ≡ *is-rwof-invar init cond step I*

end

Invariants are transferred to this locale, which is parameterized with a state.

locale *DFS-invar* =
param-DFS G param
for *G* :: ('*v*, '*more*) *graph-rec-scheme*
and *param* :: ('*v*,'*es*) *parameterization*
+
fixes *s* :: ('*v*,'*es*) *state-scheme*
assumes *rwof*: *rwof init cond step s*
begin

lemma *make-invar-thm*: *is-invar I* ⟹ *I s*

— Lemma to transfer an invariant into this locale
using *rwof-cons*[*OF* - *rwof*, *folded is-invar-def*] **.**

end

Establishing Invariants

context *param-DFS*
begin

Include this into refine-rules to discard any information about parameterization

lemmas *indep-invar-rules* =
leof-True-rule[**where** *m*=*on-init param*]
leof-True-rule[**where** *m*=*on-new-root param v0 s'* **for** *v0 s'*]
leof-True-rule[**where** *m*=*on-discover param u v s'* **for** *u v s'*]
leof-True-rule[**where** *m*=*on-finish param v s'* **for** *v s'*]
leof-True-rule[**where** *m*=*on-cross-edge param u v s'* **for** *u v s'*]
leof-True-rule[**where** *m*=*on-back-edge param u v s'* **for** *u v s'*]

lemma *rwof-eq-DFS-invar*[*simp*]:
rwof init cond step = *DFS-invar G param*
— The DFS-invar locale is equivalent to the strongest invariant of the loop.
apply (*auto intro*: *DFS-invar.rwof intro*!: *ext*)
by *unfold-locales*

lemma *DFS-invar-step*: ⟦*nofail it-dfs*; *DFS-invar G param s*; *cond s*⟧
$\Longrightarrow$ *step s* ≤ *SPEC* (*DFS-invar G param*)
— A step preserves the (best) invariant.
unfolding *it-dfs-def rwof-eq-DFS-invar*[*symmetric*]
by (*rule rwof-step*)

lemma *DFS-invar-step'*: ⟦*nofail* (*step s*); *DFS-invar G param s*; *cond s*⟧
$\Longrightarrow$ *step s* ≤ *SPEC* (*DFS-invar G param*)
unfolding *it-dfs-def rwof-eq-DFS-invar*[*symmetric*]
by (*rule rwof-step'*)

We define symbolic names for the preconditions of certain operations

definition *pre-is-break s* ≡ *DFS-invar G param s*

definition *pre-on-new-root v0 s'* ≡ ∃ *s*.
DFS-invar G param s ∧ *cond s* ∧
stack s = [] ∧ *v0* ∈ *V0* ∧ *v0* ∉ *dom* (*discovered s*) ∧
s' = *new-root v0 s*

definition *pre-on-finish u s'* ≡ ∃ *s*.
DFS-invar G param s ∧ *cond s* ∧
stack s ≠ [] ∧ *u* = *hd* (*stack s*) ∧ *pending s* `` {*u*} = {} ∧ *s'* = *finish u s*

definition *pre-edge-selected u v s* $\equiv$
DFS-invar G param s $\wedge$ *cond s* $\wedge$
stack s $\neq$ [] $\wedge$ u = *hd* (*stack s*) $\wedge$ $(u, v) \in$ *pending s*

definition *pre-on-cross-edge u v s'* $\equiv \exists s.$ *pre-edge-selected u v s* $\wedge$
$v \in$ *dom* (*discovered s*) $\wedge$ $v \in$ *dom* (*finished s*)
$\wedge$ s' = *cross-edge u v* (s(|*pending* := *pending s* $-$ $\{(u,v)\}$|))

definition *pre-on-back-edge u v s'* $\equiv \exists s.$ *pre-edge-selected u v s* $\wedge$
$v \in$ *dom* (*discovered s*) $\wedge$ $v \notin$ *dom* (*finished s*)
$\wedge$ s' = *back-edge u v* (s(|*pending* := *pending s* $-$ $\{(u,v)\}$|))

definition *pre-on-discover u v s'* $\equiv \exists s.$ *pre-edge-selected u v s* $\wedge$
$v \notin$ *dom* (*discovered s*)
$\wedge$ s' = *discover u v* (s(|*pending* := *pending s* $-$ $\{(u,v)\}$|))

lemmas *pre-on-defs* = *pre-on-new-root-def pre-on-finish-def*
pre-edge-selected-def pre-on-cross-edge-def pre-on-back-edge-def
pre-on-discover-def pre-is-break-def

Next, we define a set of rules to establish an invariant.

lemma *establish-invarI*[*case-names init new-root finish cross-edge back-edge discover*]:
— Establish a DFS invariant (explicit preconditions).
assumes *init*: *on-init param* $\leq_n$ *SPEC* ($\lambda x.$ *I* (*empty-state x*))
assumes *new-root*: $\bigwedge s\ s'\ v0.$
⟦*DFS-invar G param s*; *I s*; *cond s*; $\neg$ *is-break param s*;
stack s = []; *v0* $\in$ *V0*; *v0* $\notin$ *dom* (*discovered s*);
s' = *new-root v0 s*⟧
$\Longrightarrow$ *on-new-root param v0 s'* $\leq_n$
SPEC ($\lambda x.$ *DFS-invar G param* (s'(|*state.more* := x|))
$\longrightarrow$ *I* (s'(|*state.more* := x|)))
assumes *finish*: $\bigwedge s\ s'\ u.$
⟦*DFS-invar G param s*; *I s*; *cond s*; $\neg$ *is-break param s*;
stack s $\neq$ []; u = *hd* (*stack s*);
pending s `` $\{u\}$ = $\{\}$;
s' = *finish u s*⟧
$\Longrightarrow$ *on-finish param u s'* $\leq_n$
SPEC ($\lambda x.$ *DFS-invar G param* (s'(|*state.more* := x|))
$\longrightarrow$ *I* (s'(|*state.more* := x|)))
assumes *cross-edge*: $\bigwedge s\ s'\ u\ v.$
⟦*DFS-invar G param s*; *I s*; *cond s*; $\neg$ *is-break param s*;
stack s $\neq$ []; $(u, v) \in$ *pending s*; u = *hd* (*stack s*);
$v \in$ *dom* (*discovered s*); $v \in$ *dom* (*finished s*);
s' = *cross-edge u v* (s(|*pending* := *pending s* $-$ $\{(u,v)\}$|))⟧
$\Longrightarrow$ *on-cross-edge param u v s'* $\leq_n$
SPEC ($\lambda x.$ *DFS-invar G param* (s'(|*state.more* := x|))
$\longrightarrow$ *I* (s'(|*state.more* := x|)))

```
  assumes back-edge: ⋀s s' u v.
    ⟦DFS-invar G param s; I s; cond s; ¬ is-break param s;
     stack s ≠ []; (u, v) ∈ pending s; u = hd (stack s);
     v ∈ dom (discovered s); v∉dom (finished s);
     s' = back-edge u v (s⦇pending := pending s − {(u,v)}⦈)⟧
    ⟹ on-back-edge param u v s' ≤n
        SPEC (λx. DFS-invar G param (s'⦇state.more := x⦈)
                ⟶ I (s'⦇state.more := x⦈))
  assumes discover: ⋀s s' u v.
    ⟦DFS-invar G param s; I s; cond s; ¬ is-break param s;
     stack s ≠ [];  (u, v) ∈ pending s; u = hd (stack s);
     v ∉ dom (discovered s);
     s' = discover u v (s⦇pending := pending s − {(u,v)}⦈)⟧
    ⟹ on-discover param u v s' ≤n
        SPEC (λx. DFS-invar G param (s'⦇state.more := x⦈)
                ⟶ I (s'⦇state.more := x⦈))
  shows is-invar I
  unfolding is-invar-def
proof
  show init ≤n SPEC I
    unfolding init-def
    by (refine-rcg refine-vcg) (simp add: init)
next
  fix s
  assume rwof init cond step s and IC: I s cond s
  hence DI: DFS-invar G param s by unfold-locales
  then interpret DFS-invar G param s .

  from ‹cond s› have IB: ¬ is-break param s by (simp add: cond-def)

  have B: step s ≤n SPEC (DFS-invar G param)
    by rule (metis DFS-invar-step' DI ‹cond s›)

  note rule-assms = DI IC IB

  show step s ≤n SPEC I
    apply (rule leof-use-spec-rule[OF B])
    unfolding step-def do-defs pred-defs get-pending-def get-new-root-def
    apply (refine-rcg refine-vcg)
    apply (simp-all)

    apply (blast intro: new-root[OF rule-assms])
    apply (blast intro: finish[OF rule-assms])
    apply (rule cross-edge[OF rule-assms], auto) []
    apply (rule back-edge[OF rule-assms], auto) []
    apply (rule discover[OF rule-assms], auto) []
    done
qed
```

lemma *establish-invarI'*[*case-names init new-root finish cross-edge back-edge discover*]:
— Establish a DFS invariant (symbolic preconditions).
assumes *init*: $on\text{-}init\ param \leq_n SPEC\ (\lambda x.\ I\ (empty\text{-}state\ x))$
assumes *new-root*: $\bigwedge s'\ v0.\ pre\text{-}on\text{-}new\text{-}root\ v0\ s'$
$\Longrightarrow on\text{-}new\text{-}root\ param\ v0\ s' \leq_n$
$SPEC\ (\lambda x.\ DFS\text{-}invar\ G\ param\ (s'(\!|state.more := x|\!))$
$\longrightarrow I\ (s'(\!|state.more := x|\!)))$
assumes *finish*: $\bigwedge s'\ u.\ pre\text{-}on\text{-}finish\ u\ s'$
$\Longrightarrow on\text{-}finish\ param\ u\ s' \leq_n$
$SPEC\ (\lambda x.\ DFS\text{-}invar\ G\ param\ (s'(\!|state.more := x|\!))$
$\longrightarrow I\ (s'(\!|state.more := x|\!)))$
assumes *cross-edge*: $\bigwedge s'\ u\ v.\ pre\text{-}on\text{-}cross\text{-}edge\ u\ v\ s'$
$\Longrightarrow on\text{-}cross\text{-}edge\ param\ u\ v\ s' \leq_n$
$SPEC\ (\lambda x.\ DFS\text{-}invar\ G\ param\ (s'(\!|state.more := x|\!))$
$\longrightarrow I\ (s'(\!|state.more := x|\!)))$
assumes *back-edge*: $\bigwedge s'\ u\ v.\ pre\text{-}on\text{-}back\text{-}edge\ u\ v\ s'$
$\Longrightarrow on\text{-}back\text{-}edge\ param\ u\ v\ s' \leq_n$
$SPEC\ (\lambda x.\ DFS\text{-}invar\ G\ param\ (s'(\!|state.more := x|\!))$
$\longrightarrow I\ (s'(\!|state.more := x|\!)))$
assumes *discover*: $\bigwedge s'\ u\ v.\ pre\text{-}on\text{-}discover\ u\ v\ s'$
$\Longrightarrow on\text{-}discover\ param\ u\ v\ s' \leq_n$
$SPEC\ (\lambda x.\ DFS\text{-}invar\ G\ param\ (s'(\!|state.more := x|\!))$
$\longrightarrow I\ (s'(\!|state.more := x|\!)))$
shows *is-invar I*
apply (*rule establish-invarI*)
using *assms*
unfolding *pre-on-defs*
apply –
apply *blast*
apply (*rprems*,*blast*)+
done

lemma *establish-invarI-ND* [*case-names prereq init new-discover finish cross-edge back-edge*]:
— Establish a DFS invariant (new-root and discover cases are combined).
assumes *prereq*: $\bigwedge u\ v\ s.\ on\text{-}discover\ param\ u\ v\ s = on\text{-}new\text{-}root\ param\ v\ s$
assumes *init*: $on\text{-}init\ param \leq_n SPEC\ (\lambda x.\ I\ (empty\text{-}state\ x))$
assumes *new-discover*: $\bigwedge s\ s'\ v.$
$[\![DFS\text{-}invar\ G\ param\ s;\ I\ s;\ cond\ s;\ \neg\ is\text{-}break\ param\ s;$
$v \notin dom\ (discovered\ s);$
$discovered\ s' = (discovered\ s)(v \mapsto counter\ s);\ finished\ s' = finished\ s;$
$counter\ s' = Suc\ (counter\ s);\ stack\ s' = v\#stack\ s;$
$back\text{-}edges\ s' = back\text{-}edges\ s;\ cross\text{-}edges\ s' = cross\text{-}edges\ s;$
$tree\text{-}edges\ s' \supseteq tree\text{-}edges\ s;$
$state.more\ s' = state.more\ s]\!]$
$\Longrightarrow on\text{-}new\text{-}root\ param\ v\ s' \leq_n$
$SPEC\ (\lambda x.\ DFS\text{-}invar\ G\ param\ (s'(\!|state.more := x|\!))$
$\longrightarrow I\ (s'(\!|state.more := x|\!)))$

assumes *finish*: $\bigwedge s\ s'\ u.$
$[\![$*DFS-invar G param s*; *I s*; *cond s*; $\neg$ *is-break param s*;
stack s $\neq$ []; $u = hd\ (stack\ s)$;
pending s `` $\{u\} = \{\}$;
$s' = finish\ u\ s]\!]$
$\Longrightarrow$ *on-finish param u s'* $\leq_n$
SPEC ($\lambda x.$ *DFS-invar G param* ($s'(\!|state.more := x|\!)$)
$\longrightarrow I\ (s'(\!|state.more := x|\!)))$
assumes *cross-edge*: $\bigwedge s\ s'\ u\ v.$
$[\![$*DFS-invar G param s*; *I s*; *cond s*; $\neg$ *is-break param s*;
stack s $\neq$ []; $(u,\ v) \in pending\ s$; $u = hd\ (stack\ s)$;
$v \in dom\ (discovered\ s)$; $v \in dom\ (finished\ s)$;
$s' = cross\text{-}edge\ u\ v\ (s(\!|pending := pending\ s - \{(u,v)\}|\!))]\!]$
$\Longrightarrow$ *on-cross-edge param u v s'* $\leq_n$
SPEC ($\lambda x.$ *DFS-invar G param* ($s'(\!|state.more := x|\!)$)
$\longrightarrow I\ (s'(\!|state.more := x|\!)))$
assumes *back-edge*: $\bigwedge s\ s'\ u\ v.$
$[\![$*DFS-invar G param s*; *I s*; *cond s*; $\neg$ *is-break param s*;
stack s $\neq$ []; $(u,\ v) \in pending\ s$; $u = hd\ (stack\ s)$;
$v \in dom\ (discovered\ s)$; $v \notin dom\ (finished\ s)$;
$s' = back\text{-}edge\ u\ v\ (s(\!|pending := pending\ s - \{(u,v)\}|\!))]\!]$
$\Longrightarrow$ *on-back-edge param u v s'* $\leq_n$
SPEC ($\lambda x.$ *DFS-invar G param* ($s'(\!|state.more := x|\!)$)
$\longrightarrow I\ (s'(\!|state.more := x|\!)))$
shows *is-invar I*
proof (*induct rule*: *establish-invarI*)
case (*new-root s*) **thus** *?case* **by** (*auto intro*!: *new-discover*)
next
case (*discover s s' u v*) **hence**
on-new-root param v s' $\leq_n$
SPEC ($\lambda x.$ *DFS-invar G param* ($s'(\!|state.more := x|\!)$)
$\longrightarrow I\ (s'(\!|state.more := x|\!)))$
by (*auto intro*!: *new-discover*)
with *prereq* **show** *?case* **by** *simp*
qed *fact+*

lemma *establish-invarI-CB* [*case-names prereq init new-root finish cross-back-edge discover*]:
— Establish a DFS invariant (cross and back edge cases are combined).
assumes *prereq*: $\bigwedge u\ v\ s.$ *on-back-edge param u v s = on-cross-edge param u v s*
assumes *init*: *on-init param* $\leq_n$ *SPEC* ($\lambda x.$ *I* (*empty-state x*))
assumes *new-root*: $\bigwedge s\ s'\ v0.$
$[\![$*DFS-invar G param s*; *I s*; *cond s*; $\neg$ *is-break param s*;
stack s = []; $v0 \in V0$; $v0 \notin dom\ (discovered\ s)$;
$s' = new\text{-}root\ v0\ s]\!]$
$\Longrightarrow$ *on-new-root param v0 s'* $\leq_n$
SPEC ($\lambda x.$ *DFS-invar G param* ($s'(\!|state.more := x|\!)$)
$\longrightarrow I\ (s'(\!|state.more := x|\!)))$

assumes *finish*: $\bigwedge s\ s'\ u.$
$[\![$*DFS-invar G param s*; *I s*; *cond s*; $\neg$ *is-break param s*;
stack s $\neq$ []; $u = hd\ (stack\ s)$;
pending s ‘‘ $\{u\} = \{\}$;
$s' = finish\ u\ s]\!]$
$\Longrightarrow$ *on-finish param u* $s' \leq_n$
SPEC ($\lambda x.$ *DFS-invar G param* ($s'(\!|state.more := x|\!)$)
$\longrightarrow I\ (s'(\!|state.more := x|\!)))$
assumes *cross-back-edge*: $\bigwedge s\ s'\ u\ v.$
$[\![$*DFS-invar G param s*; *I s*; *cond s*; $\neg$ *is-break param s*;
stack s $\neq$ []; $(u, v) \in$ *pending s*; $u = hd\ (stack\ s)$;
$v \in dom\ (discovered\ s)$;
discovered s' = *discovered s*; *finished* s' = *finished s*;
stack s' = *stack s*; *tree-edges* s' = *tree-edges s*; *counter* s' = *counter s*;
pending s' = *pending s* $- \{(u,v)\}$;
cross-edges $s' \cup$ *back-edges* s' = *cross-edges s* $\cup$ *back-edges s* $\cup \{(u,v)\}$;
state.more s' = *state.more s* $]\!]$
$\Longrightarrow$ *on-cross-edge param u v* $s' \leq_n$
SPEC ($\lambda x.$ *DFS-invar G param* ($s'(\!|state.more := x|\!)$)
$\longrightarrow I\ (s'(\!|state.more := x|\!)))$
assumes *discover*: $\bigwedge s\ s'\ u\ v.$
$[\![$*DFS-invar G param s*; *I s*; *cond s*; $\neg$ *is-break param s*;
stack s $\neq$ []; $(u, v) \in$ *pending s*; $u = hd\ (stack\ s)$;
$v \notin dom\ (discovered\ s)$;
$s' = discover\ u\ v\ (s(\!|pending := pending\ s - \{(u,v)\}|\!))]\!]$
$\Longrightarrow$ *on-discover param u v* $s' \leq_n$
SPEC ($\lambda x.$ *DFS-invar G param* ($s'(\!|state.more := x|\!)$)
$\longrightarrow I\ (s'(\!|state.more := x|\!)))$
shows *is-invar I*
proof (*induct rule*: *establish-invarI*)
case *cross-edge* **thus** *?case* **by** (*auto intro*!: *cross-back-edge*)
next
case (*back-edge s s' u v*) **hence**
on-cross-edge param u v $s' \leq_n$
SPEC ($\lambda x.$ *DFS-invar G param* ($s'(\!|state.more := x|\!)$)
$\longrightarrow I\ (s'(\!|state.more := x|\!)))$
by (*auto intro*!: *cross-back-edge*)
with *prereq* **show** *?case* **by** *simp*
qed *fact+*

lemma *establish-invarI-ND-CB* [*case-names prereq-ND prereq-CB init new-discover finish cross-back-edge*]:
— Establish a DFS invariant (new-root/discover and cross/back-edge cases are combined).
assumes *prereq*:
$\bigwedge u\ v\ s.$ *on-discover param u v s* = *on-new-root param v s*
$\bigwedge u\ v\ s.$ *on-back-edge param u v s* = *on-cross-edge param u v s*
assumes *init*: *on-init param* $\leq_n$ *SPEC* ($\lambda x.$ *I* (*empty-state x*))

assumes *new-discover*: $\bigwedge s\ s'\ v.$
$[\![DFS\text{-}invar\ G\ param\ s;\ I\ s;\ cond\ s;\ \neg\ is\text{-}break\ param\ s;$
$v \notin dom\ (discovered\ s);$
$discovered\ s' = (discovered\ s)(v \mapsto counter\ s);\ finished\ s' = finished\ s;$
$counter\ s' = Suc\ (counter\ s);\ stack\ s' = v\#stack\ s;$
$back\text{-}edges\ s' = back\text{-}edges\ s;\ cross\text{-}edges\ s' = cross\text{-}edges\ s;$
$tree\text{-}edges\ s' \supseteq tree\text{-}edges\ s;$
$state.more\ s' = state.more\ s]\!]$
$\Longrightarrow on\text{-}new\text{-}root\ param\ v\ s' \le_n$
$SPEC\ (\lambda x.\ DFS\text{-}invar\ G\ param\ (s'(\!|state.more := x|\!))$
$\longrightarrow I\ (s'(\!|state.more := x|\!)))$
assumes *finish*: $\bigwedge s\ s'\ u.$
$[\![DFS\text{-}invar\ G\ param\ s;\ I\ s;\ cond\ s;\ \neg\ is\text{-}break\ param\ s;$
$stack\ s \neq [];\ u = hd\ (stack\ s);$
$pending\ s\ \text{``}\ \{u\} = \{\};$
$s' = finish\ u\ s]\!]$
$\Longrightarrow on\text{-}finish\ param\ u\ s' \le_n$
$SPEC\ (\lambda x.\ DFS\text{-}invar\ G\ param\ (s'(\!|state.more := x|\!))$
$\longrightarrow I\ (s'(\!|state.more := x|\!)))$
assumes *cross-back-edge*: $\bigwedge s\ s'\ u\ v.$
$[\![DFS\text{-}invar\ G\ param\ s;\ I\ s;\ cond\ s;\ \neg\ is\text{-}break\ param\ s;$
$stack\ s \neq [];\ (u,\ v) \in pending\ s;\ u = hd\ (stack\ s);$
$v \in dom\ (discovered\ s);$
$discovered\ s' = discovered\ s;\ finished\ s' = finished\ s;$
$stack\ s' = stack\ s;\ tree\text{-}edges\ s' = tree\text{-}edges\ s;\ counter\ s' = counter\ s;$
$pending\ s' = pending\ s - \{(u,v)\};$
$cross\text{-}edges\ s' \cup back\text{-}edges\ s' = cross\text{-}edges\ s \cup back\text{-}edges\ s \cup \{(u,v)\};$
$state.more\ s' = state.more\ s\ ]\!]$
$\Longrightarrow on\text{-}cross\text{-}edge\ param\ u\ v\ s' \le_n$
$SPEC\ (\lambda x.\ DFS\text{-}invar\ G\ param\ (s'(\!|state.more := x|\!))$
$\longrightarrow I\ (s'(\!|state.more := x|\!)))$
shows *is-invar I*
proof (*induct rule*: *establish-invarI-ND*)
case *cross-edge* **thus** *?case* **by** (*auto intro*!: *cross-back-edge*)
next
case (*back-edge s s' u v*) **hence**
$on\text{-}cross\text{-}edge\ param\ u\ v\ s' \le_n$
$SPEC\ (\lambda x.\ DFS\text{-}invar\ G\ param\ (s'(\!|state.more := x|\!))$
$\longrightarrow I\ (s'(\!|state.more := x|\!)))$
by (*auto intro*!: *cross-back-edge*)
with *prereq* **show** *?case* **by** *simp*
qed *fact+*

lemma *is-invarI-full* [*case-names init new-root finish cross-edge back-edge discover*]:
— Establish a DFS invariant not taking into account the parameterization.
assumes *init*: $\bigwedge e.\ I\ (empty\text{-}state\ e)$
assumes *new-root*: $\bigwedge s\ s'\ v0\ e.$

$\llbracket I\ s;\ cond\ s;\ DFS\text{-}invar\ G\ param\ s;\ DFS\text{-}invar\ G\ param\ s';$
$stack\ s = [];\ v0 \notin dom\ (discovered\ s);\ v0 \in V0;$
$s' = new\text{-}root\ v0\ s(\!|state.more := e|\!)\rrbracket$
$\Longrightarrow I\ s'$

and *finish*: $\bigwedge s\ s'\ u\ e.$
$\llbracket I\ s;\ cond\ s;\ DFS\text{-}invar\ G\ param\ s;\ DFS\text{-}invar\ G\ param\ s';$
$stack\ s \neq [];\ pending\ s\ ``\ \{u\} = \{\};$
$u = hd\ (stack\ s);\ s' = finish\ u\ s(\!|state.more := e|\!)\rrbracket$
$\Longrightarrow I\ s'$

and *cross-edge*: $\bigwedge s\ s'\ u\ v\ e.$
$\llbracket I\ s;\ cond\ s;\ DFS\text{-}invar\ G\ param\ s;\ DFS\text{-}invar\ G\ param\ s';$
$stack\ s \neq [];\ v \in pending\ s\ ``\ \{u\};\ v \in dom\ (discovered\ s);$
$v \in dom\ (finished\ s);$
$u = hd\ (stack\ s);$
$s' = (cross\text{-}edge\ u\ v\ (s(\!|pending := pending\ s - \{(u,v)\}|\!)))(\!|state.more := e|\!)\rrbracket$
$\Longrightarrow I\ s'$

and *back-edge*: $\bigwedge s\ s'\ u\ v\ e.$
$\llbracket I\ s;\ cond\ s;\ DFS\text{-}invar\ G\ param\ s;\ DFS\text{-}invar\ G\ param\ s';$
$stack\ s \neq [];\ v \in pending\ s\ ``\ \{u\};\ v \in dom\ (discovered\ s);\ v \notin dom\ (finished$
$s);$
$u = hd\ (stack\ s);$
$s' = (back\text{-}edge\ u\ v\ (s(\!|pending := pending\ s - \{(u,v)\}|\!)))(\!|state.more := e|\!)\rrbracket$
$\Longrightarrow I\ s'$

and *discover*: $\bigwedge s\ s'\ u\ v\ e.$
$\llbracket I\ s;\ cond\ s;\ DFS\text{-}invar\ G\ param\ s;\ DFS\text{-}invar\ G\ param\ s';$
$stack\ s \neq [];\ v \in pending\ s\ ``\ \{u\};\ v \notin dom\ (discovered\ s);$
$u = hd\ (stack\ s);$
$s' = (discover\ u\ v\ (s(\!|pending := pending\ s - \{(u,v)\}|\!)))(\!|state.more := e|\!)\rrbracket$
$\Longrightarrow I\ s'$

shows *is-invar I*
apply (*rule establish-invarI*)
apply (*blast intro*: *indep-invar-rules assms*)+
done

lemma *is-invarI* [*case-names init new-root finish visited discover*]:
— Establish a DFS invariant not taking into account the parameterization, cross/back-edges combined.

assumes *init'*: $\bigwedge e.\ I\ (empty\text{-}state\ e)$
and *new-root'*: $\bigwedge s\ s'\ v0\ e.$
$\llbracket I\ s;\ cond\ s;\ DFS\text{-}invar\ G\ param\ s;\ DFS\text{-}invar\ G\ param\ s';$
$stack\ s = [];\ v0 \notin dom\ (discovered\ s);\ v0 \in V0;$
$s' = new\text{-}root\ v0\ s(\!|state.more := e|\!)\rrbracket$
$\Longrightarrow I\ s'$

and *finish'*: $\bigwedge s\ s'\ u\ e.$
$\llbracket I\ s;\ cond\ s;\ DFS\text{-}invar\ G\ param\ s;\ DFS\text{-}invar\ G\ param\ s';$
$stack\ s \neq [];\ pending\ s\ ``\ \{u\} = \{\};$
$u = hd\ (stack\ s);\ s' = finish\ u\ s(\!|state.more := e|\!)\rrbracket$
$\Longrightarrow I\ s'$

and *visited'*: $\bigwedge s\ s'\ u\ v\ e\ c\ b.$

⟦*I s*; *cond s*; *DFS-invar G param s*; *DFS-invar G param s'*;
stack s ≠ []; *v* ∈ *pending s* `` {*u*}; *v* ∈ *dom* (*discovered s*);
u = *hd* (*stack s*);
cross-edges s ⊆ *c*; *back-edges s* ⊆ *b*;
s' = *s*⦇
pending := *pending s* − {(*u*,*v*)},
state.more := *e*,
cross-edges := *c*,
back-edges := *b*⦈⟧
⟹ *I s'*
and *discover'*: ⋀*s s' u v e*.
⟦*I s*; *cond s*; *DFS-invar G param s*; *DFS-invar G param s'*;
stack s ≠ []; *v* ∈ *pending s* `` {*u*}; *v* ∉ *dom* (*discovered s*);
u = *hd* (*stack s*);
s' = (*discover u v* (*s*⦇*pending* := *pending s* − {(*u*,*v*)}⦈))⦇*state.more* := *e*⦈⟧
⟹ *I s'*
shows *is-invar I*
proof (*induct rule*: *is-invarI-full*)
case (*cross-edge s s' u v e*) **thus** *?case*
apply −
apply (*rule visited'*[*of s s' v u insert* (*u*,*v*) (*cross-edges s*) *back-edges s e*])
apply *clarsimp-all*
done
next
case (*back-edge s s' u v e*) **thus** *?case*
apply −
apply (*rule visited'*[*of s s' v u cross-edges s insert* (*u*,*v*) (*back-edges s*) *e*])
apply *clarsimp-all*
done
qed *fact+*

end

1.1.5 Basic Invariants

We establish some basic invariants

context *param-DFS* **begin**

definition *basic-invar s* ≡
set (*stack s*) = *dom* (*discovered s*) − *dom* (*finished s*) ∧
distinct (*stack s*) ∧
(*stack s* ≠ [] ⟶ *last* (*stack s*) ∈ *V0*) ∧
dom (*finished s*) ⊆ *dom* (*discovered s*) ∧
Domain (*pending s*) ⊆ *dom* (*discovered s*) − *dom* (*finished s*) ∧
pending s ⊆ *E*

lemma *i-basic-invar*: *is-invar basic-invar*
unfolding *basic-invar-def*[*abs-def*]
apply (*induction rule*: *is-invarI*)

```
    apply (clarsimp-all simp: neq-Nil-conv last-tl)
    apply blast+
    done
end

context DFS-invar begin
  lemmas basic-invar = make-invar-thm[OF i-basic-invar]

  lemma pending-ssE: pending s ⊆ E
    using basic-invar
    by (auto simp: basic-invar-def)

  lemma pendingD:
    (u,v)∈pending s ⟹ (u,v)∈E ∧ u∈dom (discovered s)
    using basic-invar
    by (auto simp: basic-invar-def)

  lemma stack-set-def:
    set (stack s) = dom (discovered s) − dom (finished s)
    using basic-invar
    by (simp add: basic-invar-def)

  lemma stack-discovered:
    set (stack s) ⊆ dom (discovered s)
    using stack-set-def
    by auto

  lemma stack-distinct:
    distinct (stack s)
    using basic-invar
    by (simp add: basic-invar-def)

  lemma last-stack-in-V0:
    stack s ≠ [] ⟹ last (stack s) ∈ V0
    using basic-invar
    by (simp add: basic-invar-def)

  lemma stack-not-finished:
    x ∈ set (stack s) ⟹ x ∉ dom (finished s)
    using stack-set-def
    by auto

  lemma discovered-not-stack-imp-finished:
    x ∈ dom (discovered s) ⟹ x ∉ set (stack s) ⟹ x ∈ dom (finished s)
    using stack-set-def
    by auto

  lemma finished-discovered:
    dom (finished s) ⊆ dom (discovered s)
```

using *basic-invar*
by (*auto simp add*: *basic-invar-def*)

lemma *finished-no-pending*:
$v \in dom\ (finished\ s) \Longrightarrow pending\ s\ ``\ \{v\} = \{\}$
using *basic-invar*
by (*auto simp add*: *basic-invar-def*)

lemma *discovered-eq-finished-un-stack*:
$dom\ (discovered\ s) = dom\ (finished\ s) \cup set\ (stack\ s)$
using *stack-set-def finished-discovered* **by** *auto*

lemma *pending-on-stack*:
$(v,w) \in pending\ s \Longrightarrow v \in set\ (stack\ s)$
using *basic-invar*
by (*auto simp add*: *basic-invar-def*)

lemma *empty-stack-imp-empty-pending*:
$stack\ s = [] \Longrightarrow pending\ s = \{\}$
using *pending-on-stack*
by *auto*
end

context *param-DFS* **begin**

lemma *i-discovered-reachable*:
$is\text{-}invar\ (\lambda s.\ dom\ (discovered\ s) \subseteq reachable)$
proof (*induct rule*: *is-invarI*)
case (*discover s*) **then interpret** *i*: *DFS-invar* **where** *s=s* **by** *simp*
from *discover* **show** *?case*
apply (*clarsimp dest*!: *i.pendingD*)
by (*metis contra-subsetD list.set-sel*(*1*) *rtrancl-image-advance i.stack-discovered*)
qed *auto*

definition *discovered-closed* $s \equiv$
$E``dom\ (finished\ s) \subseteq dom\ (discovered\ s)$
$\wedge\ (E - pending\ s)\ ``\ set\ (stack\ s) \subseteq dom\ (discovered\ s)$

lemma *i-discovered-closed*: *is-invar discovered-closed*
proof (*induct rule*: *is-invarI*)
case (*finish s s'*)
hence $(E - pending\ s)\ ``\ set\ (stack\ s) \subseteq dom\ (discovered\ s)$
by (*simp add*: *discovered-closed-def*)
moreover from *finish* **have** $set\ (stack\ s') \subseteq set\ (stack\ s)$
by (*auto simp add*: *neq-Nil-conv cond-def*)
ultimately have $(E - pending\ s')\ ``\ set\ (stack\ s') \subseteq dom\ (discovered\ s')$

using *finish*
by *simp blast*

moreover
from ‹*stack s* $\neq$ []› *finish* **have** $E``dom\ (finished\ s') \subseteq dom\ (discovered\ s')$
apply (*cases stack s*) **apply** *simp*
apply (*simp add*: *discovered-closed-def*)
apply (*blast*)
done
ultimately show *?case* **by** (*simp add*: *discovered-closed-def*)
qed (*auto simp add*: *discovered-closed-def cond-def*)

lemma *i-discovered-finite*: *is-invar* ($\lambda s.$ *finite* (*dom* (*discovered s*)))
by (*induction rule*: *is-invarI*) *auto*

end

context *DFS-invar*
begin

lemmas *discovered-reachable* =
i-discovered-reachable [*THEN make-invar-thm*]

lemma *stack-reachable*: *set* (*stack s*) $\subseteq$ *reachable*
using *stack-discovered discovered-reachable* **by** *blast*

lemmas *discovered-closed* = *i-discovered-closed*[*THEN make-invar-thm*]

lemmas *discovered-finite*[*simp*, *intro*!] = *i-discovered-finite*[*THEN make-invar-thm*]
lemma *finished-finite*[*simp*, *intro*!]: *finite* (*dom* (*finished s*))
using *finished-discovered discovered-finite* **by** (*rule finite-subset*)

lemma *finished-closed*:
$E\ ``\ dom\ (finished\ s) \subseteq dom\ (discovered\ s)$
using *discovered-closed*[*unfolded discovered-closed-def*]
by *auto*

lemma *finished-imp-succ-discovered*:
$v \in dom\ (finished\ s) \Longrightarrow w \in succ\ v \Longrightarrow w \in dom\ (discovered\ s)$
using *discovered-closed*[*unfolded discovered-closed-def*]
by *auto*

lemma *pending-reachable*: *pending s* $\subseteq$ *reachable* $\times$ *reachable*
using *pendingD discovered-reachable*
by (*fast intro*: *rtrancl-image-advance-rtrancl*)

lemma *pending-finite*[*simp*, *intro*!]: *finite* (*pending s*)
proof −
have *pending s* $\subseteq$ (*SIGMA u*:*dom* (*discovered s*). $E``\{u\}$)

```
      by (auto dest: pendingD)
    also have finite ...
      apply rule
      apply (rule discovered-finite)
      using discovered-reachable
      by (blast intro: finitely-branching)
    finally (finite-subset) show ?thesis .
  qed

  lemma no-pending-imp-succ-discovered:
    assumes u ∈ dom (discovered s)
    and pending s `` {u} = {}
    and v ∈ succ u
    shows v ∈ dom (discovered s)
  proof (cases u ∈ dom (finished s))
    case True with finished-imp-succ-discovered assms show ?thesis by simp
  next
    case False with stack-set-def assms have u ∈ set (stack s) by auto
     with assms discovered-closed[unfolded discovered-closed-def] show ?thesis by
blast
  qed

  lemma nc-finished-eq-reachable:
    assumes NC: ¬cond s ¬is-break param s
    shows dom (finished s) = reachable
  proof -
    from NC basic-invar
    have [simp]: stack s = [] dom (discovered s) = dom (finished s) and SS: V0 ⊆
dom (discovered s)
      unfolding basic-invar-def cond-alt by auto

    show dom (finished s) = reachable
    proof
      from discovered-reachable show dom (finished s) ⊆ reachable
        by simp
    next
      from discovered-closed have E``(dom (finished s)) ⊆ dom (finished s)
        unfolding discovered-closed-def by auto
      with SS show reachable ⊆ dom (finished s)
        by (simp, metis rtrancl-reachable-induct)
    qed
  qed

  lemma nc-V0-finished:
    assumes NC: ¬ cond s ¬ is-break param s
    shows V0 ⊆ dom (finished s)
    using nc-finished-eq-reachable[OF NC]
    by blast
```

lemma *nc-discovered-eq-finished*:
  assumes *NC*: ¬ *cond s* ¬ *is-break param s*
  shows *dom* (*discovered s*) = *dom* (*finished s*)
  using *finished-discovered*
  using *nc-finished-eq-reachable*[*OF NC*] *discovered-reachable*
  by *blast*

lemma *nc-discovered-eq-reachable*:
  assumes *NC*: ¬ *cond s* ¬ *is-break param s*
  shows *dom* (*discovered s*) = *reachable*
  using *NC*
  using *nc-discovered-eq-finished nc-finished-eq-reachable*
  by *blast*

lemma *nc-fin-closed*:
  assumes *NC*: ¬*cond s*
  assumes *NB*: ¬*is-break param s*
  shows *E``dom* (*finished s*) ⊆ *dom* (*finished s*)
  using *finished-imp-succ-discovered*
  by (*auto simp*: *nc-discovered-eq-finished*[*OF NC NB*])

end

1.1.6 Total Correctness

We can show termination of the DFS algorithm, independently of the parameterization

context *param-DFS* **begin**
  definition *param-dfs-variant* ≡ *inv-image*
    (*finite-psupset reachable* <∗*lex*∗> *finite-psubset* <∗*lex*∗> *less-than*)
    (λ*s*. (*dom* (*discovered s*), *pending s*, *length* (*stack s*)))

  lemma *param-dfs-variant-wf*[*simp*, *intro*!]:
    assumes [*simp*, *intro*!]: *finite reachable*
    shows *wf param-dfs-variant*
    unfolding *param-dfs-variant-def*
    by *auto*

  lemma *param-dfs-variant-step*:
    assumes *A*: *DFS-invar G param s cond s nofail it-dfs*
    shows *step s* ≤ *SPEC* (λ*s'*. (*s'*,*s*)∈*param-dfs-variant*)
  proof −
    interpret *DFS-invar G param s* **by** *fact*

    from *A* **show** *?thesis*
      unfolding *rwof-eq-DFS-invar*[*symmetric*] *it-dfs-def*
      apply −
      apply (*drule* (*2*) *WHILE-nofail-imp-rwof-nofail*)
      unfolding *step-def get-new-root-def do-defs get-pending-def*

```
    unfolding param-dfs-variant-def
    apply refine-vcg
    using discovered-reachable

    apply (auto
      split: option.splits
      simp: refine-pw-simps pw-le-iff is-discovered-def finite-psupset-def
    ) [1]
    apply (auto simp: refine-pw-simps pw-le-iff is-empty-stack-def) []
    apply simp-all

    apply (auto
      simp: refine-pw-simps pw-le-iff is-discovered-def
      split: if-split-asm
      ) [2]

    apply (clarsimp simp: refine-pw-simps pw-le-iff is-discovered-def)
    using discovered-reachable pending-reachable
    apply (auto
      simp: is-discovered-def
      simp: refine-pw-simps pw-le-iff finite-psupset-def
      split: if-split-asm)
    done
  qed


end


context param-DFS begin
  lemma it-dfsT-eq-it-dfs:
    assumes [simp, intro!]: finite reachable
    shows it-dfsT = it-dfs
  proof −
    have it-dfs ≤ it-dfsT
      unfolding it-dfs-def it-dfsT-def WHILE-def WHILET-def
      apply (rule bind-mono)
      apply simp
      apply (rule WHILEI-le-WHILEIT)
      done
    also have it-dfsT ≤ it-dfs
    proof (cases nofail it-dfs)
      case False thus ?thesis by (simp add: not-nofail-iff)
    next
      case True

      show ?thesis
        unfolding it-dfsT-def it-dfs-def
        apply (rule bind-mono)
```

```
      apply simp
      apply (subst WHILET-eq-WHILE-tproof[
        where I=DFS-invar G param
        and V=param-dfs-variant
        ])
      apply auto []

      apply (subst rwof-eq-DFS-invar[symmetric])
      using rwof-init[OF True[unfolded it-dfs-def]]
      apply (fastforce dest: order-trans) []

      apply (rule SPEC-rule-conjI)
        apply (rule DFS-invar-step[OF True], assumption+) []
        apply (rule param-dfs-variant-step, (assumption|rule True)+) []

      apply simp
      done
  qed
  finally show ?thesis by simp
 qed
end
```

1.1.7 Non-Failing Parameterization

The proofs so far have been done modulo failure of the parameterization. In this locale, we assume that the parameterization does not fail, and derive the correctness proof of the DFS algorithm wrt. its invariant.

```
locale DFS =
 param-DFS G param
 for G :: ('v, 'more) graph-rec-scheme
 and param :: ('v,'es) parameterization
 +
 assumes nofail-on-init:
   nofail (on-init param)

 assumes nofail-on-new-root:
   pre-on-new-root v0 s ⟹ nofail (on-new-root param v0 s)

 assumes nofail-on-finish:
   pre-on-finish u s ⟹ nofail (on-finish param u s)

 assumes nofail-on-cross-edge:
   pre-on-cross-edge u v s ⟹ nofail (on-cross-edge param u v s)

 assumes nofail-on-back-edge:
   pre-on-back-edge u v s ⟹ nofail (on-back-edge param u v s)

 assumes nofail-on-discover:
   pre-on-discover u v s ⟹ nofail (on-discover param u v s)
```

begin
lemmas *nofails* = *nofail-on-init nofail-on-new-root nofail-on-finish nofail-on-cross-edge nofail-on-back-edge nofail-on-discover*

lemma *init-leof-invar*: *init* $\leq_n$ *SPEC* (*DFS-invar G param*)
unfolding *rwof-eq-DFS-invar*[*symmetric*]
by (*rule rwof-leof-init*)

lemma *it-dfs-eq-spec*: *it-dfs* = *SPEC* (λ*s*. *DFS-invar G param s* $\wedge$ $\neg$*cond s*)
unfolding *rwof-eq-DFS-invar*[*symmetric*] *it-dfs-def*
apply (*rule nofail-WHILE-eq-rwof*)
apply (*subst WHILE-eq-I-rwof*)
unfolding *rwof-eq-DFS-invar*
apply (*rule SPEC-nofail*[**where** Φ=λ-. *True*])
apply (*refine-vcg leofD*[*OF - init-leof-invar*, *THEN weaken-SPEC*])
apply (*simp add*: *init-def refine-pw-simps nofail-on-init*)
apply (*rule DFS-invar-step*′)
apply (*simp add*: *step-def refine-pw-simps nofail-on-init do-defs get-pending-def get-new-root-def pred-defs split*: *option.split*)
apply (*intro allI conjI impI nofails*)
apply (*auto simp add*: *pre-on-defs*)
done

lemma *it-dfs-correct*: *it-dfs* $\leq$ *SPEC* (λ*s*. *DFS-invar G param s* $\wedge$ $\neg$*cond s*)
by (*simp add*: *it-dfs-eq-spec*)

lemma *it-dfs-SPEC*:
assumes $\bigwedge$*s*. ⟦*DFS-invar G param s*; $\neg$*cond s*⟧ $\Longrightarrow$ *P s*
shows *it-dfs* $\leq$ *SPEC P*
using *weaken-SPEC*[*OF it-dfs-correct*]
using *assms*
by *blast*

lemma *it-dfsT-correct*:
assumes *finite reachable*
shows *it-dfsT* $\leq$ *SPEC* (λ*s*. *DFS-invar G param s* $\wedge$ $\neg$*cond s*)
apply (*subst it-dfsT-eq-it-dfs*[*OF assms*])
by (*rule it-dfs-correct*)

lemma *it-dfsT-SPEC*:
assumes *finite reachable*
assumes $\bigwedge$*s*. ⟦*DFS-invar G param s*; $\neg$*cond s*⟧ $\Longrightarrow$ *P s*
shows *it-dfsT* $\leq$ *SPEC P*
apply (*subst it-dfsT-eq-it-dfs*[*OF assms*(*1*)])
using *assms*(*2*)
by (*rule it-dfs-SPEC*)

end

end

1.2 Basic Invariant Library

theory *DFS-Invars-Basic*
imports *../Param-DFS*
begin

We provide more basic invariants of the DFS algorithm

1.2.1 Basic Timing Invariants

abbreviation *the-discovered* $s\ v \equiv$ *the* (*discovered* $s\ v$)
abbreviation *the-finished* $s\ v \equiv$ *the* (*finished* $s\ v$)

locale *timing-syntax*
begin

  notation *the-discovered* (δ)
  notation *the-finished* (φ)
end

context *param-DFS* **begin context begin interpretation** *timing-syntax* **.**

  definition *timing-common-inv* $s \equiv$
  — $\delta\ s\ v < \varphi\ s\ v$
   ($\forall v \in$ *dom* (*finished* s). $\delta\ s\ v < \varphi\ s\ v$)

  — $v \neq w \longrightarrow \delta\ s\ v \neq \delta\ s\ w \wedge \varphi\ s\ v \neq \varphi\ s\ w$
  — Can't use *card dom* = *card ran* as the maps may be infinite ...
  $\wedge$ ($\forall v \in$ *dom* (*discovered* s). $\forall w \in$ *dom* (*discovered* s). $v \neq w \longrightarrow \delta\ s\ v \neq \delta\ s\ w$)
  $\wedge$ ($\forall v \in$ *dom* (*finished* s). $\forall w \in$ *dom* (*finished* s). $v \neq w \longrightarrow \varphi\ s\ v \neq \varphi\ s\ w$)

  — $\delta\ s\ v <$ *counter* $\wedge\ \varphi\ s\ v <$ *counter*
  $\wedge$ ($\forall v \in$ *dom* (*discovered* s). $\delta\ s\ v <$ *counter* s)
  $\wedge$ ($\forall v \in$ *dom* (*finished* s). $\varphi\ s\ v <$ *counter* s)

  $\wedge$ ($\forall v \in$ *dom* (*finished* s). $\forall w \in$ *succ* v. $\delta\ s\ w < \varphi\ s\ v$)

  lemma *timing-common-inv*:
    is-invar timing-common-inv
  proof (*induction rule*: *is-invarI*)
    case (*finish* $s\ s'$) **then interpret** *DFS-invar* **where** s=s **by** *simp*

    from *finish* **have** *NE*: *stack* $s \neq$ [] **by** (*simp add*: *cond-alt*)

have ∗: *hd* (*stack s*) ∉ *dom* (*finished s*) *hd* (*stack s*) ∈ *dom* (*discovered s*)
  using *stack-not-finished stack-discovered hd-in-set*[*OF NE*]
  by *blast*+

from *discovered-closed* **have**
  (*E* − *pending s*) '' {*hd* (*stack s*)} ⊆ *dom* (*discovered s*)
  using *hd-in-set*[*OF NE*]
  by (*auto simp add*: *discovered-closed-def*)
hence *succ-hd*: *pending s* '' {*hd* (*stack s*)} = {}
  ⟹ *succ* (*hd* (*stack s*)) ⊆ *dom* (*discovered s*)
  by *blast*

from *finish* **show** *?case*
    apply (*simp add*: *timing-common-inv-def*)
    apply (*intro conjI*)
    using ∗ **apply** *simp*
    using ∗ **apply** *simp*
    apply (*metis less-irrefl*)
    apply (*metis less-irrefl*)
    apply (*metis less-SucI*)
    apply (*metis less-SucI*)
    apply (*blast dest*!: *succ-hd*)
    using ∗ **apply** *simp*
    done
next
  case (*discover s*) **then interpret** *DFS-invar* **where** *s*=*s* **by** *simp*
  from *discover* **show** *?case*
    apply (*simp add*: *timing-common-inv-def*)
    apply (*intro conjI*)
    using *finished-discovered* **apply** *fastforce*
    apply (*metis less-irrefl*)
    apply (*metis less-irrefl*)
    apply (*metis less-SucI*)
    apply (*metis less-SucI*)
    using *finished-imp-succ-discovered* **apply** *fastforce*
    done
next
  case (*new-root s s' v0*) **then interpret** *DFS-invar* **where** *s*=*s* **by** *simp*
  from *new-root* **show** *?case*
    apply (*simp add*: *timing-common-inv-def*)
    apply (*intro conjI*)
    using *finished-discovered* **apply** *fastforce*
    apply (*metis less-irrefl*)
    apply (*metis less-irrefl*)
    apply (*metis less-SucI*)
    apply (*metis less-SucI*)
    using *finished-imp-succ-discovered* **apply** *fastforce*
    done
qed (*simp-all add*: *timing-common-inv-def*)

end end

context *DFS-invar* **begin context begin interpretation** *timing-syntax* **.**

lemmas *s-timing-common-inv* =
timing-common-inv[*THEN make-invar-thm*]

lemma *timing-less-counter*:
$v \in dom\ (discovered\ s) \Longrightarrow \delta\ s\ v < counter\ s$
$v \in dom\ (finished\ s) \Longrightarrow \varphi\ s\ v < counter\ s$
using *s-timing-common-inv*
by (*auto simp add*: *timing-common-inv-def*)

lemma *disc-lt-fin*:
$v \in dom\ (finished\ s) \Longrightarrow \delta\ s\ v < \varphi\ s\ v$
using *s-timing-common-inv*
by (*auto simp add*: *timing-common-inv-def*)

lemma *disc-unequal*:
assumes $v \in dom\ (discovered\ s)\ w \in dom\ (discovered\ s)$
and $v \neq w$
shows $\delta\ s\ v \neq \delta\ s\ w$
using *s-timing-common-inv assms*
by (*auto simp add*: *timing-common-inv-def*)

lemma *fin-unequal*:
assumes $v \in dom\ (finished\ s)\ w \in dom\ (finished\ s)$
and $v \neq w$
shows $\varphi\ s\ v \neq \varphi\ s\ w$
using *s-timing-common-inv assms*
by (*auto simp add*: *timing-common-inv-def*)

lemma *finished-succ-fin*:
assumes $v \in dom\ (finished\ s)$
and $w \in succ\ v$
shows $\delta\ s\ w < \varphi\ s\ v$
using *assms s-timing-common-inv*
by (*simp add*: *timing-common-inv-def*)
end end

context *param-DFS* **begin context begin interpretation** *timing-syntax* **.**

lemma *i-prev-stack-discover-all*:
$is\text{-}invar\ (\lambda s.\ \forall\ n < length\ (stack\ s).\ \forall\ v \in set\ (drop\ (Suc\ n)\ (stack\ s)).$
$\delta\ s\ (stack\ s\ !\ n) > \delta\ s\ v)$
proof (*induct rule*: *is-invarI*)
case (*finish s*) **thus** *?case*
by (*cases stack s*) *auto*
next

case (*discover s s' u v*)
hence *EQ*[*simp*]: *discovered s'* = (*discovered s*)(*v* ↦ *counter s*)
stack s' = *v*#*stack s*
by *simp-all*

from *discover* **interpret** *DFS-invar* **where** *s=s* **by** *simp*
from *discover stack-discovered* **have** *v-ni*: $v \notin$ *set* (*stack s*) **by** *auto*

from *stack-discovered timing-less-counter* **have**
$\bigwedge w.\ w \in$ *set* (*stack s*) $\Longrightarrow \delta$ *s w* < *counter s*
by *blast*
with *v-ni* **have** $\bigwedge w.\ w \in$ *set* (*stack s*) $\Longrightarrow \delta\ s'\ w < \delta\ s'\ v$ **by** *auto*
hence $\bigwedge w.\ w \in$ *set* (*drop* (*Suc 0*) (*stack s'*)) $\Longrightarrow \delta\ s'\ w < \delta\ s'$ (*stack s'* ! *0*)
by *auto*

moreover
from *v-ni* **have**
$\bigwedge n.$ ⟦*n* < (*length* (*stack s'*)); *n* > *0*⟧
$\Longrightarrow \delta\ s'$ (*stack s'* ! *n*) = δ *s* (*stack s'* ! *n*)
by *auto*
with *discover*(*1*) *v-ni*
have $\bigwedge n.$ ⟦*n* < (*length* (*stack s'*)) − *1*; *n* > *0*⟧
$\Longrightarrow \forall\ w \in$ *set* (*drop* (*Suc n*) (*stack s'*)). $\delta\ s'$ (*stack s'* ! *n*) > $\delta\ s'\ w$
by (*auto dest*: *in-set-dropD*)
ultimately show *?case*
by (*metis drop-Suc-Cons length-drop length-pos-if-in-set length-tl list.sel*(*3*) *neq0-conv nth-Cons-0 EQ*(*2*) *zero-less-diff*)
qed *simp-all*
end end

context *DFS-invar* **begin context begin interpretation** *timing-syntax* **.**

lemmas *prev-stack-discover-all*
= *i-prev-stack-discover-all*[*THEN make-invar-thm*]

lemma *prev-stack-discover*:
⟦*n* < *length* (*stack s*); *v* ∈ *set* (*drop* (*Suc n*) (*stack s*)) ⟧
$\Longrightarrow \delta$ *s* (*stack s* ! *n*) > δ *s v*
by (*metis prev-stack-discover-all*)

lemma *Suc-stack-discover*:
assumes *n*: *n* < (*length* (*stack s*)) − *1*
shows δ *s* (*stack s* ! *n*) > δ *s* (*stack s* ! *Suc n*)
proof −
from *prev-stack-discover assms* **have**
$\bigwedge v.\ v \in$ *set* (*drop* (*Suc n*) (*stack s*)) $\Longrightarrow \delta$ *s* (*stack s* ! *n*) > δ *s v*
by *fastforce*
moreover from *n* **have** *stack s* ! *Suc n* ∈ *set* (*drop* (*Suc n*) (*stack s*))
using *in-set-conv-nth* **by** *fastforce*

```
  ultimately show ?thesis .
qed

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

lemma stack-nth-order:
  assumes l: i < length (stack s) j < length (stack s)
  shows δ s (stack s ! i) < δ s (stack s ! j) ⟷ i > j (is δ s ?i < δ s ?j ⟷ -)
proof
  assume δ: δ s ?i < δ s ?j

  from l stack-set-def have
    disc: ?i ∈ dom (discovered s) ?j ∈ dom (discovered s)
    by auto
  with disc-unequal[OF disc] δ have i ≠ j by auto

  moreover
  {
    assume i < j

    with l have stack s ! j ∈ set (drop (Suc i) (stack s))
      using in-set-drop-conv-nth[of stack s ! j Suc i stack s]
      by fastforce
    with prev-stack-discover l have δ s (stack s ! j) < δ s (stack s ! i)
      by simp
    with δ have False by simp
  }
  ultimately show i > j by force
next
  assume i > j
  with l have stack s ! i ∈ set (drop (Suc j) (stack s))
    using in-set-drop-conv-nth[of stack s ! i Suc j stack s]
    by fastforce
  with prev-stack-discover l show δ s ?i < δ s ?j by simp
qed
end end
```

1.2.2 Paranthesis Theorem

context *param-DFS* **begin context begin interpretation** *timing-syntax* **.**

definition *parenthesis s* $\equiv$
  $\forall\, v \in dom\ (discovered\ s).\ \forall\, w \in dom\ (discovered\ s).$
   $\delta\ s\ v < \delta\ s\ w \wedge v \in dom\ (finished\ s) \longrightarrow ($
     $\varphi\ s\ v < \delta\ s\ w$ — disjoint
     $\vee\ (\varphi\ s\ v > \delta\ s\ w \wedge w \in dom\ (finished\ s) \wedge \varphi\ s\ w < \varphi\ s\ v))$

lemma *i-parenthesis*: *is-invar parenthesis*
proof (*induct rule*: *is-invarI*)
  case (*finish s s'*)
  hence *EQ*[*simp*]: $discovered\ s' = discovered\ s$
    $counter\ s' = Suc\ (counter\ s)$
    $finished\ s' = (finished\ s)(hd\ (stack\ s) \mapsto counter\ s)$
   by *simp-all*

  from *finish* **interpret** *DFS-invar* **where** *s=s* **by** *simp*
  from *finish* **have** *NE*[*simp*]: $stack\ s \neq []$ **by** (*simp add*: *cond-alt*)

  {
   fix *x y*
   assume *dom*: $x \in dom\ (discovered\ s')\ y \in dom\ (discovered\ s')$
    and δ: $\delta\ s'\ x < \delta\ s'\ y$
    and *f*: $x \in dom\ (finished\ s')$
   hence *neq*: $x \neq y$ **by** *force*

   note *assms* = *dom* δ *f EQ*

   let $?DISJ = \varphi\ s'\ x < \delta\ s'\ y$
   let $?IN = \delta\ s'\ y < \varphi\ s'\ x \wedge y \in dom\ (finished\ s') \wedge \varphi\ s'\ y < \varphi\ s'\ x$

   have $?DISJ \vee ?IN$
   proof (*cases* $x = hd\ (stack\ s)$)
    case *True* **note** *x-is-hd* = *this*
    hence φx: $\varphi\ s'\ x = counter\ s$ **by** *simp*
    from *x-is-hd neq* **have** *y-not-hd*: $y \neq hd\ (stack\ s)$ **by** *simp*

    have $\delta\ s\ y < \varphi\ s'\ x \wedge y \in dom\ (finished\ s) \wedge \varphi\ s\ y < \varphi\ s'\ x$
    proof (*cases* $y \in set\ (stack\ s)$)
     — y on stack is not possible: According to

$\delta\ s'\ x < \delta\ s'\ y$

it is discovered after x ($= hd\ (stack\ s)$)
      case *True* **with** *y-not-hd* **have** $y \in set\ (tl\ (stack\ s))$
       by (*cases stack s*) *simp-all*
      with *tl-lt-stack-hd-discover*[*OF NE*] δ *x-is-hd* **have** $\delta\ s\ y < \delta\ s\ x$
       by *simp*
      with δ **have** *False* **by** *simp*
      thus *?thesis* **..**
     next

```
        case False — y must be a successor of x (= (hd (stack s)))
        from dom have y ∈ dom (discovered s) by simp
        with False discovered-not-stack-imp-finished  have ∗:
          y ∈ dom (finished s)
          by simp
        moreover with timing-less-counter φx have φ s y < φ s′ x by simp
        moreover with ∗ disc-lt-fin φx have δ s y < φ s′ x
          by (metis less-trans)
        ultimately show ?thesis by simp
      qed
      with y-not-hd show ?thesis by simp
    next
      case False note [simp] = this
      show ?thesis
      proof (cases y = hd (stack s))
        case False with finish assms show ?thesis
          by (simp add: parenthesis-def)
      next
        case True with stack-not-finished have y ∉ dom (finished s)
          using hd-in-set[OF NE]
          by auto
        with finish assms have φ s x < δ s y
          unfolding parenthesis-def
          by auto
        hence ?DISJ by simp
        thus ?thesis ..
      qed
    qed
  }
  thus ?case by (simp add: parenthesis-def)
next
  case (discover s s′ u v)
  hence EQ[simp]: discovered s′ = (discovered s)(v ↦ counter s)
                  finished s′ = finished s
                  counter s′ = Suc (counter s)
    by simp-all

  from discover interpret DFS-invar where s=s by simp
  from discover finished-discovered have
    V′: v ∉ dom (discovered s) v ∉ dom (finished s)
    by auto

  {
    fix x y
    assume  dom: x ∈ dom (discovered s′) y ∈ dom (discovered s′)
      and δ: δ s′ x < δ s′ y
      and f: x ∈ dom (finished s′)

    let ?DISJ = φ s′ x < δ s′ y
```

let *?IN* = $\delta\ s'\ y < \varphi\ s'\ x \wedge y \in$ *dom* (*finished* s') $\wedge\ \varphi\ s'\ y < \varphi\ s'\ x$

from *dom* V' *f* **have** *x*: $x \in$ *dom* (*discovered* s)$x \neq v$ **by** *auto*

have *?DISJ* $\vee$ *?IN*
proof (*cases* $y = v$)
case *True* **hence** $\delta\ s'\ y =$ *counter* s **by** *simp*
moreover from *timing-less-counter x f* **have** $\varphi\ s'\ x <$ *counter* s **by** *auto*
ultimately have *?DISJ* **by** *simp*
thus *?thesis* **..**
next
case *False* **with** *dom* **have** $y \in$ *dom* (*discovered* s) **by** *simp*
with *discover False* δ *f x* **show** *?thesis* **by** (*simp add*: *parenthesis-def*)
qed
}
thus *?case* **by** (*simp add*: *parenthesis-def*)
next
case (*new-root s s' v0*)
then interpret *DFS-invar* **where** *s=s* **by** *simp*

from *finished-discovered new-root* **have** *v0* $\notin$ *dom* (*finished* s') **by** *auto*
with *new-root timing-less-counter* **show** *?case* **by** (*simp add*: *parenthesis-def*)
qed (*simp-all add*: *parenthesis-def*)
end end

context *DFS-invar* **begin context begin interpretation** *timing-syntax* **.**

lemma *parenthesis*:
assumes $v \in$ *dom* (*finished* s) $w \in$ *dom* (*discovered* s)
and $\delta\ s\ v < \delta\ s\ w$
shows $\varphi\ s\ v < \delta\ s\ w$ — disjoint
$\vee$ ($\varphi\ s\ v > \delta\ s\ w \wedge w \in$ *dom* (*finished* s) $\wedge\ \varphi\ s\ w < \varphi\ s\ v$)
using *assms*
using *i-parenthesis*[*THEN make-invar-thm*]
using *finished-discovered*
unfolding *parenthesis-def*
by *blast*

lemma *parenthesis-contained*:
assumes $v \in$ *dom* (*finished* s) $w \in$ *dom* (*discovered* s)
and $\delta\ s\ v < \delta\ s\ w\ \varphi\ s\ v > \delta\ s\ w$
shows $w \in$ *dom* (*finished* s) $\wedge\ \varphi\ s\ w < \varphi\ s\ v$
using *parenthesis assms*
by *force*

lemma *parenthesis-disjoint*:
assumes $v \in$ *dom* (*finished* s) $w \in$ *dom* (*discovered* s)
and $\delta\ s\ v < \delta\ s\ w\ \varphi\ s\ w > \varphi\ s\ v$
shows $\varphi\ s\ v < \delta\ s\ w$

using *parenthesis assms*
by *force*

lemma *finished-succ-contained*:
assumes $v \in$ *dom* (*finished s*)
and $w \in$ *succ v*
and $\delta\ s\ v < \delta\ s\ w$
shows $w \in$ *dom* (*finished s*) $\wedge\ \varphi\ s\ w < \varphi\ s\ v$
using *finished-succ-fin finished-imp-succ-discovered parenthesis-contained*
using *assms*
by *metis*

end end

1.2.3 Edge Types

context *param-DFS*
begin
abbreviation *edges s* $\equiv$ *tree-edges s* $\cup$ *cross-edges s* $\cup$ *back-edges s*

lemma *is-invar* ($\lambda s.$ *finite* (*edges s*))
by (*induction rule*: *establish-invarI*) *auto*

Sometimes it's useful to just chose between tree-edges and non-tree.

lemma *edgesE-CB*:
assumes $x \in$ *edges s*
and $x \in$ *tree-edges s* $\Longrightarrow P$
and $x \in$ *cross-edges s* $\cup$ *back-edges s* $\Longrightarrow P$
shows P
using *assms* **by** *auto*

definition *edges-basic s* $\equiv$
Field (*back-edges s*) $\subseteq$ *dom* (*discovered s*) $\wedge$ *back-edges s* $\subseteq E -$ *pending s*
$\wedge$ *Field* (*cross-edges s*) $\subseteq$ *dom* (*discovered s*) $\wedge$ *cross-edges s* $\subseteq E -$ *pending s*
$\wedge$ *Field* (*tree-edges s*) $\subseteq$ *dom* (*discovered s*) $\wedge$ *tree-edges s* $\subseteq E -$ *pending s*
$\wedge$ *back-edges s* $\cap$ *cross-edges s* $= \{\}$
$\wedge$ *back-edges s* $\cap$ *tree-edges s* $= \{\}$
$\wedge$ *cross-edges s* $\cap$ *tree-edges s* $= \{\}$

lemma *i-edges-basic*:
is-invar edges-basic
unfolding *edges-basic-def*[*abs-def*]
proof (*induct rule*: *is-invarI-full*)
case (*back-edge s*)
then interpret *DFS-invar* **where** *s=s* **by** *simp*
from *back-edge* **show** *?case* **by** (*auto dest*: *pendingD*)
next

case (*cross-edge s*)
then interpret *DFS-invar* **where** *s=s* **by** *simp*
from *cross-edge* **show** *?case* **by** (*auto dest*: *pendingD*)
next
case (*discover s*)
then interpret *DFS-invar* **where** *s=s* **by** *simp*
from *discover* **show** *?case*

apply (*simp add*: *Field-def Range-def Domain-def*)
apply (*drule pendingD*) **apply** *simp*
by (*blast*)
next
case (*new-root s*)
thus *?case* **by** (*simp add*: *Field-def*) *blast*
qed *auto*

lemmas (**in** *DFS-invar*) *edges-basic* = *i-edges-basic*[*THEN make-invar-thm*]

lemma *i-edges-covered*:
is-invar ($\lambda s.$ ($E \cap$ *dom* (*discovered s*) $\times$ *UNIV*) $-$ *pending s* = *edges s*)
proof (*induction rule*: *is-invarI-full*)
case (*new-root s s' v0*)
interpret *DFS-invar G param s* **by** *fact*

from *new-root empty-stack-imp-empty-pending*
have [*simp*]: *pending s* = {} **by** *simp*

from ‹*v0* $\notin$ *dom* (*discovered s*)›
have [*simp*]: $E \cap$ *insert v0* (*dom* (*discovered s*)) $\times$ *UNIV* $-$ {*v0*} $\times$ *succ v0*
= $E \cap$ *dom* (*discovered s*) $\times$ *UNIV* **by** *auto*

from *new-root* **show** *?case* **by** *simp*
next
case (*cross-edge s s' u v*)
interpret *DFS-invar G param s* **by** *fact*

from *cross-edge stack-discovered* **have** *u* $\in$ *dom* (*discovered s*)
by (*cases stack s*) *auto*

with *cross-edge*(*2*$-$) *pending-ssE* **have**
$E \cap$ *dom* (*discovered s*) $\times$ *UNIV* $-$ (*pending s* $-$ {(*hd* (*stack s*), *v*)})
= *insert* (*hd* (*stack s*), *v*) ($E \cap$ *dom* (*discovered s*) $\times$ *UNIV* $-$ *pending s*)
by *auto*

thus *?case* **using** *cross-edge* **by** *simp*
next
case (*back-edge s s' u v*)
interpret *DFS-invar G param s* **by** *fact*

from *back-edge stack-discovered* **have** $u \in dom\ (discovered\ s)$
  by (*cases stack s*) *auto*

with *back-edge(2−) pending-ssE* **have**
  $E \cap dom\ (discovered\ s) \times UNIV - (pending\ s - \{(hd\ (stack\ s),\ v)\})$
  $= insert\ (hd\ (stack\ s),\ v)\ (E \cap dom\ (discovered\ s) \times UNIV - pending\ s)$
  by *auto*

thus *?case* **using** *back-edge* **by** *simp*
next
case $(discover\ s\ s'\ u\ v)$
interpret *DFS-invar G param s* **by** *fact*

from *discover stack-discovered* **have** $u \in dom\ (discovered\ s)$
  by (*cases stack s*) *auto*

with *discover(2−) pending-ssE* **have**
  $E \cap insert\ v\ (dom\ (discovered\ s)) \times UNIV$
    $- (pending\ s - \{(hd\ (stack\ s),\ v)\} \cup \{v\} \times succ\ v)$
  $= insert\ (hd\ (stack\ s),\ v)\ (E \cap dom\ (discovered\ s) \times UNIV - pending\ s)$
  by *auto*

thus *?case* **using** *discover* **by** *simp*
qed *simp-all*
end

context *DFS-invar* **begin**

lemmas *edges-covered* =
  i-edges-covered[*THEN make-invar-thm*]

lemma *edges-ss-reachable-edges*:
  $edges\ s \subseteq E \cap reachable \times UNIV$
  using *edges-covered discovered-reachable*
  by (*fast intro*: *rtrancl-image-advance-rtrancl*)

lemma *nc-edges-covered*:
  assumes ¬*cond s* ¬*is-break param s*
  shows $E \cap reachable \times UNIV = edges\ s$
proof −
  from *assms* **have** [*simp*]: *stack s* = []
    unfolding *cond-def* **by** (*auto simp*: *pred-defs*)
  hence [*simp*]: *pending s* = {} **by** (*rule empty-stack-imp-empty-pending*)

  from *edges-covered nc-discovered-eq-reachable*[*OF assms*]
  show *?thesis* **by** *simp*
qed

lemma
tree-edges-ssE: *tree-edges s* $\subseteq$ *E* **and**
tree-edges-not-pending: *tree-edges s* $\subseteq$ $-$ *pending s* **and**
tree-edge-is-succ: $(v,w) \in$ *tree-edges s* $\Longrightarrow$ $w \in$ *succ v* **and**
tree-edges-discovered: *Field* (*tree-edges s*) $\subseteq$ *dom* (*discovered s*) **and**

cross-edges-ssE: *cross-edges s* $\subseteq$ *E* **and**
cross-edges-not-pending: *cross-edges s* $\subseteq$ $-$ *pending s* **and**
cross-edge-is-succ: $(v,w) \in$ *cross-edges s* $\Longrightarrow$ $w \in$ *succ v* **and**
cross-edges-discovered: *Field* (*cross-edges s*) $\subseteq$ *dom* (*discovered s*) **and**

back-edges-ssE: *back-edges s* $\subseteq$ *E* **and**
back-edges-not-pending: *back-edges s* $\subseteq$ $-$ *pending s* **and**
back-edge-is-succ: $(v,w) \in$ *back-edges s* $\Longrightarrow$ $w \in$ *succ v* **and**
back-edges-discovered: *Field* (*back-edges s*) $\subseteq$ *dom* (*discovered s*)
using *edges-basic*
unfolding *edges-basic-def*
by *auto*

lemma *edges-disjoint*:
back-edges s $\cap$ *cross-edges s* $= \{\}$
back-edges s $\cap$ *tree-edges s* $= \{\}$
cross-edges s $\cap$ *tree-edges s* $= \{\}$
using *edges-basic*
unfolding *edges-basic-def*
by *auto*

lemma *tree-edge-imp-discovered*:
$(v,w) \in$ *tree-edges s* $\Longrightarrow$ $v \in$ *dom* (*discovered s*)
$(v,w) \in$ *tree-edges s* $\Longrightarrow$ $w \in$ *dom* (*discovered s*)
using *tree-edges-discovered*
by (*auto simp add*: *Field-def*)

lemma *back-edge-imp-discovered*:
$(v,w) \in$ *back-edges s* $\Longrightarrow$ $v \in$ *dom* (*discovered s*)
$(v,w) \in$ *back-edges s* $\Longrightarrow$ $w \in$ *dom* (*discovered s*)
using *back-edges-discovered*
by (*auto simp add*: *Field-def*)

lemma *cross-edge-imp-discovered*:
$(v,w) \in$ *cross-edges s* $\Longrightarrow$ $v \in$ *dom* (*discovered s*)
$(v,w) \in$ *cross-edges s* $\Longrightarrow$ $w \in$ *dom* (*discovered s*)
using *cross-edges-discovered*
by (*auto simp add*: *Field-def*)

lemma *edge-imp-discovered*:
$(v,w) \in$ *edges s* $\Longrightarrow$ $v \in$ *dom* (*discovered s*)
$(v,w) \in$ *edges s* $\Longrightarrow$ $w \in$ *dom* (*discovered s*)
using *tree-edge-imp-discovered cross-edge-imp-discovered back-edge-imp-discovered*

by *blast+*

lemma *tree-edges-finite*[*simp*, *intro*!]: *finite* (*tree-edges s*)
using *finite-subset*[*OF tree-edges-discovered discovered-finite*] **by** *simp*

lemma *cross-edges-finite*[*simp*, *intro*!]: *finite* (*cross-edges s*)
using *finite-subset*[*OF cross-edges-discovered discovered-finite*] **by** *simp*

lemma *back-edges-finite*[*simp*, *intro*!]: *finite* (*back-edges s*)
using *finite-subset*[*OF back-edges-discovered discovered-finite*] **by** *simp*

lemma *edges-finite*: *finite* (*edges s*)
by *auto*

end

Properties of the DFS Tree

context *DFS-invar* **begin context begin interpretation** *timing-syntax* **.**
lemma *tree-edge-disc-lt-fin*:
$(v,w) \in$ *tree-edges s* $\Longrightarrow v \in$ *dom* (*finished s*) $\Longrightarrow \delta\ s\ w < \varphi\ s\ v$
by (*metis finished-succ-fin tree-edge-is-succ*)

lemma *back-edge-disc-lt-fin*:
$(v,w) \in$ *back-edges s* $\Longrightarrow v \in$ *dom* (*finished s*) $\Longrightarrow \delta\ s\ w < \varphi\ s\ v$
by (*metis finished-succ-fin back-edge-is-succ*)

lemma *cross-edge-disc-lt-fin*:
$(v,w) \in$ *cross-edges s* $\Longrightarrow v \in$ *dom* (*finished s*) $\Longrightarrow \delta\ s\ w < \varphi\ s\ v$
by (*metis finished-succ-fin cross-edge-is-succ*)
end end

context *param-DFS* **begin**

lemma *i-stack-is-tree-path*:
is-invar ($\lambda s.$ *stack s* $\neq$ [] $\longrightarrow$ ($\exists$ *v0* $\in$ *V0*.
path (*tree-edges s*) *v0* (*rev* (*tl* (*stack s*))) (*hd* (*stack s*))))
proof (*induct rule*: *is-invarI*)
case (*discover s s' u v*)
hence *EQ*[*simp*]: *stack s'* = *v* # *stack s*
tree-edges s' = *insert* (*hd* (*stack s*), *v*) (*tree-edges s*)
by *simp-all*
from *discover* **have** *NE*[*simp*]: *stack s* $\neq$ [] **by** *simp*

from *discover* **obtain** *v0* **where**
v0 $\in$ *V0*
path (*tree-edges s*) *v0* (*rev* (*tl* (*stack s*))) (*hd* (*stack s*))

by *blast*
with *path-mono*[*OF* - *this*(*2*)] *EQ* **have**
path (*tree-edges s′*) *v0* (*rev* (*tl* (*stack s*))) (*hd* (*stack s*))
by *blast*
with ‹*v0* ∈ *V0*› **show** *?case*
by (*cases stack s*) (*auto simp*: *path-simps*)

next
case (*finish s s′*)
hence *EQ*[*simp*]: *stack s′* = *tl* (*stack s*)
tree-edges s′ = *tree-edges s*
by *simp-all*

from *finish* **obtain** *v0* **where**
v0 ∈ *V0*
path (*tree-edges s*) *v0* (*rev* (*tl* (*stack s*))) (*hd* (*stack s*))
by *blast*
hence *P*: *path* (*tree-edges s′*) *v0* (*rev* (*stack s′*)) (*hd* (*stack s*)) **by** *simp*

show *?case*
proof
assume *A*: *stack s′* ≠ []
with *P* **have** (*hd* (*stack s′*), *hd* (*stack s*)) ∈ *tree-edges s′*
by (*auto simp*: *neq-Nil-conv path-simps*)
moreover from *P A* **have**
path (*tree-edges s′*) *v0* (*rev* (*tl* (*stack s′*)) @ [*hd* (*stack s′*)]) (*hd* (*stack s*))
by (*simp*)
moreover note ‹*v0* ∈ *V0*›
ultimately show ∃ *v0*∈*V0*. *path* (*tree-edges s′*) *v0* (*rev* (*tl* (*stack s′*))) (*hd* (*stack s′*))
by (*auto simp add*: *path-append-conv*)
qed
qed *simp-all*
end

context *DFS-invar* **begin**

lemmas *stack-is-tree-path* =
i-stack-is-tree-path[*THEN make-invar-thm*, *rule-format*]

lemma *stack-is-path*:
stack s ≠ [] ⟹ ∃ *v0*∈*V0*. *path E v0* (*rev* (*tl* (*stack s*))) (*hd* (*stack s*))
using *stack-is-tree-path path-mono*[*OF tree-edges-ssE*]
by *blast*

lemma *hd-succ-stack-is-path*:
assumes *ne*: *stack s* ≠ []
and *succ*: *v* ∈ *succ* (*hd* (*stack s*))
shows ∃ *v0*∈*V0*. *path E v0* (*rev* (*stack s*)) *v*

proof −
  from *stack-is-path*[*OF ne*] *succ* **obtain** *v0* **where**
    v0 ∈ *V0*
    path E v0 (*rev* (*tl* (*stack s*)) @ [*hd* (*stack s*)]) *v*
    by (*auto simp add*: *path-append-conv*)
  thus *?thesis* **using** *ne*
    by (*cases stack s*) *auto*
qed

lemma *tl-stack-hd-tree-path*:
  assumes *stack s* ≠ []
  and *v* ∈ *set* (*tl* (*stack s*))
  shows (*v*, *hd* (*stack s*)) ∈ (*tree-edges s*)$^+$
proof −
  from *stack-is-tree-path assms* **obtain** *v0* **where**
    path (*tree-edges s*) *v0* (*rev* (*tl* (*stack s*))) (*hd* (*stack s*))
    by *auto*
  from *assms path-member-reach-end*[*OF this*] **show** *?thesis* **by** *simp*
qed
end

context *param-DFS* **begin**
definition *tree-discovered-inv s* ≡
(*tree-edges s* = {} ⟶ *dom* (*discovered s*) ⊆ *V0* ∧ (*stack s* = [] ∨ (∃ *v0*∈*V0*. *stack s* = [*v0*])))
∧ (*tree-edges s* ≠ {} ⟶ (*tree-edges s*)$^+$ ‘‘ *V0* ∪ *V0* = *dom* (*discovered s*) ∪ *V0*)

lemma *i-tree-discovered-inv*:
  is-invar tree-discovered-inv
proof (*induct rule*: *is-invarI*)
  case (*discover s s′ u v*)
  hence *EQ*[*simp*]: *stack s′* = *v* # *stack s*
    tree-edges s′ = *insert* (*hd* (*stack s*), *v*) (*tree-edges s*)
    discovered s′ = (*discovered s*)(*v* ↦ *counter s*)
    by *simp-all*

  from *discover* **interpret** *DFS-invar* **where** *s*=*s* **by** *simp*

  from *discover* **have** *NE*[*simp*]: *stack s* ≠ [] **by** *simp*
  note *TDI* = ‹*tree-discovered-inv s*›[*unfolded tree-discovered-inv-def*]

  have *tree-edges s′* = {} ⟶ *dom* (*discovered s′*) ⊆ *V0* ∧ (*stack s′* = [] ∨ (∃ *v0*∈*V0*. *stack s′* = [*v0*]))
    by *simp* — *tree-edges s′* ≠ {}

  moreover {
    fix *x*

assume A: $x \in (\mathit{tree\text{-}edges}\ s')^+$ `` $V0 \cup V0$ $x \notin V0$
then obtain y **where** y: $(y,x) \in (\mathit{tree\text{-}edges}\ s')^+$ $y \in V0$ **by** *auto*

have $x \in \mathit{dom}\ (\mathit{discovered}\ s') \cup V0$
proof (*cases tree-edges* $s = \{\}$)
case *True* **with** *discover A* **have** $(\mathit{tree\text{-}edges}\ s')^+ = \{(\mathit{hd}\ (\mathit{stack}\ s),\ v)\}$
by (*simp add*: *trancl-single*)
with *A* **show** *?thesis* **by** *auto*
next
case *False* **note** *t-ne* = *this*

show *?thesis*
proof (*cases* $x = v$)
case *True* **thus** *?thesis* **by** *simp*
next
case *False* **with** *y* **have** $(y,x) \in (\mathit{tree\text{-}edges}\ s)^+$
proof (*induct rule*: *trancl-induct*)
case (*step a b*) **hence** $(a,b) \in \mathit{tree\text{-}edges}\ s$ **by** *simp*
with *tree-edge-imp-discovered* **have** $a \in \mathit{dom}\ (\mathit{discovered}\ s)$ **by** *simp*
with *discover* **have** $a \neq v$ **by** *blast*
with *step* **show** *?case* **by** *auto*
qed *simp*
with ‹$y \in V0$› **have** $x \in (\mathit{tree\text{-}edges}\ s)^+$ `` $V0$ **by** *auto*
with *t-ne TDI* **show** *?thesis* **by** *auto*
qed
qed
} **note** *t-d* = *this*

{
fix x
assume $x \in \mathit{dom}\ (\mathit{discovered}\ s') \cup V0$ $x \notin V0$
hence A: $x \in \mathit{dom}\ (\mathit{discovered}\ s')$ **by** *simp*

have $x \in (\mathit{tree\text{-}edges}\ s')^+$ `` $V0 \cup V0$
proof (*cases tree-edges* $s = \{\}$)
case *True* **with** *trancl-single* **have** $(\mathit{tree\text{-}edges}\ s')^+ = \{(\mathit{hd}\ (\mathit{stack}\ s),\ v)\}$ **by** *simp*
moreover from *True TDI* **have** $\mathit{hd}\ (\mathit{stack}\ s) \in V0$ $\mathit{dom}\ (\mathit{discovered}\ s) \subseteq V0$ **by** *auto*
ultimately show *?thesis* **using** *A* ‹$x \notin V0$› **by** *auto*
next
case *False* **note** *t-ne* = *this*

show *?thesis*
proof (*cases* $x=v$)
case *False* **with** *A* **have** $x \in \mathit{dom}\ (\mathit{discovered}\ s)$ **by** *simp*
with *TDI t-ne* ‹$x \notin V0$› **have** $x \in (\mathit{tree\text{-}edges}\ s)^+$ `` $V0$ **by** *auto*
with *trancl-sub-insert-trancl* **show** *?thesis* **by** *simp blast*

next
case *True*
from *t-ne TDI* **have** $dom\ (discovered\ s) \cup V0 = (tree\text{-}edges\ s)^+$ `` $V0 \cup V0$
by *simp*

moreover from *stack-is-tree-path*[*OF NE*] **obtain** *v0* **where** $v0 \in V0$ **and**
$(v0,\ hd\ (stack\ s)) \in (tree\text{-}edges\ s)^*$
by (*blast intro*!: *path-is-rtrancl*)
with *EQ* **have** $(v0,\ hd\ (stack\ s)) \in (tree\text{-}edges\ s')^*$ **by** (*auto intro*: *rtrancl-mono-mp*)
ultimately show *?thesis* **using** ‹$v0 \in V0$› *True* **by** (*auto elim*: *rtrancl-into-trancl1*)
qed
qed
} **with** *t-d* **have** $(tree\text{-}edges\ s')^+$ `` $V0 \cup V0 = dom\ (discovered\ s') \cup V0$ **by** *blast*

ultimately show *?case* **by** (*simp add*: *tree-discovered-inv-def*)
qed (*auto simp add*: *tree-discovered-inv-def*)

lemmas (**in** *DFS-invar*) *tree-discovered-inv* =
i-tree-discovered-inv[*THEN make-invar-thm*]

lemma (**in** *DFS-invar*) *discovered-iff-tree-path*:
$v \notin V0 \Longrightarrow v \in dom\ (discovered\ s) \longleftrightarrow (\exists\, v0 \in V0.\ (v0,v) \in (tree\text{-}edges\ s)^+)$
using *tree-discovered-inv*
by (*auto simp add*: *tree-discovered-inv-def*)

lemma *i-tree-one-predecessor*:
is-invar $(\lambda s.\ \forall (v,v') \in tree\text{-}edges\ s.\ \forall y.\ y \neq v \longrightarrow (y,v') \notin tree\text{-}edges\ s)$
proof (*induct rule*: *is-invarI*)
case (*discover s s' u v*)
hence *EQ*[*simp*]: $tree\text{-}edges\ s' = insert\ (hd\ (stack\ s),v)\ (tree\text{-}edges\ s)$ **by** *simp*

from *discover* **interpret** *DFS-invar* **where** *s*=*s* **by** *simp*
from *discover* **have** *NE*[*simp*]: $stack\ s \neq []$ **by** (*simp add*: *cond-alt*)

{
fix $w\ w'\ y$
assume $*$: $(w,w') \in tree\text{-}edges\ s'$
and $y \neq w$

from *discover stack-discovered* **have** *v-hd*: $hd\ (stack\ s) \neq v$
using *hd-in-set*[*OF NE*] **by** *blast*
from *discover tree-edges-discovered* **have**
v-notin-tree: $\forall (x,x') \in tree\text{-}edges\ s.\ x \neq v \wedge x' \neq v$
by (*blast intro*!: *Field-not-elem*)

have $(y,w') \notin$ *tree-edges* s'
proof (*cases* $w = hd$ (*stack* s))
case *True*
have $(y,v) \notin$ *tree-edges* s'
proof (*rule notI*)
assume $(y,v) \in$ *tree-edges* s'
with *True* ‹$y \neq w$› **have** $(y,v) \in$ *tree-edges* s **by** *simp*
with *v-notin-tree* **show** *False* **by** *auto*
qed
with *True* $*$ ‹$y \neq w$› *v-hd* **show** *?thesis*
apply (*cases* $w = v$)
apply *simp*
using *discover* **apply** *simp* **apply** *blast*
done
next
case *False* **with** *v-notin-tree* $*$ ‹$y \neq w$› *v-hd*
show *?thesis*
apply (*cases* $w' = v$)
apply *simp* **apply** *blast*
using *discover* **apply** *simp* **apply** *blast*
done
qed
}
thus *?case* **by** *blast*
qed *simp-all*

lemma (**in** *DFS-invar*) *tree-one-predecessor*:
assumes $(v,w) \in$ *tree-edges* s
and $a \neq v$
shows $(a,w) \notin$ *tree-edges* s
using *assms make-invar-thm*[*OF i-tree-one-predecessor*]
by *blast*

lemma (**in** *DFS-invar*) *tree-eq-rule*:
$[\![(v,w) \in$ *tree-edges* s; $(u,w) \in$ *tree-edges* $s]\!] \Longrightarrow v=u$
using *tree-one-predecessor*
by *blast*

context begin interpretation *timing-syntax* **.**

lemma *i-tree-edge-disc*:
is-invar ($\lambda s.\ \forall (v,v') \in$ *tree-edges* $s.\ \delta\ s\ v < \delta\ s\ v'$)
proof (*induct rule*: *is-invarI*)
case (*discover* s s' u v)
hence *EQ*[*simp*]: *tree-edges* s' = *insert* (*hd* (*stack* s), v) (*tree-edges* s)
discovered s' = (*discovered* s)($v \mapsto$ *counter* s)
by *simp-all*

from *discover* **interpret** *DFS-invar* **where** $s=s$ **by** *simp*

from *discover* **have** *NE*[*simp*]: *stack s* $\neq$ [] **by** (*simp add*: *cond-alt*)

from *discover tree-edges-discovered* **have**
v-notin-tree: $\forall$ (*x*,*x*′) $\in$ *tree-edges s*. *x* $\neq$ *v* $\wedge$ *x*′ $\neq$ *v*
by (*blast intro*!: *Field-not-elem*)
from *discover stack-discovered* **have**
v-hd: *hd* (*stack s*) $\neq$ *v*
using *hd-in-set*[*OF NE*]
by *blast*

{
fix *a b*
assume *T*: (*a*,*b*) $\in$ *tree-edges s*′
have δ *s*′ *a* < δ *s*′ *b*
proof (*cases b* = *v*)
case *True* **with** *T v-notin-tree* **have** [*simp*]: *a* = *hd* (*stack s*) **by** *auto*
with *stack-discovered* **have** *a* $\in$ *dom* (*discovered s*)
by (*metis hd-in-set NE subsetD*)
with *v-hd True timing-less-counter* **show** *?thesis* **by** *simp*
next
case *False* **with** *v-notin-tree T* **have** (*a*,*b*) $\in$ *tree-edges s a* $\neq$ *v* **by** *auto*
with *discover* **have** δ *s a* < δ *s b* **by** *auto*
with *False* ‹*a*≠*v*› **show** *?thesis* **by** *simp*
qed
} **thus** *?case* **by** *blast*
next
case (*new-root s s*′ *v0*) **then interpret** *DFS-invar* **where** *s*=*s* **by** *simp*
from *new-root* **have** *tree-edges s*′ = *tree-edges s* **by** *simp*
moreover from *tree-edge-imp-discovered new-root* **have** $\forall$ (*v*,*v*′) $\in$ *tree-edges s*. *v* $\neq$ *v0* $\wedge$ *v*′ $\neq$ *v0* **by** *blast*
ultimately show *?case* **using** *new-root* **by** *auto*
qed *simp-all*
end end

context *DFS-invar* **begin context begin interpretation** *timing-syntax* **.**

lemma *tree-edge-disc*:
(*v*,*w*) $\in$ *tree-edges s* $\Longrightarrow$ δ *s v* < δ *s w*
using *i-tree-edge-disc*[*THEN make-invar-thm*]
by *blast*

lemma *tree-path-disc*:
(*v*,*w*) $\in$ (*tree-edges s*)$^+$ $\Longrightarrow$ δ *s v* < δ *s w*
by (*auto elim*!: *trancl-induct dest*: *tree-edge-disc*)

lemma *no-loop-in-tree*:
(*v*,*v*) $\notin$ (*tree-edges s*)$^+$
using *tree-path-disc* **by** *auto*

lemma *tree-acyclic*:
  acyclic (*tree-edges s*)
  by (*metis acyclicI no-loop-in-tree*)

lemma *no-self-loop-in-tree*:
  (v,v) $\notin$ *tree-edges s*
  using *tree-edge-disc* **by** *auto*

lemma *tree-edge-unequal*:
  (v,w) $\in$ *tree-edges s* $\Longrightarrow$ $v \neq w$
  by (*metis no-self-loop-in-tree*)

lemma *tree-path-unequal*:
  (v,w) $\in$ (*tree-edges s*)$^+$ $\Longrightarrow$ $v \neq w$
  by (*metis no-loop-in-tree*)

lemma *tree-subpath'*:
  assumes *x*: (x,v) $\in$ (*tree-edges s*)$^+$
  and *y*: (y,v) $\in$ (*tree-edges s*)$^+$
  and $x \neq y$
  shows (x,y) $\in$ (*tree-edges s*)$^+$ $\vee$ (y,x) $\in$ (*tree-edges s*)$^+$
proof −
  from *x* **obtain** *px* **where** *px*: *path* (*tree-edges s*) *x px v* **and** *px* $\neq$ []
    using *trancl-is-path* **by** *metis*
  from *y* **obtain** *py* **where** *py*: *path* (*tree-edges s*) *y py v* **and** *py* $\neq$ []
    using *trancl-is-path* **by** *metis*

  from ‹*px* $\neq$ []› ‹*py* $\neq$ []› *px py*
  show *?thesis*
  proof (*induction arbitrary*: *v rule*: *rev-nonempty-induct2'*)
    case (*single*) **hence** (x,v) $\in$ *tree-edges s* (y,v) $\in$ *tree-edges s*
      by (*simp-all add*: *path-simps*)
    with *tree-eq-rule* **have** $x=y$ **by** *simp*
    with ‹$x\neq y$› **show** *?case* **by** *contradiction*
  next
    case (*snocl a as*) **hence** (y,v) $\in$ *tree-edges s* **by** (*simp add*: *path-simps*)
    moreover from *snocl* **have** *path* (*tree-edges s*) *x as a* (a,v) $\in$ *tree-edges s*
      by (*simp-all add*: *path-simps*)
    ultimately have *path* (*tree-edges s*) *x as y*
      using *tree-eq-rule*
      by *auto*
    with *path-is-trancl* ‹*as* $\neq$ []› **show** *?case* **by** *metis*
  next
    case (*snocr - a as*) **hence** (x,v) $\in$ *tree-edges s* **by** (*simp add*: *path-simps*)
    moreover from *snocr* **have** *path* (*tree-edges s*) *y as a* (a,v) $\in$ *tree-edges s*
      by (*simp-all add*: *path-simps*)
    ultimately have *path* (*tree-edges s*) *y as x*
      using *tree-eq-rule*
      by *auto*

with *path-is-trancl* ‹*as* ≠ []› **show** *?case* **by** *metis*
next
case (*snoclr a as b bs*) **hence**
path (*tree-edges s*) *x as a* (*a*,*v*) ∈ *tree-edges s*
path (*tree-edges s*) *y bs b* (*b*,*v*) ∈ *tree-edges s*
by (*simp-all add*: *path-simps*)
moreover hence *a*=*b* **using** *tree-eq-rule* **by** *simp*
ultimately show *?thesis* **using** *snoclr.IH* **by** *metis*
qed
qed

lemma *tree-subpath*:
assumes $(x,v) \in (\mathit{tree\text{-}edges}\ s)^+$
and $(y,v) \in (\mathit{tree\text{-}edges}\ s)^+$
and δ: $\delta\ s\ x < \delta\ s\ y$
shows $(x,y) \in (\mathit{tree\text{-}edges}\ s)^+$
proof −
from δ **have** $x \neq y$ **by** *auto*
with *assms tree-subpath′* **have** $(x,y) \in (\mathit{tree\text{-}edges}\ s)^+ \vee (y,x) \in (\mathit{tree\text{-}edges}\ s)^+$

by *simp*
moreover from δ *tree-path-disc* **have** $(y,x) \notin (\mathit{tree\text{-}edges}\ s)^+$ **by** *force*
ultimately show *?thesis* **by** *simp*
qed

lemma *on-stack-is-tree-path*:
assumes *x*: *x* ∈ *set* (*stack s*)
and *y*: *y* ∈ *set* (*stack s*)
and δ: $\delta\ s\ x < \delta\ s\ y$
shows $(x,y) \in (\mathit{tree\text{-}edges}\ s)^+$
proof −
from *x* **obtain** *i* **where** *i*: *stack s* ! *i* = *x i* < *length* (*stack s*)
by (*metis in-set-conv-nth*)

from *y* **obtain** *j* **where** *j*: *stack s* ! *j* = *y j* < *length* (*stack s*)
by (*metis in-set-conv-nth*)

with *i* δ *stack-nth-order* **have** *j* < *i* **by** *force*

from *x* **have** *ne*[*simp*]: *stack s* ≠ [] **by** *auto*

from ‹*j*<*i*› **have** *x* ∈ *set* (*tl* (*stack s*))
using *nth-mem nth-tl*[*OF ne*, *of i* − *1*] *i*
by *auto*
with *tl-stack-hd-tree-path* **have**
x-path: $(x,\ hd\ (stack\ s)) \in (\mathit{tree\text{-}edges}\ s)^+$
by *simp*

then show *?thesis*

proof (*cases j=0*)
case *True* **with** *j* **have** *hd* (*stack s*) = *y* **by** (*metis hd-conv-nth ne*)
with *x-path* **show** *?thesis* **by** *simp*
next
case *False* **hence** $y \in$ *set* (*tl* (*stack s*))
using *nth-mem nth-tl*[*OF ne*, *of j* − *1*] *j*
by *auto*
with *tl-stack-hd-tree-path* **have** (*y*, *hd* (*stack s*)) $\in$ (*tree-edges s*)$^+$
by *simp*
with *x-path* δ **show** *?thesis*
using *tree-subpath*
by *metis*
qed
qed

lemma *hd-stack-tree-path-finished*:
assumes *stack s* $\neq$ []
assumes (*hd* (*stack s*), *v*) $\in$ (*tree-edges s*)$^+$
shows $v \in$ *dom* (*finished s*)
proof (*cases* $v \in$ *set* (*stack s*))
case *True*
from *assms no-loop-in-tree* **have** *hd* (*stack s*) $\neq$ *v* **by** *auto*
with *True* **have** $v \in$ *set* (*tl* (*stack s*)) **by** (*cases stack s*) *auto*
with *tl-stack-hd-tree-path assms* **have** (*hd* (*stack s*), *hd* (*stack s*)) $\in$ (*tree-edges s*)$^+$ **by** (*metis trancl-trans*)
with *no-loop-in-tree* **show** *?thesis* **by** *contradiction*
next
case *False*
from *assms* **obtain** *x* **where** (*x*,*v*) $\in$ *tree-edges s* **by** (*metis tranclE*)
with *tree-edge-imp-discovered* **have** $v \in$ *dom* (*discovered s*) **by** *blast*
with *False* **show** *?thesis* **by** (*simp add*: *stack-set-def*)
qed

lemma *tree-edge-impl-parenthesis*:
assumes *t*: (*v*,*w*) $\in$ *tree-edges s*
and *f*: $v \in$ *dom* (*finished s*)
shows $w \in$ *dom* (*finished s*)
$\wedge\ \delta\ s\ v < \delta\ s\ w$
$\wedge\ \varphi\ s\ w < \varphi\ s\ v$
proof −
from *tree-edge-disc-lt-fin assms* **have** $\delta\ s\ w < \varphi\ s\ v$ **by** *simp*
with *f tree-edge-imp-discovered*[*OF t*] *tree-edge-disc*[*OF t*]
show *?thesis*
using *parenthesis-contained*
by *metis*
qed

lemma *tree-path-impl-parenthesis*:
assumes (*v*,*w*) $\in$ (*tree-edges s*)$^+$

and $v \in$ *dom* (*finished* s)
shows $w \in$ *dom* (*finished* s)
$\wedge\ \delta\ s\ v < \delta\ s\ w$
$\wedge\ \varphi\ s\ w < \varphi\ s\ v$
using *assms*
by (*auto elim*!: *trancl-induct dest*: *tree-edge-impl-parenthesis*)

lemma *nc-reachable-v0-parenthesis*:
assumes *C*: $\neg$ *cond s* $\neg$ *is-break param s*
and *v*: $v \in$ *reachable* $v \notin V0$
obtains *v0* **where** $v0 \in V0$
and $\delta\ s\ v0 < \delta\ s\ v \wedge \varphi\ s\ v < \varphi\ s\ v0$
proof $-$
from *nc-discovered-eq-reachable*[*OF C*] *discovered-iff-tree-path v*
obtain *v0* **where** $v0 \in V0$ **and**
$(v0,v) \in (\textit{tree-edges}\ s)^+$
by *auto*
moreover with *nc-V0-finished*[*OF C*] **have** $v0 \in$ *dom* (*finished* s)
by *auto*
ultimately show *?thesis*
using *tree-path-impl-parenthesis that*[*OF* ‹$v0 \in V0$›]
by *simp*
qed

end end

context *param-DFS* **begin context begin interpretation** *timing-syntax* **.**

definition *paren-imp-tree-reach* **where**
paren-imp-tree-reach s $\equiv \forall\, v \in$ *dom* (*discovered s*). $\forall\, w \in$ *dom* (*finished s*).
$\delta\ s\ v < \delta\ s\ w \wedge (v \notin \textit{dom}\ (\textit{finished}\ s) \vee \varphi\ s\ v > \varphi\ s\ w)$
$\longrightarrow (v,w) \in (\textit{tree-edges}\ s)^+$

lemma *paren-imp-tree-reach*:
is-invar paren-imp-tree-reach
unfolding *paren-imp-tree-reach-def*[*abs-def*]
proof (*induct rule*: *is-invarI*)
case (*discover s s' u v*)
hence *EQ*[*simp*]: *tree-edges s'* = *insert* (*hd* (*stack s*), *v*) (*tree-edges s*)
finished s' = *finished s*
discovered s' = (*discovered s*)($v \mapsto$ *counter s*)
by *simp-all*

from *discover* **interpret** *DFS-invar* **where** *s=s* **by** *simp*
from *discover* **have** *NE*[*simp*]: *stack s* $\neq$ [] **by** (*simp add*: *cond-alt*)

show *?case*
proof (*intro ballI impI*)
fix *a b*

```
      assume F:a ∈ dom (discovered s') b ∈ dom (finished s')
        and D: δ s' a < δ s' b ∧ (a ∉ dom (finished s') ∨ φ s' a > φ s' b)

      from F finished-discovered discover have b ≠ v by auto
      show (a,b) ∈ (tree-edges s')⁺
      proof (cases a = v)
        case True with D ‹b≠v› have counter s < δ s b by simp
        also from F have b ∈ dom (discovered s)
          using finished-discovered by auto
        with timing-less-counter have δ s b < counter s by simp
        finally have False .
        thus ?thesis ..
      next
        case False with ‹b≠v› F D discover have (a,b) ∈ (tree-edges s)⁺ by simp
        thus ?thesis by (auto intro: trancl-mono-mp)
      qed
    qed
  next
    case (finish s s' u)
    hence EQ[simp]: tree-edges s' = tree-edges s
                    finished s' = (finished s)(hd (stack s) ↦ counter s)
                    discovered s' = discovered s
                    stack s' = tl (stack s)
      by simp-all

    from finish interpret DFS-invar where s=s by simp
    from finish have NE[simp]: stack s ≠ [] by (simp add: cond-alt)

    show ?case
    proof (intro ballI impI)
      fix a b
      assume F: a ∈ dom (discovered s') b ∈ dom (finished s')
        and paren: δ s' a < δ s' b ∧ (a ∉ dom (finished s') ∨ φ s' a > φ s' b)
      hence a ≠ b by auto

      show (a,b) ∈ (tree-edges s')⁺
      proof (cases b = hd (stack s))
        case True hence φb: φ s' b = counter s by simp
        have a ∈ set (stack s)
          unfolding stack-set-def
        proof
          from F show a ∈ dom (discovered s) by simp
         from True ‹a≠b› φb paren have a ∈ dom (finished s) ⟶ φ s a > counter
s by simp
          with timing-less-counter show a ∉ dom (finished s) by force
        qed
        with paren True on-stack-is-tree-path have (a,b) ∈ (tree-edges s)⁺ by auto
        thus ?thesis by (auto intro: trancl-mono-mp)
      next
```

case *False* **note** *b-not-hd* = *this*
show *?thesis*
proof (*cases a* = *hd* (*stack s*))
case *False* **with** *b-not-hd F paren finish* **show** *?thesis* **by** *simp*
next
case *True* **with** *paren b-not-hd F* **have**
$a \in dom\ (discovered\ s)$ $b \in dom\ (finished\ s)$ $\delta\ s\ a < \delta\ s\ b$
by *simp-all*
moreover from *True stack-not-finished* **have** $a \notin dom\ (finished\ s)$
by *simp*
ultimately show *?thesis* **by** (*simp add*: *finish*)
qed
qed
qed
next
case (*new-root s s′ v0*) **then interpret** *DFS-invar* **where** *s*=*s* **by** *simp*
from *new-root finished-discovered* **have** $v0 \notin dom\ (finished\ s)$ **by** *auto*
moreover note *timing-less-counter finished-discovered*
ultimately show *?case* **using** *new-root* **by** *clarsimp force*
qed *simp-all*
end end

context *DFS-invar* **begin context begin interpretation** *timing-syntax* **.**

lemmas *s-paren-imp-tree-reach* =
paren-imp-tree-reach[*THEN make-invar-thm*]

lemma *parenthesis-impl-tree-path-not-finished*:
assumes $v \in dom\ (discovered\ s)$
and $w \in dom\ (finished\ s)$
and $\delta\ s\ v < \delta\ s\ w$
and $v \notin dom\ (finished\ s)$
shows $(v,w) \in (tree\text{-}edges\ s)^+$
using *s-paren-imp-tree-reach assms*
by (*auto simp add*: *paren-imp-tree-reach-def*)

lemma *parenthesis-impl-tree-path*:
assumes $v \in dom\ (finished\ s)$ $w \in dom\ (finished\ s)$
and $\delta\ s\ v < \delta\ s\ w$ $\varphi\ s\ v > \varphi\ s\ w$
shows $(v,w) \in (tree\text{-}edges\ s)^+$
proof −
from *assms*(*1*) **have** $v \in dom\ (discovered\ s)$
using *finished-discovered* **by** *blast*
with *assms* **show** *?thesis*
using *s-paren-imp-tree-reach assms*
by (*auto simp add*: *paren-imp-tree-reach-def*)
qed

lemma *tree-path-iff-parenthesis*:

assumes $v \in dom\ (finished\ s)$ $w \in dom\ (finished\ s)$
shows $(v,w) \in (tree\text{-}edges\ s)^+ \longleftrightarrow \delta\ s\ v < \delta\ s\ w \wedge \varphi\ s\ v > \varphi\ s\ w$
using *assms*
by (*metis parenthesis-impl-tree-path tree-path-impl-parenthesis*)

lemma *no-pending-succ-impl-path-in-tree*:
assumes v: $v \in dom\ (discovered\ s)$ $pending\ s$ `` $\{v\} = \{\}$
and w: $w \in succ\ v$
and δ: $\delta\ s\ v < \delta\ s\ w$
shows $(v,w) \in (tree\text{-}edges\ s)^+$
proof (*cases* $v \in dom\ (finished\ s)$)
case *True*
with *assms assms* **have** $\delta\ s\ w < \varphi\ s\ v$ $w \in dom\ (discovered\ s)$
using *finished-succ-fin finished-imp-succ-discovered*
by *simp-all*
with *True* δ **show** *?thesis*
using *parenthesis-contained parenthesis-impl-tree-path*
by *blast*
next
case *False*
show *?thesis*
proof (*cases* $w \in dom\ (finished\ s)$)
case *True* **with** *False* v δ **show** *?thesis* **by** (*simp add*: *parenthesis-impl-tree-path-not-finished*)
next
case *False* **with** ‹$v \notin dom\ (finished\ s)$› *no-pending-imp-succ-discovered v w*
have
$v \in set\ (stack\ s)$ $w \in set\ (stack\ s)$
by (*simp-all add*: *stack-set-def*)
with *on-stack-is-tree-path* δ **show** *?thesis* **by** *simp*
qed
qed

lemma *finished-succ-impl-path-in-tree*:
assumes f: $v \in dom\ (finished\ s)$
and s: $w \in succ\ v$
and δ: $\delta\ s\ v < \delta\ s\ w$
shows $(v,w) \in (tree\text{-}edges\ s)^+$
using *no-pending-succ-impl-path-in-tree finished-no-pending finished-discovered*
using *assms*
by *blast*
end end

Properties of Cross Edges

context *param-DFS* **begin context begin interpretation** *timing-syntax* **.**

lemma *i-cross-edges-finished*: *is-invar* $(\lambda s.\ \forall (u,v) \in cross\text{-}edges\ s.$
$v \in dom\ (finished\ s) \wedge (u \in dom\ (finished\ s) \longrightarrow \varphi\ s\ v < \varphi\ s\ u))$
proof (*induction rule*: *is-invarI-full*)

case (*finish s s′ u e*)
interpret *DFS-invar G param s* **by** *fact*
from *finish stack-not-finished* **have** $u \notin$ *dom* (*finished s*) **by** *auto*
with *finish* **show** *?case* **by** (*auto intro*: *timing-less-counter*)
next
case (*cross-edge s s′ u v e*)
interpret *DFS-invar G param s* **by** *fact*
from *cross-edge stack-not-finished* **have** $u \notin$ *dom* (*finished s*) **by** *auto*
with *cross-edge* **show** *?case* **by** (*auto intro*: *timing-less-counter*)
qed *simp-all*

end end

context *DFS-invar* **begin context begin interpretation** *timing-syntax* **.**
lemmas *cross-edges-finished*
= *i-cross-edges-finished*[*THEN make-invar-thm*]

lemma *cross-edges-target-finished*:
(*u*,*v*)∈*cross-edges s* ⟹ *v* ∈ *dom* (*finished s*)
using *cross-edges-finished* **by** *auto*

lemma *cross-edges-finished-decr*:
⟦(*u*,*v*)∈*cross-edges s*; *u*∈*dom* (*finished s*)⟧ ⟹ φ *s v* < φ *s u*
using *cross-edges-finished* **by** *auto*

lemma *cross-edge-unequal*:
assumes *cross*: (*v*,*w*) ∈ *cross-edges s*
shows $v \neq w$
proof −
from *cross-edges-target-finished*[*OF cross*] **have**
w-fin: *w* ∈ *dom* (*finished s*) **.**

show *?thesis*
proof (*cases v* ∈ *dom* (*finished s*))
case *True* **with** *cross-edges-finished-decr*[*OF cross*]
show *?thesis* **by** *force*
next
case *False* **with** *w-fin* **show** *?thesis* **by** *force*
qed
qed
end end

Properties of Back Edges

context *param-DFS* **begin context begin interpretation** *timing-syntax* **.**

lemma *i-back-edge-impl-tree-path*:
is-invar (λ*s*. ∀ (*v*,*w*) ∈ *back-edges s*. (*w*,*v*) ∈ (*tree-edges s*)$^+$ ∨ *w* = *v*)
proof (*induct rule*: *is-invarI-full*)

case (*back-edge s s' u v*) **then interpret** *DFS-invar* **where** *s=s* **by** *simp*

from *back-edge* **have** *st*: $v \in$ *set* (*stack s*) $u \in$ *set* (*stack s*)
  using *stack-set-def*
  by *auto*

have $(v,u) \in$ (*tree-edges s*)$^+$ $\vee$ $u = v$
proof (*rule disjCI*)
  assume $u \neq v$
  with *st back-edge* **have** $v \in$ *set* (*tl* (*stack s*)) **by** (*metis not-hd-in-tl*)
  with *tl-lt-stack-hd-discover st back-edge* **have** δ *s v* $<$ δ *s u* **by** *simp*
  with *on-stack-is-tree-path st* **show** $(v,u) \in$ (*tree-edges s*)$^+$ **by** *simp*
qed
with *back-edge* **show** *?case* **by** *auto*
next
case *discover* **thus** *?case* **using** *trancl-sub-insert-trancl* **by** *force*
qed *simp-all*

end end

context *DFS-invar* **begin context begin interpretation** *timing-syntax* **.**

lemma *back-edge-impl-tree-path*:
  $[\![(v,w) \in$ *back-edges s*; $v \neq w]\!] \Longrightarrow (w,v) \in$ (*tree-edges s*)$^+$
  using *i-back-edge-impl-tree-path*[*THEN make-invar-thm*]
  by *blast*

lemma *back-edge-disc*:
  assumes $(v,w) \in$ *back-edges s*
  shows δ *s w* $\leq$ δ *s v*
proof *cases*
  assume $v \neq w$
  with *assms back-edge-impl-tree-path* **have** $(w,v) \in$ (*tree-edges s*)$^+$ **by** *simp*
  with *tree-path-disc* **show** *?thesis* **by** *force*
qed *simp*

lemma *back-edges-tree-disjoint*:
  back-edges s $\cap$ *tree-edges s* $= \{\}$
  using *back-edge-disc tree-edge-disc*
  by *force*

lemma *back-edges-tree-pathes-disjoint*:
  back-edges s $\cap$ (*tree-edges s*)$^+$ $= \{\}$
  using *back-edge-disc tree-path-disc*
  by *force*

lemma *back-edge-finished*:
  assumes $(v,w) \in$ *back-edges s*
  and $w \in$ *dom* (*finished s*)

```
    shows v ∈ dom (finished s) ∧ φ s v ≤ φ s w
  proof (cases v=w)
    case True with assms show ?thesis by simp
  next
     case False with back-edge-impl-tree-path assms have (w,v) ∈ (tree-edges s)⁺
by simp
    with tree-path-impl-parenthesis assms show ?thesis by fastforce
  qed

end end

context param-DFS begin context begin interpretation timing-syntax .

  lemma i-disc-imp-back-edge-or-pending:
    is-invar (λs. ∀(v,w) ∈ E.
        v ∈ dom (discovered s) ∧ w ∈ dom (discovered s)
      ∧ δ s v ≥ δ s w
      ∧ (w ∈ dom (finished s) ⟶ v ∈ dom (finished s) ∧ φ s w ≥ φ s v)
      ⟶ (v,w) ∈ back-edges s ∨ (v,w) ∈ pending s)
  proof (induct rule: is-invarI-full)
    case (cross-edge s s' u v) then interpret DFS-invar where s=s by simp
    from cross-edge stack-not-finished[of u] have u ∉ dom (finished s)
      using hd-in-set
      by (auto simp add: cond-alt)
    with cross-edge show ?case by auto
  next
    case (finish s s' u v) then interpret DFS-invar where s=s by simp

    from finish have
      IH: ⋀v w. ⟦w ∈ succ v; v ∈ dom (discovered s); w ∈ dom (discovered s);
                  δ s w ≤ δ s v;
                  (w ∈ dom (finished s) ⟹ v ∈ dom (finished s) ∧ φ s v ≤ φ s w)⟧
                 ⟹ (v, w) ∈ back-edges s ∨ (v, w) ∈ pending s
      by blast

    from finish have ne[simp]: stack s ≠ []
              and p[simp]: pending s `` {hd (stack s)} = {}
      by (simp-all)

    from hd-in-set[OF ne] have disc: hd (stack s) ∈ dom (discovered s)
                        and not-fin: hd (stack s) ∉ dom (finished s)
      using stack-discovered stack-not-finished
      by blast+

    {
      fix w
      assume w: w ∈ succ (hd (stack s)) w ≠ hd (stack s) w ∈ dom (discovered s)
        and f: w ∈ dom (finished s) ⟶ counter s ≤ φ s w
        and δ: δ s w ≤ δ s (hd (stack s))
```

with *timing-less-counter* **have** $w \notin dom\ (finished\ s)$ **by** *force*
with *finish w δ disc* **have** $(hd\ (stack\ s),\ w) \in back\text{-}edges\ s$ **by** *blast*
}

moreover
{
fix *w*
assume $hd\ (stack\ s) \in succ\ w\ w \neq hd\ (stack\ s)$
and $w \in dom\ (finished\ s)\ \delta\ s\ (hd\ (stack\ s)) \leq \delta\ s\ w$
with *IH[of hd (stack s) w] disc not-fin* **have**
$(w,\ hd\ (stack\ s)) \in back\text{-}edges\ s$
using *finished-discovered finished-no-pending[of w]*
by *blast*
}

ultimately show *?case*
using *finish*
by *clarsimp auto*
next
case *(discover s s′ u v)* **then interpret** *DFS-invar* **where** *s=s* **by** *simp*

from *discover* **show** *?case*
using *timing-less-counter*
by *clarsimp fastforce*
next
case *(new-root s s′ v0)* **then interpret** *DFS-invar* **where** *s=s* **by** *simp*

from *new-root empty-stack-imp-empty-pending* **have** $pending\ s = \{\}$ **by** *simp*
with *new-root* **show** *?case*
using *timing-less-counter*
by *clarsimp fastforce*
qed *auto*
end end

context *DFS-invar* **begin context begin interpretation** *timing-syntax* **.**

lemma *disc-imp-back-edge-or-pending*:
$[\![w \in succ\ v;\ v \in dom\ (discovered\ s);\ w \in dom\ (discovered\ s);\ \delta\ s\ w \leq \delta\ s\ v;$
$(w \in dom\ (finished\ s) \Longrightarrow v \in dom\ (finished\ s) \wedge \varphi\ s\ v \leq \varphi\ s\ w)]\!]$
$\Longrightarrow (v,\ w) \in back\text{-}edges\ s \vee (v,\ w) \in pending\ s$
using *i-disc-imp-back-edge-or-pending[THEN make-invar-thm]*
by *blast*

lemma *finished-imp-back-edge*:
$[\![w \in succ\ v;\ v \in dom\ (finished\ s);\ w \in dom\ (finished\ s);$
$\delta\ s\ w \leq \delta\ s\ v;\ \varphi\ s\ v \leq \varphi\ s\ w]\!]$
$\Longrightarrow (v,\ w) \in back\text{-}edges\ s$
using *disc-imp-back-edge-or-pending finished-discovered finished-no-pending*

by *fast*

lemma *finished-not-finished-imp-back-edge*:
$[\![w \in succ\ v;\ v \in dom\ (finished\ s);\ w \in dom\ (discovered\ s);$
$w \notin dom\ (finished\ s);$
$\delta\ s\ w \leq \delta\ s\ v]\!]$
$\Longrightarrow (v,\ w) \in back\text{-}edges\ s$
using *disc-imp-back-edge-or-pending finished-discovered finished-no-pending*
by *fast*

lemma *finished-self-loop-in-back-edges*:
assumes $v \in dom\ (finished\ s)$
and $(v,v) \in E$
shows $(v,v) \in back\text{-}edges\ s$
using *assms*
using *finished-imp-back-edge*
by *blast*
end end

context *DFS-invar* **begin**

context begin interpretation *timing-syntax* **.**

lemma *tree-cross-acyclic*:
acyclic (*tree-edges s* $\cup$ *cross-edges s*) (**is** *acyclic ?E*)
proof (*rule ccontr*)
{
fix *u v*
assume $*$: $u \in dom\ (finished\ s)$ **and** $(u,v) \in ?E^+$
from *this*(*2*) **have** $\varphi\ s\ v < \varphi\ s\ u \wedge v \in dom\ (finished\ s)$
proof *induct*
case *base* **thus** *?case*
by (*metis Un-iff* $*$ *cross-edges-finished-decr cross-edges-target-finished tree-edge-impl-parenthesis*)
next
case (*step v w*)
hence $\varphi\ s\ w < \varphi\ s\ v \wedge w \in dom\ (finished\ s)$
by (*metis Un-iff cross-edges-finished-decr cross-edges-target-finished tree-edge-impl-parenthesis*)
with *step* **show** *?case* **by** *auto*
qed
} **note** *aux* = *this*

assume $\neg$ *acyclic ?E*
then obtain *u* **where** *path*: $(u,u) \in ?E^+$ **by** (*auto simp add*: *acyclic-def*)
show *False*
proof *cases*

assume $u \in$ *dom* (*finished s*)
with *aux path* **show** *False* **by** *blast*
next
assume $*$: $u \notin$ *dom* (*finished s*)
moreover
from *no-loop-in-tree* **have** $(u,u) \notin$ (*tree-edges s*)$^+$ **.**
with *trancl-union-outside*[*OF path*] **obtain** *x y* **where** $(u,x) \in$ *?E** $(x,y) \in$ *cross-edges s* $(y,u) \in$ *?E** **by** *auto*
with *cross-edges-target-finished* **have** $y \in$ *dom* (*finished s*) **by** *simp*
moreover with $*$ ‹$(y,u) \in$ *?E**› **have** $(y,u) \in$ *?E*$^+$ **by** (*auto simp add*: *rtrancl-eq-or-trancl*)
ultimately show *False* **by** (*metis aux*)
qed
qed
end

lemma *cycle-contains-back-edge*:
assumes *cycle*: $(u,u) \in$ (*edges s*)$^+$
shows $\exists$ *v w*. $(u,v) \in$ (*edges s*)* $\wedge$ $(v,w) \in$ *back-edges s* $\wedge$ $(w,u) \in$ (*edges s*)*
proof −
from *tree-cross-acyclic* **have** $(u,u) \notin$ (*tree-edges s* $\cup$ *cross-edges s*)$^+$ **by** (*simp add*: *acyclic-def*)
with *trancl-union-outside*[*OF cycle*] **show** *?thesis* **.**
qed

lemma *cycle-needs-back-edge*:
assumes *back-edges s* = {}
shows *acyclic* (*edges s*)
proof (*rule ccontr*)
assume ¬ *acyclic* (*edges s*)
then obtain *u* **where** $(u,u) \in$ (*edges s*)$^+$ **by** (*auto simp*: *acyclic-def*)
with *assms* **have** $(u,u) \in$ (*tree-edges s* $\cup$ *cross-edges s*)$^+$ **by** *auto*
with *tree-cross-acyclic* **show** *False* **by** (*simp add*: *acyclic-def*)
qed

lemma *back-edge-closes-cycle*:
assumes *back-edges s* $\neq$ {}
shows ¬ *acyclic* (*edges s*)
proof −
from *assms* **obtain** *v w* **where** *be*: $(v,w) \in$ *back-edges s* **by** *auto*
hence $(w,w) \in$ (*edges s*)$^+$
proof (*cases* $v{=}w$)
case *False*
with *be back-edge-impl-tree-path* **have** $(w,v) \in$ (*tree-edges s*)$^+$ **by** *simp*
hence $(w,v) \in$ (*edges s*)$^+$ **by** (*blast intro*: *trancl-mono-mp*)
also from *be* **have** $(v,w) \in$ *edges s* **by** *simp*
finally show *?thesis* **.**
qed *auto*
thus *?thesis* **by** (*auto simp add*: *acyclic-def*)

qed

lemma *back-edge-closes-reachable-cycle*:
back-edges s $\neq$ {} $\Longrightarrow$ $\neg$ *acyclic* (*E* $\cap$ *reachable* $\times$ *UNIV*)
by (*metis back-edge-closes-cycle edges-ss-reachable-edges cyclic-subset*)

lemma *cycle-iff-back-edges*:
acyclic (*edges s*) $\longleftrightarrow$ *back-edges s* = {}
by (*metis back-edge-closes-cycle cycle-needs-back-edge*)
end

1.2.4 White Path Theorem

context *DFS* **begin**
context begin interpretation *timing-syntax* **.**

definition *white-path* **where**
white-path s x y $\equiv$ *x*$\neq$*y*
$\longrightarrow$ ($\exists$ *p*. *path E x p y* $\wedge$
(δ *s x* < δ *s y* $\wedge$ ($\forall$ *v* $\in$ *set* (*tl p*). δ *s x* < δ *s v*)))

lemma *white-path*:
it-dfs $\leq$ *SPEC*(λ*s*. $\forall$ *x* $\in$ *reachable*. $\forall$ *y* $\in$ *reachable*. $\neg$ *is-break param s* $\longrightarrow$
white-path s x y $\longleftrightarrow$ (*x*,*y*) $\in$ (*tree-edges s*)*)
proof (*rule it-dfs-SPEC*, *intro ballI impI*)
fix *s x y*
assume *DI*: *DFS-invar G param s*
and *C*: $\neg$ *cond s* $\neg$ *is-break param s*
and *reach*: *x* $\in$ *reachable y* $\in$ *reachable*

from *DI* **interpret** *DFS-invar* **where** *s*=*s* **.**

note *fin-eq-reach* = *nc-finished-eq-reachable*[*OF C*]

show *white-path s x y* $\longleftrightarrow$ (*x*,*y*) $\in$ (*tree-edges s*)*
proof (*cases x*=*y*)
case *True* **thus** *?thesis* **by** (*simp add*: *white-path-def*)
next
case *False*

show *?thesis*
proof
assume (*x*,*y*) $\in$ (*tree-edges s*)*
with ‹*x*$\neq$*y*› **have** *T*: (*x*,*y*) $\in$ (*tree-edges s*)$^+$ **by** (*metis rtranclD*)
then obtain *p* **where** *P*: *path* (*tree-edges s*) *x p y* **by** (*metis trancl-is-path*)
with *tree-edges-ssE* **have** *path E x p y* **using** *path-mono*[**where** *E*=*tree-edges s*]
by *simp*
moreover

from P **have** $\delta\ s\ x < \delta\ s\ y \wedge (\forall\ \ v \in set\ (tl\ p).\ \delta\ s\ x < \delta\ s\ v)$
  using ‹$x \neq y$›
  proof (*induct rule*: *path-tl-induct*)
    case (*single u*) **thus** *?case* **by** (*fact tree-edge-disc*)
  next
    case (*step u v*) **note** ‹$\delta\ s\ x < \delta\ s\ u$›
    also from *step* **have** $\delta\ s\ u < \delta\ s\ v$ **by** (*metis tree-edge-disc*)
    finally show *?case* **.**
  qed
  ultimately show *white-path s x y*
    by (*auto simp add*: ‹$x \neq y$› *white-path-def*)
next
  assume *white-path s x y*
  with ‹$x \neq y$› **obtain** p **where**
      P:*path E x p y* **and**
      white: $\delta\ s\ x < \delta\ s\ y \wedge (\forall\ \ v \in set\ (tl\ p).\ \delta\ s\ x < \delta\ s\ v)$
    unfolding *white-path-def*
    by *blast*
  hence $p \neq []$ **by** *auto*
  thus $(x,y) \in (tree\text{-}edges\ s)^*$ **using** *P white reach*(*2*)
  proof (*induction p arbitrary*: *y rule*: *rev-nonempty-induct*)
    case *single* **hence** $y \in succ\ x$ **by** (*simp add*: *path-cons-conv*)
    with *reach single* **show** *?case*
      using *fin-eq-reach finished-succ-impl-path-in-tree*[*of x y*]
      by *simp*
  next
    case (*snoc u us*) **hence** *path E x us u* **by** (*simp add*: *path-append-conv*)
    moreover hence $(x,u) \in E^*$ **by** (*simp add*: *path-is-rtrancl*)
    with *reach* **have** *ureach*: $u \in reachable$
      by (*metis rtrancl-image-advance-rtrancl*)
    moreover from *snoc* **have** $\delta\ s\ x < \delta\ s\ u\ (\forall\ v{\in}set\ (tl\ us).\ \delta\ s\ x < \delta\ s\ v)$
      by *simp-all*
    ultimately have *x-u*: $(x,u) \in (tree\text{-}edges\ s)^*$ **by** (*metis snoc.IH*)

    from *snoc* **have** $y \in succ\ u$ **by** (*simp add*: *path-append-conv*)
    from *snoc*(*5*) *fin-eq-reach finished-discovered* **have**
      y-f-d: $y \in dom\ (finished\ s)\ y \in dom\ (discovered\ s)$
      by *auto*

    from ‹$y \in succ\ u$› *ureach fin-eq-reach* **have** $\delta\ s\ y < \varphi\ s\ u$
      using *finished-succ-fin* **by** *simp*
    also from ‹$\delta\ s\ x < \delta\ s\ u$› **have** $x \neq u$ **by** *auto*
    with *x-u* **have** $(x,u) \in (tree\text{-}edges\ s)^+$ **by** (*metis rtrancl-eq-or-trancl*)
    with *fin-eq-reach reach* **have** $\varphi\ s\ u < \varphi\ s\ x$
      using *tree-path-impl-parenthesis*
      by *simp*
    finally have $\varphi\ s\ y < \varphi\ s\ x$
      using *reach fin-eq-reach y-f-d snoc*
      using *parenthesis-contained*

by *blast*
hence $(x,y) \in (tree\text{-}edges\ s)^+$
using *reach fin-eq-reach y-f-d snoc*
using *parenthesis-impl-tree-path*
by *blast*
thus *?case* **by** *auto*
qed
qed
qed
qed
end end

end

1.3 Invariants for SCCs

theory *DFS-Invars-SCC*
imports
DFS-Invars-Basic
begin

definition *scc-root′* :: $('v \times 'v)$ *set* $\Rightarrow$ $('v,'es)$ *state-scheme* $\Rightarrow 'v \Rightarrow 'v$ *set* $\Rightarrow$ *bool*
— v is a root of its scc iff all the discovered parts of the scc can be reached by tree edges from v
where
scc-root′ E s v scc $\longleftrightarrow$ *is-scc E scc*
$\wedge\ v \in scc$
$\wedge\ v \in dom\ (discovered\ s)$
$\wedge\ scc \cap dom\ (discovered\ s) \subseteq (tree\text{-}edges\ s)^*\ ``\ \{v\}$

context *param-DFS-defs* **begin**
abbreviation *scc-root* $\equiv$ *scc-root′ E*
lemmas *scc-root-def* = *scc-root′-def*

lemma *scc-rootI*:
assumes *is-scc E scc*
and $v \in dom\ (discovered\ s)$
and $v \in scc$
and $scc \cap dom\ (discovered\ s) \subseteq (tree\text{-}edges\ s)^*\ ``\ \{v\}$
shows *scc-root s v scc*
using *assms* **by** (*simp add*: *scc-root-def*)

definition *scc-roots s* = $\{v.\ \exists\, scc.\ scc\text{-}root\ s\ v\ scc\}$
end

context *DFS-invar* **begin**
lemma *scc-root-is-discovered*:
scc-root s v scc $\Longrightarrow v \in dom\ (discovered\ s)$

by (*simp add*: *scc-root-def*)

lemma *scc-root-scc-tree-rtrancl*:
assumes *scc-root s v scc*
and $x \in scc$ $x \in dom$ (*discovered s*)
shows $(v,x) \in (\textit{tree-edges}\ s)^*$
using *assms*
by (*auto simp add*: *scc-root-def*)

lemma *scc-root-scc-reach*:
assumes *scc-root s r scc*
and $v \in scc$
shows $(r,v) \in E^*$
proof −
from *assms* **have** *is-scc E scc* $r \in scc$ **by** (*simp-all add*: *scc-root-def*)
with *is-scc-connected assms* **show** *?thesis* **by** *metis*
qed

lemma *scc-reach-scc-root*:
assumes *scc-root s r scc*
and $v \in scc$
shows $(v,r) \in E^*$
proof −
from *assms* **have** *is-scc E scc* $r \in scc$ **by** (*simp-all add*: *scc-root-def*)
with *is-scc-connected assms* **show** *?thesis* **by** *metis*
qed

lemma *scc-root-scc-tree-trancl*:
assumes *scc-root s v scc*
and $x \in scc$ $x \in dom$ (*discovered s*) $x \neq v$
shows $(v,x) \in (\textit{tree-edges}\ s)^+$
using *assms scc-root-scc-tree-rtrancl*
by (*auto simp add*: *rtrancl-eq-or-trancl*)

lemma *scc-root-unique-scc*:
scc-root s v scc $\Longrightarrow$ *scc-root s v scc'* $\Longrightarrow$ *scc = scc'*
unfolding *scc-root-def*
by (*metis is-scc-unique*)

lemma *scc-root-unique-root*:
assumes *scc1*: *scc-root s v scc*
and *scc2*: *scc-root s v' scc*
shows $v = v'$
proof (*rule ccontr*)
assume $v \neq v'$
from *scc1* **have** $v \in scc$ $v \in dom$ (*discovered s*)
by (*simp-all add*: *scc-root-def*)
with *scc-root-scc-tree-trancl*[*OF scc2*] ‹$v \neq v'$› **have** $(v',v) \in (\textit{tree-edges}\ s)^+$
by *simp*

also from *scc2* **have** $v' \in scc$ $v' \in dom$ (*discovered s*)
  by (*simp-all add*: *scc-root-def*)
with *scc-root-scc-tree-trancl*[*OF scc1*] ‹$v \neq v'$› **have** $(v,v') \in (tree\text{-}edges\ s)^+$
  by *simp*
finally show *False* **using** *no-loop-in-tree* **by** *contradiction*
qed

lemma *scc-root-unique-is-scc*:
  assumes *scc-root s v scc*
  shows *scc-root s v* (*scc-of E v*)
proof −
  from *assms* **have** $v \in scc$ *is-scc E scc* **by** (*simp-all add*: *scc-root-def*)
  moreover have $v \in$ *scc-of E v is-scc E* (*scc-of E v*) **by** *simp-all*
  ultimately have *scc* = *scc-of E v* **using** *is-scc-unique* **by** *metis*
  thus *?thesis* **using** *assms* **by** *simp*
qed

lemma *scc-root-finished-impl-scc-finished*:
  assumes $v \in dom$ (*finished s*)
  and *scc-root s v scc*
  shows $scc \subseteq dom$ (*finished s*)
proof
  fix *x*
  assume $x \in scc$

  let *?E* = *Restr E scc*

  from *assms* **have** *is-scc E scc* $v \in scc$ **by** (*simp-all add*: *scc-root-def*)
  hence $(v,x) \in (Restr\ E\ scc)^*$ **using** ‹$x \in scc$›
    by (*simp add*: *is-scc-connected'*)
  with *rtrancl-is-path* **obtain** *p* **where** *path ?E v p x* **by** *metis*
  thus $x \in dom$ (*finished s*)
  proof (*induction p arbitrary*: *x rule*: *rev-induct*)
    case *Nil* **hence** $v = x$ **by** *simp*
    with *assms* **show** *?case* **by** *simp*
  next
    case (*snoc y ys*) **hence** *path ?E v ys y* $(y,x) \in$ *?E*
      by (*simp-all add*: *path-append-conv*)
    with *snoc.IH* **have** $y \in dom$ (*finished s*) **by** *simp*
    moreover from ‹$(y,x) \in$ *?E*› **have** $(y,x) \in E$ $x \in scc$ **by** *auto*
    ultimately have $x \in dom$ (*discovered s*)
      using *finished-imp-succ-discovered*
      by *blast*
    with ‹$x \in scc$› **show** *?case*
      using *assms scc-root-scc-tree-trancl tree-path-impl-parenthesis*
      by *blast*
  qed
qed

context begin interpretation *timing-syntax* **.**
  lemma *scc-root-disc-le*:
    assumes *scc-root s v scc*
    and $x \in scc$ $x \in dom$ (*discovered s*)
    shows $\delta\ s\ v \leq \delta\ s\ x$
  proof (*cases* $x = v$)
    case *False* **with** *assms scc-root-scc-tree-trancl tree-path-disc* **have**
      $\delta\ s\ v < \delta\ s\ x$
      by *blast*
    thus *?thesis* **by** *simp*
  qed *simp*

  lemma *scc-root-fin-ge*:
    assumes *scc-root s v scc*
    and $v \in dom$ (*finished s*)
    and $x \in scc$
    shows $\varphi\ s\ v \geq \varphi\ s\ x$
  proof (*cases* $x = v$)
    case *False*
    from *assms scc-root-finished-impl-scc-finished* **have**
      $x \in dom$ (*finished s*) **by** *auto*
    hence $x \in dom$ (*discovered s*) **using** *finished-discovered* **by** *auto*
    with *assms False* **have** $(v,x) \in (\textit{tree-edges } s)^+$
      using *scc-root-scc-tree-trancl* **by** *simp*
    with *tree-path-impl-parenthesis assms False* **show** *?thesis* **by** *force*
  qed *simp*

  lemma *scc-root-is-Min-disc*:
    assumes *scc-root s v scc*
    shows *Min* ($\delta\ s$ ‘ ($scc \cap dom$ (*discovered s*))) $= \delta\ s\ v$ (**is** *Min ?S* = -)
  proof (*rule Min-eqI*)
    from *discovered-finite* **show** *finite ?S* **by** *auto*
    from *scc-root-disc-le*[*OF assms*] **show** $\bigwedge y.\ y \in\ ?S \Longrightarrow \delta\ s\ v \leq y$ **by** *force*

    from *assms* **have** $v \in scc$ $v \in dom$ (*discovered s*)
      by (*simp-all add*: *scc-root-def*)
    thus $\delta\ s\ v \in\ ?S$ **by** *auto*
  qed

  lemma *Min-disc-is-scc-root*:
    assumes $v \in scc$ $v \in dom$ (*discovered s*)
    and *is-scc E scc*
    and *min*: $\delta\ s\ v =$ *Min* ($\delta\ s$ ‘ ($scc \cap dom$ (*discovered s*)))
    shows *scc-root s v scc*
  proof −
    {
      fix *y*
      assume *A*: $y \in scc$ $y \in dom$ (*discovered s*) $y \neq v$
      with *min* **have** $\delta\ s\ v \leq \delta\ s\ y$ **by** *auto*

```
    with assms disc-unequal A have δ s v < δ s y by fastforce
  } note scc-disc = this

  {
    fix x
    assume A: x ∈ scc ∩ dom (discovered s)

    have x ∈ (tree-edges s)* `` {v}
    proof (cases v = x)
      case False with A scc-disc have δ: δ s v < δ s x by simp

      have (v,x) ∈ (tree-edges s)+
      proof (cases v ∈ dom (finished s))
        case False with stack-set-def assms have
          v-stack: v ∈ set (stack s) by auto
        show ?thesis
        proof (cases x ∈ dom (finished s))
          case True
          with parenthesis-impl-tree-path-not-finished[of v x] assms δ False
          show ?thesis by auto
        next
          case False with A stack-set-def have x ∈ set (stack s) by auto
          with v-stack δ show ?thesis
            using on-stack-is-tree-path
            by simp
        qed
      next
        case True note v-fin = this

        let ?E = Restr E scc

        {
          fix y
          assume (v, y) ∈ ?E and v ≠ y
          hence ∗: y ∈ succ v y ∈ scc by auto
          with finished-imp-succ-discovered v-fin have
            y ∈ dom (discovered s) by simp
          with scc-disc ‹v ≠ y› ∗ have δ s v < δ s y by simp
          with ∗ finished-succ-impl-path-in-tree v-fin have (v,y) ∈ (tree-edges s)+
by simp
        } note trancl-base = this

        from A have x ∈ scc by simp
        with assms have (v,x) ∈ ?E*
          by (simp add: is-scc-connected')
        with ‹v≠x› have (v,x) ∈ ?E+ by (metis rtrancl-eq-or-trancl)
        thus ?thesis using ‹v≠x›
        proof (induction)
          case (base y) with trancl-base show ?case .
```

next
  case (*step y z*)
  show *?case*
  proof (*cases v* = *y*)
    case *True* **with** *step trancl-base* **show** *?thesis* **by** *simp*
  next
    case *False* **with** *step* **have** $(v,y) \in (\textit{tree-edges } s)^+$ **by** *simp*
    with *tree-path-impl-parenthesis*[*OF - v-fin*] **have**
      y-fin: $y \in \textit{dom}$ (*finished s*)
      and *y-t*: $\delta\ s\ v < \delta\ s\ y\ \varphi\ s\ y < \varphi\ s\ v$
      by *auto*
    with *finished-discovered* **have** *y-disc*: $y \in \textit{dom}$ (*discovered s*)
      by *auto*

    from *step* **have** $*$: $z \in \textit{succ}\ y\ z \in \textit{scc}$ **by** *auto*
    with *finished-imp-succ-discovered y-fin* **have**
      z-disc: $z \in \textit{dom}$ (*discovered s*) **by** *simp*
    with $*$ ‹$v \neq z$› **have** δz: $\delta\ s\ v < \delta\ s\ z$ **by** (*simp add*: *scc-disc*)

    from $*$ *edges-covered finished-no-pending*[*OF* ‹$y \in \textit{dom}$ (*finished s*)›]
        y-disc **have** $(y,z) \in \textit{edges } s$ **by** *auto*
    thus *?thesis*
    proof *safe*
        assume $(y,z) \in \textit{tree-edges } s$ **with** ‹$(v,y) \in (\textit{tree-edges } s)^+$› **show**
?thesis **..**
    next
      assume *CE*: $(y,z) \in \textit{cross-edges } s$
      with *cross-edges-finished-decr y-fin y-t* **have** $\varphi\ s\ z < \varphi\ s\ v$
        by *force*
      moreover note δz
      moreover from *CE cross-edges-target-finished* **have**
        $z \in \textit{dom}$ (*finished s*) **by** *simp*
      ultimately show *?thesis*
        using *parenthesis-impl-tree-path*[*OF v-fin*] **by** *metis*
    next
      assume *BE*: $(y,z) \in \textit{back-edges } s$
      with *back-edge-disc-lt-fin y-fin y-t* **have**
        $\delta\ s\ z < \varphi\ s\ v$ **by** *force*
      moreover note δz
      moreover note *z-disc*
      ultimately have $z \in \textit{dom}$ (*finished s*) $\varphi\ s\ z < \varphi\ s\ v$
        using *parenthesis-contained*[*OF v-fin*] **by** *simp-all*
      with δz **show** *?thesis*
        using *parenthesis-impl-tree-path*[*OF v-fin*] **by** *metis*
    qed
  qed
qed
qed

thus *?thesis* **by** *auto*
qed *simp*
}
hence $scc \cap dom\ (discovered\ s) \subseteq (tree\text{-}edges\ s)^*$ `` $\{v\}$ **by** *blast*

with *assms* **show** *?thesis* **by** (*auto intro*: *scc-rootI*)
qed

lemma *scc-root-iff-Min-disc*:
assumes *is-scc E scc r* $\in$ *scc r* $\in$ *dom* (*discovered s*)
shows *scc-root s r scc* $\longleftrightarrow$ *Min* (δ *s* ` (*scc* $\cap$ *dom* (*discovered s*))) = δ *s r* (**is** *?L* $\longleftrightarrow$ *?R*)
proof
assume *?L* **with** *scc-root-is-Min-disc* **show** *?R* **.**
next
assume *?R* **with** *Min-disc-is-scc-root assms* **show** *?L* **by** *simp*
qed

lemma *scc-root-exists*:
assumes *is-scc E scc*
and *scc*: *scc* $\cap$ *dom* (*discovered s*) $\neq$ {}
shows $\exists$ *r. scc-root s r scc*
proof –
let *?S* = *scc* $\cap$ *dom* (*discovered s*)

from *discovered-finite* **have** *finite* (δ *s*`*?S*) **by** *auto*
moreover from *scc* **have** δ *s* ` *?S* $\neq$ {} **by** *auto*
moreover have $\bigwedge$(*x*::*nat*) *f A. x* $\notin$ *f* ` *A* $\vee$ ($\exists$ *y. x* = *f y* $\wedge$ *y* $\in$ *A*) **by** *blast*
— autogenerated by sledgehammer
ultimately have $\exists$ *x* $\in$ *?S.* δ *s x* = *Min* (δ *s* ` *?S*) **by** (*metis Min-in*)
with *Min-disc-is-scc-root* ‹*is-scc E scc*› **show** *?thesis* **by** *auto*
qed

lemma *scc-root-of-node-exists*:
assumes *v* $\in$ *dom* (*discovered s*)
shows $\exists$ *r. scc-root s r* (*scc-of E v*)
proof –
have *is-scc E* (*scc-of E v*) **by** *simp*
moreover have *v* $\in$ *scc-of E v* **by** *simp*
with *assms* **have** *scc-of E v* $\cap$ *dom* (*discovered s*) $\neq$ {} **by** *blast*
ultimately show *?thesis* **using** *scc-root-exists* **by** *metis*
qed

lemma *scc-root-transfer'*:
assumes *discovered s* = *discovered s' tree-edges s* = *tree-edges s'*
shows *scc-root s r scc* $\longleftrightarrow$ *scc-root s' r scc*
unfolding *scc-root-def*
by (*simp add*: *assms*)

lemma *scc-root-transfer*:
  assumes *inv*: *DFS-invar G param s'*
  assumes *r-d*: $r \in dom\ (discovered\ s)$
  assumes *d*: $dom\ (discovered\ s) \subseteq dom\ (discovered\ s')$
    $\forall x{\in}dom\ (discovered\ s).\ \delta\ s\ x = \delta\ s'\ x$
    $\forall x{\in}dom\ (discovered\ s') - dom\ (discovered\ s).\ \delta\ s'\ x \geq counter\ s$
  and *t*: $tree\text{-}edges\ s \subseteq tree\text{-}edges\ s'$
  shows $scc\text{-}root\ s\ r\ scc \longleftrightarrow scc\text{-}root\ s'\ r\ scc$
proof −
  interpret *s'*: *DFS-invar* **where** *s=s'* **by** *fact*

  let $?sd = scc \cap dom\ (discovered\ s)$
  let $?sd' = scc \cap dom\ (discovered\ s')$
  let $?sdd = scc \cap (dom\ (discovered\ s') - dom\ (discovered\ s))$

  {
    assume *r-s*: $r \in scc$ *is-scc E scc*
    with *r-d* **have** *ne*: $\delta\ s`?sd \neq \{\}$ **by** *blast*
    from *discovered-finite* **have** *fin*: *finite* $(\delta\ s'\ `\ ?sd)$ **by** *simp*

    from *timing-less-counter d* **have** $\bigwedge x.\ x{\in}\delta\ s'\ `\ ?sd \Longrightarrow x < counter\ s$ **by** *auto*
    hence *Min*: $Min\ (\delta\ s'\ `\ ?sd) < counter\ s$
      using *Min-less-iff*[*OF fin*] *ne* **by** *blast*

    from *d* **have** $Min\ (\delta\ s\ `\ ?sd) = Min\ (\delta\ s'\ `\ ?sd)$ **by** (*auto simp*: *image-def*)
    also from *d* **have** $?sd' = ?sd \cup ?sdd$ **by** *auto*
    hence $*$: $\delta\ s'\ `\ ?sd' = \delta\ s'\ `\ ?sd \cup \delta\ s'\ `\ ?sdd$ **by** *auto*
    hence $Min\ (\delta\ s'\ `\ ?sd) = Min\ (\delta\ s'\ `\ ?sd')$
    proof (*cases* $?sdd = \{\}$)
      case *False*
      from *d* **have** $\bigwedge x.\ x \in \delta\ s'\ `\ ?sdd \Longrightarrow x \geq counter\ s$ **by** *auto*
      moreover from *False* **have** *ne'*: $\delta\ s'\ `\ ?sdd \neq \{\}$ **by** *blast*
      moreover from *s'.discovered-finite* **have** *fin'*: *finite* $(\delta\ s'\ `\ ?sdd)$ **by** *blast*
      ultimately have $Min\ (\delta\ s'\ `\ ?sdd) \geq counter\ s$
        using *Min-ge-iff* **by** *metis*
      with *Min Min-Un*[*OF fin ne fin' ne'*] $*$ **show** *?thesis* **by** *simp*
    qed *simp*
    finally have $Min\ (\delta\ s\ `\ ?sd) = Min\ (\delta\ s'\ `\ ?sd')$ **.**
  } **note** *aux = this*

  show *?thesis*
  proof
    assume *r*: *scc-root s r scc*
    from *r-d d* **have** $\delta\ s'\ r = \delta\ s\ r$ **by** *simp*
    also from *r scc-root-is-Min-disc* **have** $\delta\ s\ r = Min\ (\delta\ s\ `\ ?sd)$ **by** *simp*
    also from *r aux* **have** $Min\ (\delta\ s\ `\ ?sd) = Min\ (\delta\ s'\ `\ ?sd')$ **by** (*simp add*: *scc-root-def*)
    finally show *scc-root s' r scc*
      using *r-d d r*[*unfolded scc-root-def*]

by (*blast intro*!: *s′.Min-disc-is-scc-root*)
next
assume *r′*: *scc-root s′ r scc*
from *r-d d* **have** δ *s r* = δ *s′ r* **by** *simp*
also from *r′ s′.scc-root-is-Min-disc* **have** δ *s′ r* = *Min* (δ *s′* ‘ *?sd′*) **by** *simp*
also from *r′ aux* **have** *Min* (δ *s′* ‘ *?sd′*) = *Min* (δ *s* ‘ *?sd*) **by** (*simp add*: *scc-root-def*)
finally show *scc-root s r scc*
using *r-d d r′*[*unfolded scc-root-def*]
by (*blast intro*!: *Min-disc-is-scc-root*)
qed
qed

end end

end

1.4 Generic DFS and Refinement

theory *General-DFS-Structure*
imports *../../Param-DFS*
begin

We define the generic structure of DFS algorithms, and use this to define a notion of refinement between DFS algorithms.

named-theorems *DFS-code-unfold* ‹*DFS framework*: *Unfolding theorems to prepare term for automatic refinement*›

lemmas [*DFS-code-unfold*] =
REC-annot-def
GHOST-elim-Let
comp-def

1.4.1 Generic DFS Algorithm

record (*′v*,*′s*) *gen-dfs-struct* =
gds-init :: *′s nres*
gds-is-break :: *′s* ⇒ *bool*
gds-is-empty-stack :: *′s* ⇒ *bool*
gds-new-root :: *′v* ⇒ *′s* ⇒ *′s nres*
gds-get-pending :: *′s* ⇒ (*′v* × *′v option* × *′s*) *nres*
gds-finish :: *′v* ⇒ *′s* ⇒ *′s nres*
gds-is-discovered :: *′v* ⇒ *′s* ⇒ *bool*
gds-is-finished :: *′v* ⇒ *′s* ⇒ *bool*
gds-back-edge :: *′v* ⇒ *′v* ⇒ *′s* ⇒ *′s nres*
gds-cross-edge :: *′v* ⇒ *′v* ⇒ *′s* ⇒ *′s nres*
gds-discover :: *′v* ⇒ *′v* ⇒ *′s* ⇒ *′s nres*

locale *gen-dfs-defs* =
  fixes *gds* :: ('*v*,'*s*) *gen-dfs-struct*
  fixes *V0* :: '*v set*
begin

  definition *gen-step s* ≡
    if gds-is-empty-stack gds s then do {
      v0 ← *SPEC* (λ*v0*. *v0*∈*V0* ∧ ¬*gds-is-discovered gds v0 s*);
      gds-new-root gds v0 s
    } *else do* {
        (*u*, *Vs*,*s*) ← *gds-get-pending gds s*;
        case Vs of
          None ⇒ *gds-finish gds u s*
        | *Some v* ⇒ *do* {
          if gds-is-discovered gds v s then (
            if gds-is-finished gds v s then
              gds-cross-edge gds u v s
            else
              gds-back-edge gds u v s
) *else*
            gds-discover gds u v s
        }
      }

  definition *gen-cond s*
    ≡ (*V0* ⊆ {*v*. *gds-is-discovered gds v s*} ⟶ ¬*gds-is-empty-stack gds s*)
      ∧ ¬*gds-is-break gds s*

  definition *gen-dfs*
    ≡ *gds-init gds* ≫= *WHILE gen-cond gen-step*

  definition *gen-dfsT*
    ≡ *gds-init gds* ≫= *WHILET gen-cond gen-step*

  abbreviation *gen-discovered s* ≡ {*v* . *gds-is-discovered gds v s*}

  abbreviation *gen-rwof* ≡ *rwof* (*gds-init gds*) *gen-cond gen-step*

  definition *pre-new-root v0 s* ≡
    gen-rwof s ∧ *gds-is-empty-stack gds s* ∧ ¬*gds-is-break gds s*
    ∧ *v0*∈*V0* − *gen-discovered s*

  definition *pre-get-pending s* ≡
    gen-rwof s ∧ ¬*gds-is-empty-stack gds s* ∧ ¬*gds-is-break gds s*

  definition *post-get-pending u Vs s0 s* ≡ *pre-get-pending s0*

$\wedge$ *inres* (*gds-get-pending gds s0*) (*u*, *Vs*,*s*)

definition *pre-finish u s0 s* $\equiv$ *post-get-pending u None s0 s*
definition *pre-cross-edge u v s0 s* $\equiv$
  post-get-pending u (*Some v*) *s0 s* $\wedge$ *gds-is-discovered gds v s*
  $\wedge$ *gds-is-finished gds v s*
definition *pre-back-edge u v s0 s* $\equiv$
  post-get-pending u (*Some v*) *s0 s* $\wedge$ *gds-is-discovered gds v s*
    $\wedge$ $\neg$*gds-is-finished gds v s*
definition *pre-discover u v s0 s* $\equiv$
  post-get-pending u (*Some v*) *s0 s* $\wedge$ $\neg$*gds-is-discovered gds v s*

lemmas *pre-defs* = *pre-new-root-def pre-get-pending-def post-get-pending-def*
  pre-finish-def pre-cross-edge-def pre-back-edge-def pre-discover-def

definition *gen-step-assert s* $\equiv$
```
  if gds-is-empty-stack gds s then do {
    v0 ← SPEC (λv0. v0∈V0 ∧ ¬gds-is-discovered gds v0 s);
    ASSERT (pre-new-root v0 s);
    gds-new-root gds v0 s
  } else do {
      ASSERT (pre-get-pending s);
      let s0=GHOST s;
      (u,Vs,s) ← gds-get-pending gds s;
      case Vs of
        None ⇒ do {ASSERT (pre-finish u s0 s); gds-finish gds u s}
      | Some v ⇒ do {
        if gds-is-discovered gds v s then do {
          if gds-is-finished gds v s then do {
            ASSERT (pre-cross-edge u v s0 s);
            gds-cross-edge gds u v s
          } else do {
            ASSERT (pre-back-edge u v s0 s);
            gds-back-edge gds u v s
          }
        } else do {
          ASSERT (pre-discover u v s0 s);
          gds-discover gds u v s
        }
      }
    }
```

definition *gen-dfs-assert*
  $\equiv$ *gds-init gds* $\gg=$ *WHILE gen-cond gen-step-assert*

definition *gen-dfsT-assert*
  $\equiv$ *gds-init gds* $\gg=$ *WHILET gen-cond gen-step-assert*

abbreviation *gen-rwof-assert* $\equiv$ *rwof* (*gds-init gds*) *gen-cond gen-step-assert*

lemma *gen-step-eq-assert*: ⟦*gen-cond s*; *gen-rwof s*⟧
$\Longrightarrow$ *gen-step s* = *gen-step-assert s*
apply (*rule antisym*)
subgoal
apply (*unfold gen-step-def*[*abs-def*] *gen-step-assert-def*[*abs-def*]) []
apply (*unfold GHOST-elim-Let*) []
apply (*rule refine-IdD*)
apply *refine-rcg*
apply *refine-dref-type*
by *simp-all*

subgoal
apply (*simp* (*no-asm*) *only*: *gen-step-def*[*abs-def*] *gen-step-assert-def*[*abs-def*])
[]
apply (*unfold GHOST-elim-Let*) []
apply (*rule refine-IdD*)
apply (*refine-rcg bind-refine′*)
apply *refine-dref-type*
by (*auto simp*: *pre-defs gen-cond-def*)
done

lemma *gen-dfs-eq-assert*: *gen-dfs* = *gen-dfs-assert*
unfolding *gen-dfs-def gen-dfs-assert-def*
apply (*rule antisym*)

subgoal
apply (*unfold gen-step-def*[*abs-def*] *gen-step-assert-def*[*abs-def*]) []
apply (*unfold GHOST-elim-Let*) []
apply (*rule refine-IdD*)
by (*refine-rcg*, *refine-dref-type*, *simp-all*) []

subgoal
apply (*subst* (*2*) *WHILE-eq-I-rwof*)
apply (*rule refine-IdD*)
apply (*refine-rcg*, *simp-all*)

apply (*simp* (*no-asm*) *only*: *gen-step-def*[*abs-def*] *gen-step-assert-def*[*abs-def*])
[]
apply (*unfold GHOST-elim-Let*) []
apply (*rule refine-IdD*)
apply (*refine-rcg bind-refine′*)
apply *refine-dref-type*
by (*auto simp*: *pre-defs gen-cond-def*)
done

lemma *gen-dfsT-eq-assert*: *gen-dfsT* = *gen-dfsT-assert*
unfolding *gen-dfsT-def gen-dfsT-assert-def*
apply (*rule antisym*)

subgoal
apply (*unfold gen-step-def*[*abs-def*] *gen-step-assert-def*[*abs-def*]) []
apply (*unfold GHOST-elim-Let*) []
apply (*rule refine-IdD*)
by (*refine-rcg*, *refine-dref-type*, *simp-all*) []

subgoal
apply (*subst* (*2*) *WHILET-eq-I-rwof*)
apply (*rule refine-IdD*)
apply (*refine-rcg*, *simp-all*)

apply (*simp* (*no-asm*) *only*: *gen-step-def*[*abs-def*] *gen-step-assert-def*[*abs-def*])
[]
apply (*unfold GHOST-elim-Let*) []
apply (*rule refine-IdD*)
apply (*refine-rcg bind-refine′*, *refine-dref-type*)
by (*auto simp*: *pre-defs gen-cond-def*)
done

lemma *gen-rwof-eq-assert*:
assumes *NF*: *nofail gen-dfs*
shows *gen-rwof* = *gen-rwof-assert*
apply (*rule ext*)
apply (*rule iffI*)

subgoal
apply (*rule rwof-step-refine*)
apply (*fold gen-dfs-assert-def gen-dfs-eq-assert*, *rule NF*)
apply *assumption*

apply (*simp* (*no-asm*) *only*: *gen-step-def*[*abs-def*] *gen-step-assert-def*[*abs-def*])
[]
apply (*unfold GHOST-elim-Let*) []
apply (*rule leofI*)
apply (*rule refine-IdD*)
by (*refine-rcg bind-refine′*, *refine-dref-type*,
auto simp: *pre-defs gen-cond-def*) []

subgoal
apply (*rule rwof-step-refine*)
apply (*fold gen-dfs-def*, *rule NF*)
apply *assumption*

apply (*simp* (*no-asm*) *only*: *gen-step-def*[*abs-def*] *gen-step-assert-def*[*abs-def*])
[]
apply (*unfold GHOST-elim-Let*) []
apply (*rule leofI*)
apply (*rule refine-IdD*)

by (*refine-rcg bind-refine'*, *refine-dref-type*,
auto simp: *pre-defs gen-cond-def*) []
done

lemma *gen-dfs-le-gen-dfsT*: *gen-dfs* $\leq$ *gen-dfsT*
unfolding *gen-dfs-def gen-dfsT-def*
apply (*rule bind-mono*)
apply *simp*
unfolding *WHILET-def WHILE-def*
apply (*rule WHILEI-le-WHILEIT*)
done

end

locale *gen-dfs* = *gen-dfs-defs gds V0*
for *gds* :: ('*v*,'*s*) *gen-dfs-struct*
and *V0* :: '*v set*

record ('*v*,'*s*,'*es*) *gen-basic-dfs-struct* =
gbs-init :: '*es* $\Rightarrow$ '*s nres*
gbs-is-empty-stack :: '*s* $\Rightarrow$ *bool*
gbs-new-root :: '*v* $\Rightarrow$ '*s* $\Rightarrow$ '*s nres*
gbs-get-pending :: '*s* $\Rightarrow$ ('*v* $\times$ '*v option* $\times$ '*s*) *nres*
gbs-finish :: '*v* $\Rightarrow$ '*s* $\Rightarrow$ '*s nres*
gbs-is-discovered :: '*v* $\Rightarrow$ '*s* $\Rightarrow$ *bool*
gbs-is-finished :: '*v* $\Rightarrow$ '*s* $\Rightarrow$ *bool*
gbs-back-edge :: '*v* $\Rightarrow$ '*v* $\Rightarrow$ '*s* $\Rightarrow$ '*s nres*
gbs-cross-edge :: '*v* $\Rightarrow$ '*v* $\Rightarrow$ '*s* $\Rightarrow$ '*s nres*
gbs-discover :: '*v* $\Rightarrow$ '*v* $\Rightarrow$ '*s* $\Rightarrow$ '*s nres*

locale *gen-param-dfs-defs* =
fixes *gbs* :: ('*v*,'*s*,'*es*) *gen-basic-dfs-struct*
fixes *param* :: ('*v*,'*s*,'*es*) *gen-parameterization*
fixes *upd-ext* :: ('*es*$\Rightarrow$'*es*) $\Rightarrow$ '*s* $\Rightarrow$ '*s*
fixes *V0* :: '*v set*
begin

definition *do-action bf ef s* $\equiv$ *do* {
s $\leftarrow$ *bf s*;
e $\leftarrow$ *ef s*;
RETURN (*upd-ext* (λ-. *e*) *s*)
}

definition *do-init* ≡ *do* {
  e ← *on-init param*;
  gbs-init gbs e
}

definition *do-new-root v0*
  ≡ *do-action* (*gbs-new-root gbs v0*) (*on-new-root param v0*)

definition *do-finish u*
  ≡ *do-action* (*gbs-finish gbs u*) (*on-finish param u*)

definition *do-back-edge u v*
  ≡ *do-action* (*gbs-back-edge gbs u v*) (*on-back-edge param u v*)

definition *do-cross-edge u v*
  ≡ *do-action* (*gbs-cross-edge gbs u v*) (*on-cross-edge param u v*)

definition *do-discover u v*
  ≡ *do-action* (*gbs-discover gbs u v*) (*on-discover param u v*)

lemmas *do-action-defs*[*DFS-code-unfold*] =
  do-action-def do-init-def do-new-root-def
  do-finish-def do-back-edge-def do-cross-edge-def do-discover-def

definition *gds* ≡ (|
  gds-init = *do-init*,
  gds-is-break = *is-break param*,
  gds-is-empty-stack = *gbs-is-empty-stack gbs*,
  gds-new-root = *do-new-root*,
  gds-get-pending = *gbs-get-pending gbs*,
  gds-finish = *do-finish*,
  gds-is-discovered = *gbs-is-discovered gbs*,
  gds-is-finished = *gbs-is-finished gbs*,
  gds-back-edge = *do-back-edge*,
  gds-cross-edge = *do-cross-edge*,
  gds-discover = *do-discover*
|)

lemmas *gds-simps*[*simp*,*DFS-code-unfold*]
  = *gen-dfs-struct.simps*[*mk-record-simp*, *OF gds-def*]

sublocale *gen-dfs-defs gds V0* **.**
end

locale *gen-param-dfs* = *gen-param-dfs-defs gbs param upd-ext V0*
  for *gbs* :: ('*v*,'*s*,'*es*) *gen-basic-dfs-struct*
  and *param* :: ('*v*,'*s*,'*es*) *gen-parameterization*
  and *upd-ext* :: ('*es*⇒'*es*) ⇒ '*s* ⇒ '*s*
  and *V0* :: '*v set*

```
context param-DFS-defs begin

  definition gbs ≡ (|
    gbs-init = RETURN o empty-state,
    gbs-is-empty-stack = is-empty-stack ,
    gbs-new-root = RETURN oo new-root ,
    gbs-get-pending = get-pending ,
    gbs-finish = RETURN oo finish ,
    gbs-is-discovered = is-discovered ,
    gbs-is-finished = is-finished ,
    gbs-back-edge = RETURN ooo back-edge ,
    gbs-cross-edge = RETURN ooo cross-edge ,
    gbs-discover = RETURN ooo discover
  |)

  lemmas gbs-simps[simp] = gen-basic-dfs-struct.simps[mk-record-simp, OF gbs-def]

  sublocale gen-dfs: gen-param-dfs-defs gbs param state.more-update V0 .

  lemma gen-cond-simp[simp]: gen-dfs.gen-cond = cond
    apply (intro ext)
    unfolding cond-def gen-dfs.gen-cond-def
    by simp

  lemma gen-step-simp[simp]: gen-dfs.gen-step = step
    apply (intro ext)
    unfolding gen-dfs.gen-step-def[abs-def]
    apply (simp
      cong: if-cong option.case-cong
      add: gen-dfs.do-action-defs[abs-def])

    unfolding step-def[abs-def] do-defs get-new-root-def pred-defs
    apply (simp
      cong: if-cong option.case-cong)
    done

  lemma gen-init-simp[simp]: gen-dfs.do-init = init
    unfolding init-def
    apply (simp add: gen-dfs.do-action-defs[abs-def])
    done

  lemma gen-dfs-simp[simp]: gen-dfs.gen-dfs = it-dfs
    unfolding it-dfs-def gen-dfs.gen-dfs-def
    apply (simp)
    done

  lemma gen-dfsT-simp[simp]: gen-dfs.gen-dfsT = it-dfsT
    unfolding it-dfsT-def gen-dfs.gen-dfsT-def
```

apply (*simp*)
done

end

context *param-DFS* **begin**
sublocale *gen-dfs*: *gen-param-dfs gbs param state.more-update V0* **.**
end

1.4.2 Refinement Between DFS Implementations

locale *gen-dfs-refine-defs* =
c: *gen-dfs-defs gdsi V0i* + *a*: *gen-dfs-defs gds V0*
for *gdsi V0i gds V0*

locale *gen-dfs-refine* =
c: *gen-dfs gdsi V0i* + *a*: *gen-dfs gds V0* + *gen-dfs-refine-defs gdsi V0i gds V0*
for *gdsi V0i gds V0* +
fixes *V S*
assumes *BIJV*[*relator-props*]: *bijective V*
assumes *V0-param*[*param*]: (*V0i*, *V0*)∈⟨*V*⟩*set-rel*
assumes *is-discovered-param*[*param*]:
(*gds-is-discovered gdsi*,*gds-is-discovered gds*)∈*V*→*S*→*bool-rel*
assumes *is-finished-param*[*param*]:
(*gds-is-finished gdsi*,*gds-is-finished gds*)∈*V*→*S*→*bool-rel*
assumes *is-empty-stack-param*[*param*]:
(*gds-is-empty-stack gdsi*,*gds-is-empty-stack gds*)∈*S*→*bool-rel*
assumes *is-break-param*[*param*]:
(*gds-is-break gdsi*,*gds-is-break gds*)∈*S*→*bool-rel*
assumes *init-refine*[*refine*]:
gds-init gdsi ≤ ⇓ *S* (*gds-init gds*)
assumes *new-root-refine*[*refine*]:
⟦*a.pre-new-root v0 s*; (*v0i*,*v0*)∈*V*; (*si*,*s*)∈*S*⟧
⟹ *gds-new-root gdsi v0i si* ≤ ⇓ *S* (*gds-new-root gds v0 s*)
assumes *get-pending-refine*[*refine*]:
⟦*a.pre-get-pending s*; (*si*,*s*)∈*S*⟧
⟹ *gds-get-pending gdsi si* ≤ ⇓(*V* $\times_r$ ⟨*V*⟩*option-rel* $\times_r$ *S*) (*gds-get-pending gds s*)
assumes *finish-refine*[*refine*]:
⟦*a.pre-finish v s0 s*; (*vi*,*v*)∈*V*; (*si*,*s*)∈*S*⟧
⟹ *gds-finish gdsi vi si* ≤ ⇓ *S* (*gds-finish gds v s*)
assumes *cross-edge-refine*[*refine*]:
⟦*a.pre-cross-edge u v s0 s*; (*ui*,*u*)∈*V*; (*vi*,*v*)∈*V*; (*si*,*s*)∈*S*⟧
⟹ *gds-cross-edge gdsi ui vi si* ≤ ⇓ *S* (*gds-cross-edge gds u v s*)
assumes *back-edge-refine*[*refine*]:
⟦*a.pre-back-edge u v s0 s*; (*ui*,*u*)∈*V*; (*vi*,*v*)∈*V*; (*si*,*s*)∈*S*⟧
⟹ *gds-back-edge gdsi ui vi si* ≤ ⇓ *S* (*gds-back-edge gds u v s*)
assumes *discover-refine*[*refine*]:
⟦*a.pre-discover u v s0 s*; (*ui*,*u*)∈*V*; (*vi*,*v*)∈*V*; (*si*,*s*)∈*S*⟧

$\Longrightarrow$ *gds-discover gdsi ui vi si* $\leq \Downarrow S$ (*gds-discover gds u v s*)

begin
  term *gds-is-discovered gdsi*

  lemma *select-v0-refine*[*refine*]:
    assumes *s-param*: (*si*,*s*)$\in$*S*
    shows *SPEC* (λ*v0*. *v0* $\in$ *V0i* $\wedge$ $\neg$ *gds-is-discovered gdsi v0 si*)
        $\leq \Downarrow$ *V* (*SPEC* (λ*v0*. *v0* $\in$ *V0* $\wedge$ $\neg$ *gds-is-discovered gds v0 s*))
    apply (*rule RES-refine*)
    apply (*simp add*: *Bex-def*[*symmetric*], *elim conjE*)
    apply (*drule set-relD1*[*OF V0-param*], *elim bexE*)
    apply (*erule bexI*[*rotated*])
    using *is-discovered-param*[*param-fo*, *OF - s-param*]
    apply *auto*
    done

  lemma *gen-rwof-refine*:
    assumes *NF*: *nofail* (*a.gen-dfs*)
    assumes *RW*: *c.gen-rwof s*
    obtains *s'* **where** (*s*,*s'*)$\in$*S* **and** *a.gen-rwof s'*
  proof −
    from *NF* **have** *NFa*: *nofail* (*a.gen-dfs-assert*)
      unfolding *a.gen-dfs-eq-assert* **.**

    have $\exists$ *s'*. (*s*, *s'*) $\in$ *S* $\wedge$ *a.gen-rwof-assert s'*
      apply (*rule rwof-refine*[*OF RW NFa*[*unfolded a.gen-dfs-assert-def*]])

      apply (*rule leofI*, *rule init-refine*)

      unfolding *c.gen-cond-def a.gen-cond-def*
      apply (*rule IdD*)
      apply (*simp only*: *subset-Collect-conv*)
      apply *parametricity*

      unfolding *c.gen-step-def a.gen-step-assert-def GHOST-elim-Let*
      apply (*rule leofI*)
      apply (*refine-rcg IdD*)
      apply *simp-all*
      apply ((*rule IdD*, *parametricity*) | (*auto*) [])+
      done
    thus *?thesis*
      unfolding *a.gen-rwof-eq-assert*[*OF NF*, *symmetric*]
      by (*blast intro*: *that*)
  qed

lemma *gen-step-refine*[*refine*]: $(si,s) \in S \Longrightarrow$ *c.gen-step si* $\leq \Downarrow S$ (*a.gen-step-assert s*)
unfolding *c.gen-step-def a.gen-step-assert-def GHOST-elim-Let*
apply (*refine-rcg IdD*)
apply *simp-all*
apply ((*rule IdD*, *parametricity*) | (*auto*) [])+
done

lemma *gen-dfs-refine*[*refine*]: *c.gen-dfs* $\leq \Downarrow S$ *a.gen-dfs*
unfolding *c.gen-dfs-def a.gen-dfs-eq-assert*[*unfolded a.gen-dfs-assert-def*]
apply *refine-rcg*
unfolding *c.gen-cond-def a.gen-cond-def*
apply (*rule IdD*)
apply (*simp only*: *subset-Collect-conv*)
apply *parametricity*
done

lemma *gen-dfsT-refine*[*refine*]: *c.gen-dfsT* $\leq \Downarrow S$ *a.gen-dfsT*
unfolding *c.gen-dfsT-def a.gen-dfsT-eq-assert*[*unfolded a.gen-dfsT-assert-def*]
apply *refine-rcg*
unfolding *c.gen-cond-def a.gen-cond-def*
apply (*rule IdD*)
apply (*simp only*: *subset-Collect-conv*)
apply *parametricity*
done

end

locale *gbs-refinement* =
c: *gen-param-dfs gbsi parami upd-exti V0i* +
a: *gen-param-dfs gbs param upd-ext V0*
for *gbsi parami upd-exti V0i gbs param upd-ext V0* +
fixes *V S ES*
assumes *BIJV*: *bijective V*
assumes *V0-param*[*param*]: $(V0i, V0) \in \langle V \rangle$*set-rel*

assumes *is-discovered-param*[*param*]:
(*gbs-is-discovered gbsi*,*gbs-is-discovered gbs*)$\in V \rightarrow S \rightarrow$*bool-rel*

assumes *is-finished-param*[*param*]:
(*gbs-is-finished gbsi*,*gbs-is-finished gbs*)$\in V \rightarrow S \rightarrow$*bool-rel*

assumes *is-empty-stack-param*[*param*]:
(*gbs-is-empty-stack gbsi*,*gbs-is-empty-stack gbs*)$\in S \rightarrow$*bool-rel*

assumes *is-break-param*[*param*]:
(*is-break parami*,*is-break param*)∈*S*→*bool-rel*

assumes *gbs-init-refine*[*refine*]: $(ei,\ e) \in ES \Longrightarrow$ *gbs-init gbsi ei* $\le \Downarrow S$ (*gbs-init gbs e*)

assumes *gbs-new-root-refine*[*refine*]:
⟦*a.pre-new-root v0 s*; $(v0i,\ v0) \in V$; $(si,\ s) \in S$⟧
$\Longrightarrow$ *gbs-new-root gbsi v0i si* $\le \Downarrow S$ (*gbs-new-root gbs v0 s*)

assumes *gbs-get-pending-refine*[*refine*]:
⟦*a.pre-get-pending s*; $(si,\ s) \in S$⟧
$\Longrightarrow$ *gbs-get-pending gbsi si*
$\le \Downarrow (V \times_r \langle V\rangle option\text{-}rel \times_r S)$ (*gbs-get-pending gbs s*)

assumes *gbs-finish-refine*[*refine*]:
⟦*a.pre-finish v s0 s*; $(vi,\ v) \in V$; $(si,\ s) \in S$⟧
$\Longrightarrow$ *gbs-finish gbsi vi si* $\le \Downarrow S$ (*gbs-finish gbs v s*)

assumes *gbs-cross-edge-refine*[*refine*]:
⟦*a.pre-cross-edge u v s0 s*; $(ui,\ u) \in V$; $(vi,\ v) \in V$; $(si,\ s) \in S$⟧
$\Longrightarrow$ *gbs-cross-edge gbsi ui vi si* $\le \Downarrow S$ (*gbs-cross-edge gbs u v s*)

assumes *gbs-back-edge-refine*[*refine*]:
⟦*a.pre-back-edge u v s0 s*; $(ui,\ u) \in V$; $(vi,\ v) \in V$; $(si,\ s) \in S$⟧
$\Longrightarrow$ *gbs-back-edge gbsi ui vi si* $\le \Downarrow S$ (*gbs-back-edge gbs u v s*)

assumes *gbs-discover-refine*[*refine*]:
⟦*a.pre-discover u v s0 s*; $(ui,\ u) \in V$; $(vi,\ v) \in V$; $(si,\ s) \in S$⟧
$\Longrightarrow$ *gbs-discover gbsi ui vi si* $\le \Downarrow S$ (*gbs-discover gbs u v s*)

locale *param-refinement* =
c: *gen-param-dfs gbsi parami upd-exti V0i* +
a: *gen-param-dfs gbs param upd-ext V0*
for *gbsi parami upd-exti V0i gbs param upd-ext V0* +
fixes *V S ES*
assumes *upd-ext-param*[*param*]: (*upd-exti*, *upd-ext*)∈$(ES \to ES) \to S \to S$

assumes *on-init-refine*[*refine*]: *on-init parami* $\le \Downarrow ES$ (*on-init param*)

assumes *is-break-param*[*param*]:
(*is-break parami*, *is-break param*) $\in S \to$ *bool-rel*

assumes *on-new-root-refine*[*refine*]:
⟦*a.pre-new-root v0 s*; $(v0i,\ v0) \in V$; $(si,\ s) \in S$;
$(si',\ s') \in S$; *nf-inres* (*gbs-new-root gbs v0 s*) s'⟧
$\Longrightarrow$ *on-new-root parami v0i* $si' \le \Downarrow ES$ (*on-new-root param v0* s')

```
  assumes on-finish-refine[refine]:
    ⟦a.pre-finish v s0 s; (vi, v) ∈ V; (si, s) ∈ S; (si', s') ∈ S;
        nf-inres (gbs-finish gbs v s) s'⟧
        ⟹ on-finish parami vi si' ≤ ⇓ ES (on-finish param v s')

  assumes on-cross-edge-refine[refine]:
    ⟦a.pre-cross-edge u v s0 s; (ui, u) ∈ V; (vi, v) ∈ V; (si, s) ∈ S;
        (si', s') ∈ S; nf-inres (gbs-cross-edge gbs u v s) s'⟧
        ⟹ on-cross-edge parami ui vi si' ≤ ⇓ ES (on-cross-edge param u v s')

  assumes on-back-edge-refine[refine]:
    ⟦a.pre-back-edge u v s0 s; (ui, u) ∈ V; (vi, v) ∈ V; (si, s) ∈ S;
        (si', s') ∈ S; nf-inres (gbs-back-edge gbs u v s) s'⟧
        ⟹ on-back-edge parami ui vi si' ≤ ⇓ ES (on-back-edge param u v s')

  assumes on-discover-refine[refine]:
    ⟦a.pre-discover u v s0 s; (ui, u) ∈ V; (vi, v) ∈ V; (si, s) ∈ S;
        (si', s') ∈ S; nf-inres (gbs-discover gbs u v s) s'⟧
        ⟹ on-discover parami ui vi si' ≤ ⇓ ES (on-discover param u v s')


locale gen-param-dfs-refine-defs =
  c: gen-param-dfs-defs gbsi parami upd-exti V0i +
  a: gen-param-dfs-defs gbs param upd-ext V0
  for gbsi parami upd-exti V0i gbs param upd-ext V0
begin
  sublocale gen-dfs-refine-defs c.gds V0i a.gds V0 .
end

locale gen-param-dfs-refine =
  gbs-refinement where V=V and S=S and ES=ES
+ param-refinement where V=V and S=S and ES=ES
+ gen-param-dfs-refine-defs
  for V :: ('vi×'v) set and S:: ('si×'s) set and ES :: ('esi×'es) set
begin

  sublocale gen-dfs-refine c.gds V0i a.gds V0 V S
    apply unfold-locales
    apply (simp-all add: BIJV V0-param a.do-action-defs c.do-action-defs)
    apply (parametricity+) [4]
    apply refine-rcg
    apply (refine-rcg bind-refine-abs', assumption+, parametricity) []
    apply refine-rcg
    apply (refine-rcg bind-refine-abs', assumption+, parametricity) []
    apply (refine-rcg bind-refine-abs', assumption+, parametricity) []
    apply (refine-rcg bind-refine-abs', assumption+, parametricity) []
    apply (refine-rcg bind-refine-abs', assumption+, parametricity) []
    done
```

end

end

1.5 Tail-Recursive Implementation

```
theory Tailrec-Impl
imports General-DFS-Structure
begin

locale tailrec-impl-defs =
  graph-defs G + gen-dfs-defs gds V0
  for G :: ('v, 'more) graph-rec-scheme
  and gds :: ('v,'s)gen-dfs-struct
begin
  definition [DFS-code-unfold]: tr-impl-while-body ≡ λs. do {
    (u,Vs,s) ← gds-get-pending gds s;
    case Vs of
      None ⇒ gds-finish gds u s
    | Some v ⇒ do {
      if gds-is-discovered gds v s then do {
        if gds-is-finished gds v s then
          gds-cross-edge gds u v s
        else
          gds-back-edge gds u v s
      } else
        gds-discover gds u v s
    }
  }

  definition tailrec-implT where [DFS-code-unfold]:
  tailrec-implT ≡ do {
    s ← gds-init gds;

    FOREACHci
      (λit s.
          gen-rwof s
        ∧ (¬gds-is-break gds s ⟶ gds-is-empty-stack gds s )
        ∧ V0−it ⊆ gen-discovered s)
      V0
      (Not o gds-is-break gds)
      (λv0 s. do {
        let — ghost: s0 = s;
        if gds-is-discovered gds v0 s then
          RETURN s
        else do {
          s ← gds-new-root gds v0 s;
          WHILEIT
            (λs. gen-rwof s ∧ insert v0 (gen-discovered s0) ⊆ gen-discovered s)
```

```
      (λs. ¬gds-is-break gds s ∧ ¬gds-is-empty-stack gds s)
      tr-impl-while-body s
    }
  }) s
}
```

definition *tailrec-impl* **where** [*DFS-code-unfold*]:

```
tailrec-impl ≡ do {
  s ← gds-init gds;

  FOREACHci
    (λit s.
        gen-rwof s
      ∧ (¬gds-is-break gds s ⟶ gds-is-empty-stack gds s )
      ∧ V0−it ⊆ gen-discovered s)
    V0
    (Not o gds-is-break gds)
    (λv0 s. do {
      let — ghost: s0 = s;
      if gds-is-discovered gds v0 s then
        RETURN s
      else do {
        s ← gds-new-root gds v0 s;
        WHILEI
          (λs. gen-rwof s ∧ insert v0 (gen-discovered s0) ⊆ gen-discovered s)
          (λs. ¬gds-is-break gds s ∧ ¬gds-is-empty-stack gds s)
          (λs. do {
            (u,Vs,s) ← gds-get-pending gds s;
            case Vs of
              None ⇒ gds-finish gds u s
            | Some v ⇒ do {
              if gds-is-discovered gds v s then do {
                if gds-is-finished gds v s then
                  gds-cross-edge gds u v s
                else
                  gds-back-edge gds u v s
              } else
                gds-discover gds u v s
            }
          }) s
      }
    }) s
}
```

end

Implementation of general DFS with outer foreach-loop

locale *tailrec-impl* =
fb-graph G + gen-dfs gds V0 + tailrec-impl-defs G gds

for *G* :: (*'v*, *'more*) *graph-rec-scheme*
and *gds* :: (*'v*,*'s*)*gen-dfs-struct*
\+
assumes *init-empty-stack*:
  gds-init gds $\leq_n$ *SPEC* (*gds-is-empty-stack gds*)
assumes *new-root-discovered*:
  ⟦*pre-new-root v0 s*⟧
    ⟹ *gds-new-root gds v0 s* $\leq_n$ *SPEC* (λ*s'*.
      insert v0 (*gen-discovered s*) ⊆ *gen-discovered s'*)
assumes *get-pending-incr*:
  ⟦*pre-get-pending s*⟧ ⟹ *gds-get-pending gds s* $\leq_n$ *SPEC* (λ(-,-,*s'*).
      gen-discovered s ⊆ *gen-discovered s'*
    ~~∧ *gds-is-break gds s'* = *gds-is-break gds s*~~)
assumes *finish-incr*: ⟦*pre-finish u s0 s*⟧
  ⟹ *gds-finish gds u s* $\leq_n$ *SPEC* (λ*s'*.
    gen-discovered s ⊆ *gen-discovered s'*)
assumes *cross-edge-incr*: *pre-cross-edge u v s0 s*
  ⟹ *gds-cross-edge gds u v s* $\leq_n$ *SPEC* (λ*s'*.
    gen-discovered s ⊆ *gen-discovered s'*)
assumes *back-edge-incr*: *pre-back-edge u v s0 s*
  ⟹ *gds-back-edge gds u v s* $\leq_n$ *SPEC* (λ*s'*.
    gen-discovered s ⊆ *gen-discovered s'*)
assumes *discover-incr*: *pre-discover u v s0 s*
  ⟹ *gds-discover gds u v s* $\leq_n$ *SPEC* (λ*s'*.
    gen-discovered s ⊆ *gen-discovered s'*)
begin

context
  assumes *nofail*:
    nofail (*gds-init gds* ≫= *WHILE gen-cond gen-step*)
begin
  lemma *gds-init-refine*: *gds-init gds*
    ≤ *SPEC* (λ*s*. *gen-rwof s* ∧ *gds-is-empty-stack gds s*)
    apply (*rule SPEC-rule-conj-leofI1*)
    apply (*rule rwof-init*[*OF nofail*])
    apply (*rule init-empty-stack*)
    done

  lemma *gds-new-root-refine*:
    assumes *PNR*: *pre-new-root v0 s*
    shows *gds-new-root gds v0 s*
      ≤ *SPEC* (λ*s'*. *gen-rwof s'*
        ∧ *insert v0* (*gen-discovered s*) ⊆ *gen-discovered s'*)
    apply (*rule SPEC-rule-conj-leofI1*)

      apply (*rule order-trans*[*OF* - *rwof-step*[*OF nofail*]])
      using *PNR* **apply** (*unfold gen-step-def gen-cond-def pre-new-root-def*) [*3*]
      apply (*simp add*: *pw-le-iff refine-pw-simps*, *blast*)

apply *simp*
apply *blast*

apply (*rule new-root-discovered*[*OF PNR*])
done

lemma *get-pending-nofail*:
assumes *A*: *pre-get-pending s*
shows *nofail* (*gds-get-pending gds s*)
proof −

from *A*[*unfolded pre-get-pending-def*] **have**
RWOF: *gen-rwof s* **and**
C: ¬ *gds-is-empty-stack gds s* ¬ *gds-is-break gds s*
by *auto*

from *C* **have** *COND*: *gen-cond s* **unfolding** *gen-cond-def* **by** *auto*

from *rwof-step*[*OF nofail RWOF COND*]
have *gen-step s* ≤ *SPEC gen-rwof* **.**
hence *nofail* (*gen-step s*) **by** (*simp add*: *pw-le-iff*)

with *C* **show** *?thesis* **unfolding** *gen-step-def* **by** (*simp add*: *refine-pw-simps*)
qed

lemma *gds-get-pending-refine*:
assumes *PRE*: *pre-get-pending s*
shows *gds-get-pending gds s* ≤ *SPEC* (λ(*u*, *Vs*,*s'*).
post-get-pending u Vs s s'
∧ *gen-discovered s* ⊆ *gen-discovered s'*)
proof −
have *gds-get-pending gds s* ≤ *SPEC* (λ(*u*, *Vs*,*s'*). *post-get-pending u Vs s s'*)
unfolding *post-get-pending-def*
apply (*simp add*: *PRE*)
using *get-pending-nofail*[*OF PRE*]
apply (*simp add*: *pw-le-iff*)
done
moreover note *get-pending-incr*[*OF PRE*]
ultimately show *?thesis* **by** (*simp add*: *pw-le-iff pw-leof-iff*)
qed

lemma *gds-finish-refine*:
assumes *PRE*: *pre-finish u s0 s*
shows *gds-finish gds u s* ≤ *SPEC* (λ*s'*. *gen-rwof s'*
∧ *gen-discovered s* ⊆ *gen-discovered s'*)
apply (*rule SPEC-rule-conj-leofI1*)

apply (*rule order-trans*[*OF* - *rwof-step*[*OF nofail*]])
using *PRE*
apply (*unfold gen-step-def gen-cond-def pre-finish-def post-get-pending-def pre-get-pending-def*) [*3*]
apply (*simp add*: *pw-le-iff refine-pw-simps split*: *option.split*, *blast*)
apply *simp*
apply *blast*

apply (*rule finish-incr*[*OF PRE*])
done

lemma *gds-cross-edge-refine*:
assumes *PRE*: *pre-cross-edge u v s0 s*
shows *gds-cross-edge gds u v s* $\leq$ *SPEC* ($\lambda s'$. *gen-rwof* s' $\wedge$ *gen-discovered s* $\subseteq$ *gen-discovered* s')
apply (*rule SPEC-rule-conj-leofI1*)

apply (*rule order-trans*[*OF* - *rwof-step*[*OF nofail*]])
using *PRE*
apply (*unfold gen-step-def gen-cond-def pre-cross-edge-def post-get-pending-def pre-get-pending-def*) [*3*]
apply (*simp add*: *pw-le-iff refine-pw-simps split*: *option.split*, *blast*)
apply *simp*
apply *blast*

apply (*rule cross-edge-incr*[*OF PRE*])
done

lemma *gds-back-edge-refine*:
assumes *PRE*: *pre-back-edge u v s0 s*
shows *gds-back-edge gds u v s* $\leq$ *SPEC* ($\lambda s'$. *gen-rwof* s' $\wedge$ *gen-discovered s* $\subseteq$ *gen-discovered* s')
apply (*rule SPEC-rule-conj-leofI1*)

apply (*rule order-trans*[*OF* - *rwof-step*[*OF nofail*]])
using *PRE*
apply (*unfold gen-step-def gen-cond-def pre-back-edge-def post-get-pending-def pre-get-pending-def*) [*3*]
apply (*simp add*: *pw-le-iff refine-pw-simps split*: *option.split*, *blast*)
apply *simp*
apply *blast*

apply (*rule back-edge-incr*[*OF PRE*])
done

lemma *gds-discover-refine*:
assumes *PRE*: *pre-discover u v s0 s*
shows *gds-discover gds u v s* $\leq$ *SPEC* ($\lambda s'$. *gen-rwof* s'

$\wedge$ *gen-discovered s* $\subseteq$ *gen-discovered s'*)
apply (*rule SPEC-rule-conj-leofI1*)

apply (*rule order-trans*[*OF* - *rwof-step*[*OF nofail*]])
using *PRE*
apply (*unfold gen-step-def gen-cond-def pre-discover-def post-get-pending-def pre-get-pending-def*) [*3*]
apply (*simp add*: *pw-le-iff refine-pw-simps split*: *option.split*, *blast*)
apply *simp*
apply *blast*

apply (*rule discover-incr*[*OF PRE*])
done

end

lemma *gen-step-disc-incr*:
assumes *nofail gen-dfs*
assumes *gen-rwof s insert v0* (*gen-discovered s0*) $\subseteq$ *gen-discovered s*
assumes $\neg$*gds-is-break gds s* $\neg$*gds-is-empty-stack gds s*
shows *gen-step s* $\leq$ *SPEC* (λ*s. insert v0* (*gen-discovered s0*) $\subseteq$ *gen-discovered s*)
using *assms*
apply (*simp only*: *gen-step-def gen-dfs-def*)
apply (*refine-rcg refine-vcg*
order-trans[*OF gds-init-refine*]
order-trans[*OF gds-new-root-refine*]
order-trans[*OF gds-get-pending-refine*]
order-trans[*OF gds-finish-refine*]
order-trans[*OF gds-cross-edge-refine*]
order-trans[*OF gds-back-edge-refine*]
order-trans[*OF gds-discover-refine*]
)
apply (*auto*
simp: *it-step-insert-iff gen-cond-def*
pre-new-root-def pre-get-pending-def pre-finish-def
pre-cross-edge-def pre-back-edge-def pre-discover-def)
done

theorem *tailrec-impl*: *tailrec-impl* $\leq$ *gen-dfs*
unfolding *gen-dfs-def*
apply (*rule WHILE-refine-rwof*)
unfolding *tailrec-impl-def*
apply (*refine-rcg refine-vcg*
order-trans[*OF gds-init-refine*]
order-trans[*OF gds-new-root-refine*]
order-trans[*OF gds-get-pending-refine*]
order-trans[*OF gds-finish-refine*]

order-trans[*OF gds-cross-edge-refine*]
order-trans[*OF gds-back-edge-refine*]
order-trans[*OF gds-discover-refine*]
)
apply (*auto*
simp: *it-step-insert-iff gen-cond-def*
pre-new-root-def pre-get-pending-def pre-finish-def
pre-cross-edge-def pre-back-edge-def pre-discover-def)
done

lemma *tr-impl-while-body-gen-step*:
assumes [*simp*]: $\neg$*gds-is-empty-stack gds s*
shows *tr-impl-while-body s* $\leq$ *gen-step s*
unfolding *tr-impl-while-body-def gen-step-def*
by *simp*

lemma *tailrecT-impl*: *tailrec-implT* $\leq$ *gen-dfsT*
proof (*rule le-nofailI*)
let *?V* = *rwof-rel* (*gds-init gds*) *gen-cond gen-step*
assume *NF*: *nofail gen-dfsT*
from *nofail-WHILEIT-wf-rel*[*of gds-init gds* λ-. *True gen-cond gen-step*]
and *this*[*unfolded gen-dfsT-def WHILET-def*]
have *WF*: *wf* ($?V^{-1}$) **by** *simp*

from *NF* **have** *NF'*: *nofail gen-dfs* **using** *gen-dfs-le-gen-dfsT*
by (*auto simp*: *pw-le-iff*)

from *rwof-rel-spec*[*of gds-init gds gen-cond gen-step*] **have**
$\bigwedge s$. ⟦*gen-rwof s*; *gen-cond s*⟧ $\Longrightarrow$ *gen-step s* $\leq_n$ *SPEC* ($\lambda s'$. $(s,s') \in ?V$)
.
hence
aux: $\bigwedge s$. ⟦*gen-rwof s*; *gen-cond s*⟧ $\Longrightarrow$ *gen-step s* $\leq$ *SPEC* ($\lambda s'$. $(s,s') \in ?V$)
apply (*rule leofD*[*rotated*])
apply *assumption*
apply *assumption*
using *NF*[*unfolded gen-dfsT-def*]
by (*drule* (*1*) *WHILET-nofail-imp-rwof-nofail*)

show *?thesis*
apply (*rule order-trans*[*OF - gen-dfs-le-gen-dfsT*])
apply (*rule order-trans*[*OF - tailrec-impl*])
unfolding *tailrec-implT-def tailrec-impl-def*
unfolding *tr-impl-while-body-def*[*symmetric*]
apply (*rule refine-IdD*)
apply (*refine-rcg bind-refine' inj-on-id*)
apply *refine-dref-type*
apply *simp-all*
apply (*subst WHILEIT-eq-WHILEI-tproof*[**where** $V = ?V^{-1}$])
apply (*rule WF*; *fail*)

```
    subgoal
      apply clarsimp
      apply (rule order-trans[OF tr-impl-while-body-gen-step], assumption)
      apply (rule aux, assumption, (simp add: gen-cond-def; fail))
    done
    apply (simp; fail)
  done
qed

end
end
```

1.6 Recursive DFS Implementation

```
theory Rec-Impl
imports General-DFS-Structure
begin

locale rec-impl-defs =
  graph-defs G + gen-dfs-defs gds V0
  for G :: ('v, 'more) graph-rec-scheme
  and gds :: ('v,'s)gen-dfs-struct
  +
  fixes pending :: 's ⇒ 'v rel
  fixes stack :: 's ⇒ 'v list
  fixes choose-pending :: 'v ⇒ 'v option ⇒ 's ⇒ 's nres
begin

  definition gen-step' s ≡ do { ASSERT (gen-rwof s);
    if gds-is-empty-stack gds s then do {
      v0 ← SPEC (λv0. v0 ∈ V0 ∧ ¬ gds-is-discovered gds v0 s);
      gds-new-root gds v0 s
    } else do {
      let u = hd (stack s);
      Vs ← SELECT (λv. (u,v)∈pending s);
      s ← choose-pending u Vs s;
      case Vs of
        None ⇒ gds-finish gds u s
      | Some v ⇒
          if gds-is-discovered gds v s
          then if gds-is-finished gds v s then gds-cross-edge gds u v s
               else gds-back-edge gds u v s
          else gds-discover gds u v s
    }}

  definition gen-dfs' ≡ gds-init gds ≫= WHILE gen-cond gen-step'
  abbreviation gen-rwof' ≡ rwof (gds-init gds) gen-cond gen-step'

  definition rec-impl where [DFS-code-unfold]:
```

```
rec-impl ≡ do {
  s ← gds-init gds;

  FOREACHci
    (λit s.
        gen-rwof' s
      ∧ (¬gds-is-break gds s ⟶ gds-is-empty-stack gds s
          ∧ V0−it ⊆ gen-discovered s))
    V0
    (Not o gds-is-break gds)
    (λv0 s. do {
      let s0 = GHOST s;
      if gds-is-discovered gds v0 s then
        RETURN s
      else do {
        s ← gds-new-root gds v0 s;
        if gds-is-break gds s then
          RETURN s
        else do {
          REC-annot
          (λ(u,s). gen-rwof' s ∧ ¬gds-is-break gds s
              ∧ (∃ stk. stack s = u#stk)
              ∧ E ∩ {u}×UNIV ⊆ pending s)
          (λ(u,s) s'.
              gen-rwof' s'
              ∧ (¬gds-is-break gds s' ⟶
                  stack s' = tl (stack s)
                ∧ pending s' = pending s − {u} × UNIV
                ∧ gen-discovered s' ⊇ gen-discovered s
                ))
          (λD (u,s). do {
            s ← FOREACHci
              (λit s'. gen-rwof' s'
              ∧ (¬gds-is-break gds s' ⟶
                  stack s' = stack s
                ∧ pending s' = (pending s − {u}×(E``{u} − it))
                ∧ gen-discovered s' ⊇ gen-discovered s ∪ (E``{u} − it)
                ))
              (E``{u}) (λs. ¬gds-is-break gds s)
              (λv s. do {
                s ← choose-pending u (Some v) s;
                if gds-is-discovered gds v s then do {
                  if gds-is-finished gds v s then
                    gds-cross-edge gds u v s
                  else
                    gds-back-edge gds u v s
                } else do {
                  s ← gds-discover gds u v s;
                  if gds-is-break gds s then RETURN s else D (v,s)
```

```
          }
        })
        s;
      if gds-is-break gds s then
        RETURN s
      else do {
        s ← choose-pending u (None) s;
        s ← gds-finish gds u s;
        RETURN s
      }
    }) (v0,s)
  }
 }
}) s
}
```

definition *rec-impl-for-paper* **where** *rec-impl-for-paper* ≡ *do* {

```
s ← gds-init gds;
FOREACHc V0 (Not o gds-is-break gds) (λv0 s. do {
  if gds-is-discovered gds v0 s then RETURN s
  else do {
    s ← gds-new-root gds v0 s;
    if gds-is-break gds s then RETURN s
    else do {
      REC (λD (u,s). do {
        s ← FOREACHc (E``{u}) (λs. ¬gds-is-break gds s) (λv s. do {
            s ← choose-pending u (Some v) s;
            if gds-is-discovered gds v s then do {
              if gds-is-finished gds v s then gds-cross-edge gds u v s
              else gds-back-edge gds u v s
            } else do {
              s ← gds-discover gds u v s;
              if gds-is-break gds s then RETURN s else D (v,s)
            }
          })
          s;
        if gds-is-break gds s then RETURN s
        else do {
          s ← choose-pending u (None) s;
          gds-finish gds u s
        }
      }) (v0,s)
    }
  }
}) s
}
```

end

locale *rec-impl* =
fb-graph G + gen-dfs gds V0 + rec-impl-defs G gds pending stack choose-pending
for *G* :: ('v, 'more) *graph-rec-scheme*
and *gds* :: ('v,'s)*gen-dfs-struct*
and *pending* :: 's ⇒ 'v *rel*
and *stack* :: 's ⇒ 'v *list*
and *choose-pending* :: 'v ⇒ 'v *option* ⇒ 's ⇒ 's *nres*
+
assumes [*simp*]: *gds-is-empty-stack gds s* ⟷ *stack s* = []
assumes *init-spec*:
gds-init gds $\le_n$ *SPEC* (λ*s*. *stack s* = [] ∧ *pending s* = {})
assumes *new-root-spec*:
⟦*pre-new-root v0 s*⟧
⟹ *gds-new-root gds v0 s* $\le_n$ *SPEC* (λ*s'*.
stack s' = [*v0*] ∧ *pending s'* = {*v0*}×*E*``{*v0*} ∧
gen-discovered s' = *insert v0* (*gen-discovered s*))

assumes *get-pending-fmt*: ⟦ *pre-get-pending s* ⟧ ⟹
do {
let u = *hd* (*stack s*);
vo ← *SELECT* (λ*v*. (*u*,*v*)∈*pending s*);
s ← *choose-pending u vo s*;
RETURN (*u*,*vo*,*s*)
}
≤ *gds-get-pending gds s*

assumes *choose-pending-spec*: ⟦*pre-get-pending s*; *u* = *hd* (*stack s*);
case vo of
None ⇒ *pending s* `` {*u*} = {}
| *Some v* ⇒ *v*∈*pending s* `` {*u*}
⟧ ⟹
choose-pending u vo s $\le_n$ *SPEC* (λ*s'*.
(*case vo of*
None ⇒ *pending s'* = *pending s*
| *Some v* ⇒ *pending s'* = *pending s* − {(*u*,*v*)}) ∧
stack s' = *stack s* ∧
(∀ *x*. *gds-is-discovered gds x s'* = *gds-is-discovered gds x s*)
~~∧ *gds-is-break gds s'* = *gds-is-break gds s*~~
)
assumes *finish-spec*: ⟦*pre-finish u s0 s*⟧
⟹ *gds-finish gds u s* $\le_n$ *SPEC* (λ*s'*.
pending s' = *pending s* ∧
stack s' = *tl* (*stack s*) ∧
(∀ *x*. *gds-is-discovered gds x s'* = *gds-is-discovered gds x s*))
assumes *cross-edge-spec*: *pre-cross-edge u v s0 s*
⟹ *gds-cross-edge gds u v s* $\le_n$ *SPEC* (λ*s'*.
pending s' = *pending s* ∧ *stack s'* = *stack s* ∧
(∀ *x*. *gds-is-discovered gds x s'* = *gds-is-discovered gds x s*))

assumes *back-edge-spec*: *pre-back-edge u v s0 s*
$\Longrightarrow$ *gds-back-edge gds u v s* $\leq_n$ *SPEC* $(\lambda s'$.
pending s' = *pending s* $\wedge$ *stack* s' = *stack s* $\wedge$
$(\forall\, x.$ *gds-is-discovered gds x* s' = *gds-is-discovered gds x s*))
assumes *discover-spec*: *pre-discover u v s0 s*
$\Longrightarrow$ *gds-discover gds u v s* $\leq_n$ *SPEC* $(\lambda s'$.
pending s' = *pending s* $\cup$ ($\{v\} \times E``\{v\}$) $\wedge$ *stack* s' = *v#stack s* $\wedge$
gen-discovered s' = *insert v* (*gen-discovered s*))

begin

lemma *gen-step'-refine*:
⟦*gen-rwof s*; *gen-cond s*⟧ $\Longrightarrow$ *gen-step' s* $\leq$ *gen-step s*
apply (*simp only*: *gen-step'-def gen-step-def*)
apply (*clarsimp*)
apply (*rule order-trans*[*OF* - *bind-mono*(*1*)[*OF get-pending-fmt order-refl*]])
apply (*simp add*: *pw-le-iff refine-pw-simps*
split: *option.splits if-split*)
apply (*simp add*: *pre-defs gen-cond-def*)
done

lemma *gen-dfs'-refine*: *gen-dfs'* $\leq$ *gen-dfs*
unfolding *gen-dfs'-def gen-dfs-def WHILE-eq-I-rwof*[**where** *f*=*gen-step*]
apply (*rule refine-IdD*)
apply (*refine-rcg*)
by (*simp-all add*: *gen-step'-refine*)

lemma *gen-rwof'-imp-rwof*:
assumes *NF*: *nofail gen-dfs*
assumes *A*: *gen-rwof' s*
shows *gen-rwof s*
apply (*rule rwof-step-refine*)
apply (*rule NF*[*unfolded gen-dfs-def*])

apply *fact*

apply (*rule leof-lift*[*OF gen-step'-refine*], *assumption*+) []
done

lemma *reachable-invar*:
gen-rwof' s $\Longrightarrow$ *set* (*stack s*) $\subseteq$ *reachable* $\wedge$ *pending s* $\subseteq$ *E*
$\wedge$ *set* (*stack s*) $\subseteq$ *gen-discovered s* $\wedge$ *distinct* (*stack s*)
$\wedge$ *pending s* $\subseteq$ *set* (*stack s*) $\times$ *UNIV*
apply (*erule establish-rwof-invar*[*rotated* -1])

apply (*rule leof-trans*[*OF init-spec*], *auto*) []
apply (*subst gen-step′-def*)
apply (*refine-rcg refine-vcg*
  leof-trans[*OF new-root-spec*]
  SELECT-rule[*THEN leof-lift*]
  leof-trans[*OF choose-pending-spec*[*THEN leof-strengthen-SPEC*]]
  leof-trans[*OF finish-spec*]
  leof-trans[*OF cross-edge-spec*]
  leof-trans[*OF back-edge-spec*]
  leof-trans[*OF discover-spec*]
)

apply *simp-all*
subgoal by (*simp add*: *pre-defs*, *simp add*: *gen-cond-def*)
subgoal by *auto*
subgoal by *auto*
subgoal by *auto*
subgoal by (*simp add*: *pre-defs*, *simp add*: *gen-cond-def*)

apply ((*unfold pre-defs*, *intro conjI*); *assumption?*) []
  subgoal by (*clarsimp simp*: *gen-cond-def*)
  subgoal by (*clarsimp simp*: *gen-cond-def*)
  subgoal
    apply (*rule pwD2*[*OF get-pending-fmt*])
      subgoal by (*simp add*: *pre-defs gen-cond-def*)
      subgoal by (*clarsimp simp*: *refine-pw-simps*; *blast*)
    done
  subgoal by (*force simp*: *neq-Nil-conv*) []

subgoal by (*clarsimp simp*: *neq-Nil-conv gen-cond-def*, *blast*) []
subgoal by (*clarsimp simp*: *neq-Nil-conv gen-cond-def*; *auto*)

apply (*unfold pre-defs*, *intro conjI*, *assumption*) []
  subgoal by (*clarsimp-all simp*: *gen-cond-def*)
  subgoal by (*clarsimp-all simp*: *gen-cond-def*)
  apply (*rule pwD2*[*OF get-pending-fmt*])
    apply (*simp add*: *pre-defs gen-cond-def*; *fail*)
    apply (*clarsimp simp*: *refine-pw-simps select-def*, *blast*; *fail*)
    apply (*simp*; *fail*)
    apply (*simp*; *fail*)

subgoal by *auto*
subgoal by *fast*

apply (*unfold pre-defs*, *intro conjI*, *assumption*) []
  apply (*clarsimp simp*: *gen-cond-def*; *fail*)

apply (*clarsimp simp*: *gen-cond-def*; *fail*)
apply (*rule pwD2*[*OF get-pending-fmt*])
apply (*simp add*: *pre-defs gen-cond-def*; *fail*)
apply (*clarsimp simp*: *refine-pw-simps select-def*, *blast*; *fail*)
apply (*simp*; *fail*)

subgoal
apply *clarsimp*
by (*meson ImageI SigmaD1 rtrancl-image-unfold-right subset-eq*)

subgoal
apply *clarsimp*
by *blast*

apply *force*
apply *force*
apply *fast*
apply (*auto simp*: *pre-defs gen-cond-def*; *fail*)
apply *fast*

apply ((*unfold pre-defs*, *intro conjI*); *assumption?*)
apply (*clarsimp simp*: *gen-cond-def*; *fail*)
apply (*clarsimp simp*: *gen-cond-def*; *fail*)
apply (*rule pwD2*[*OF get-pending-fmt*])
apply (*simp add*: *pre-defs gen-cond-def*; *fail*)
apply (*clarsimp simp*: *refine-pw-simps*; *fail*)

apply (*auto simp*: *neq-Nil-conv*; *fail*)
apply (*auto simp*: *neq-Nil-conv*; *fail*)
apply (*clarsimp simp*: *neq-Nil-conv*; *blast*)
done

lemma *mk-spec-aux*:
$[\![m \le_n SPEC\ \Phi;\ m \le SPEC\ gen\text{-}rwof'\,]\!] \Longrightarrow m \le SPEC\ (\lambda s.\ gen\text{-}rwof'\ s \wedge \Phi$
$s)$
by (*rule SPEC-rule-conj-leofI1*)

definition *post-choose-pending u vo s0 s* ≡
gen-rwof′ s0
∧ *gen-cond s0*
∧ *stack s0* ≠ []
∧ *u*=*hd* (*stack s0*)
∧ *inres* (*choose-pending u vo s0*) *s*
∧ *stack s* = *stack s0*
∧ (∀ *x*. *gds-is-discovered gds x s* = *gds-is-discovered gds x s0*)
~~∧ gds-is-break gds s = gds-is-break gds s0~~
∧ (*case vo of*
None ⇒ *pending s0*‘‘{*u*}={} ∧ *pending s* = *pending s0*

| *Some v* ⇒ *v* ∈ *pending s0*‘‘{*u*} ∧ *pending s* = *pending s0* − {(*u*,*v*)})

context
  assumes *nofail*:
    nofail (*gds-init gds* ≫= *WHILE gen-cond gen-step*′)
  assumes *nofail2*:
    nofail (*gen-dfs*)
begin
  lemma *pcp-imp-pgp*:
    post-choose-pending u vo s0 s ⟹ *post-get-pending u vo s0 s*
    unfolding *post-choose-pending-def pre-defs*
    apply (*intro conjI*)
    apply (*simp add*: *gen-rwof*′*-imp-rwof*[*OF nofail2*])
    apply *simp*
    apply (*simp add*: *gen-cond-def*)
    apply (*rule pwD2*[*OF get-pending-fmt*])
    apply (*simp add*: *pre-defs gen-cond-def*
        gen-rwof′*-imp-rwof*[*OF nofail2*])
    apply (*auto simp add*: *refine-pw-simps select-def split*: *option.splits*) []
    done

  schematic-goal *gds-init-refine*: *?prop*
    apply (*rule mk-spec-aux*[*OF init-spec*])
    apply (*rule rwof-init*[*OF nofail*])
    done

  schematic-goal *gds-new-root-refine*:
    ⟦*pre-new-root v0 s*; *gen-rwof*′ *s*⟧ ⟹ *gds-new-root gds v0 s* ≤ *SPEC ?*Φ
    apply (*rule mk-spec-aux*[*OF new-root-spec*], *assumption*)
    apply (*rule order-trans*[*OF - rwof-step*[*OF nofail*, **where** *s*=*s*]])
    unfolding *gen-step*′*-def pre-new-root-def gen-cond-def*
    apply (*auto simp*: *pw-le-iff refine-pw-simps*)
    done

  schematic-goal *gds-choose-pending-refine*:
    assumes *1*: *pre-get-pending s*
    assumes *2*: *gen-rwof*′ *s*
    assumes [*simp*]: *u*=*hd* (*stack s*)
    assumes *3*: *case vo of*
        None ⇒ *pending s* ‘‘ {*u*} = {}
      | *Some v* ⇒ *v* ∈ *pending s* ‘‘ {*u*}
    shows *choose-pending u vo s* ≤ *SPEC* (*post-choose-pending u vo s*)
  proof −
    from *WHILE-nofail-imp-rwof-nofail*[*OF nofail 2*] *1 3* **have**
      nofail (*choose-pending u vo s*)
      unfolding *pre-defs gen-step*′*-def gen-cond-def*
      by (*auto simp*: *refine-pw-simps select-def*
        split: *option.splits if-split-asm*)
    also have *choose-pending u vo s* $\leq_n$ *SPEC* (*post-choose-pending u vo s*)

apply (*rule leof-trans*[*OF choose-pending-spec*[*OF 1 - 3* , *THEN leof-strengthen-SPEC*]])
apply *simp*
apply (*rule leof-RES-rule*)
using *1*
apply (*simp add*: *post-choose-pending-def 2 pre-defs gen-cond-def split*: *option.splits*)
using *3*
apply *auto*
done
finally (*leofD*) **show** *?thesis* **.**
qed

schematic-goal *gds-finish-refine*:
$[\![$*pre-finish u s0 s*; *post-choose-pending u None s0 s*$]\!] \Longrightarrow$ *gds-finish gds u s* $\leq$ *SPEC ?*Φ
apply (*rule mk-spec-aux*[*OF finish-spec*], *assumption*)
apply (*rule order-trans*[*OF - rwof-step*[*OF nofail*, **where** *s=s0*]])
unfolding *gen-step'-def pre-defs gen-cond-def post-choose-pending-def*
apply (*auto simp*: *pw-le-iff refine-pw-simps split*: *option.split*)
done

schematic-goal *gds-cross-edge-refine*:
$[\![$*pre-cross-edge u v s0 s*; *post-choose-pending u* (*Some v*) *s0 s*$]\!] \Longrightarrow$ *gds-cross-edge gds u v s* $\leq$ *SPEC ?*Φ
apply (*rule mk-spec-aux*[*OF cross-edge-spec*], *assumption*)
apply (*rule order-trans*[*OF - rwof-step*[*OF nofail*, **where** *s=s0*]])
unfolding *gen-step'-def pre-defs gen-cond-def post-choose-pending-def*
apply (*simp add*: *pw-le-iff refine-pw-simps select-def split*: *option.split*, *blast*)
apply *simp*
apply *blast*
done

schematic-goal *gds-back-edge-refine*:
$[\![$*pre-back-edge u v s0 s*; *post-choose-pending u* (*Some v*) *s0 s*$]\!] \Longrightarrow$ *gds-back-edge gds u v s* $\leq$ *SPEC ?*Φ
apply (*rule mk-spec-aux*[*OF back-edge-spec*], *assumption*)
apply (*rule order-trans*[*OF - rwof-step*[*OF nofail*, **where** *s=s0*]])
unfolding *gen-step'-def pre-defs gen-cond-def post-choose-pending-def*
apply (*simp add*: *pw-le-iff refine-pw-simps select-def split*: *option.split*, *blast*)
apply *simp*
apply *blast*
done

schematic-goal *gds-discover-refine*:
$[\![$*pre-discover u v s0 s*; *post-choose-pending u* (*Some v*) *s0 s*$]\!] \Longrightarrow$ *gds-discover gds u v s* $\leq$ *SPEC ?*Φ
apply (*rule mk-spec-aux*[*OF discover-spec*], *assumption*)
apply (*rule order-trans*[*OF - rwof-step*[*OF nofail*, **where** *s=s0*]])
unfolding *gen-step'-def pre-defs gen-cond-def post-choose-pending-def*

apply (*simp add*: *pw-le-iff refine-pw-simps select-def split*: *option.split*, *blast*)
apply *simp*
apply *blast*
done
end

lemma *rec-impl-aux*: $\llbracket$ *xd*$\notin$*Domain P* $\rrbracket \Longrightarrow$ *P* $-$ $\{y\}$ $\times$ (*succ y* $-$ *ita*) $-$ $\{(y,$ *xd*$)\}$ $-$ $\{xd\}$ $\times$ *UNIV* $=$
P $-$ *insert* (*y*, *xd*) ($\{y\}$ $\times$ (*succ y* $-$ *ita*))
apply *auto*
done

lemma *rec-impl*: *rec-impl* $\leq$ *gen-dfs*
apply (*rule le-nofailI*)
apply (*rule order-trans*[*OF - gen-dfs′-refine*])
unfolding *gen-dfs′-def*
apply (*rule WHILE-refine-rwof*)
unfolding *rec-impl-def*
apply (*refine-rcg refine-vcg*
order-trans[*OF gds-init-refine*]
order-trans[*OF gds-choose-pending-refine*]
order-trans[*OF gds-new-root-refine*]
order-trans[*OF gds-finish-refine*]
order-trans[*OF gds-back-edge-refine*]
order-trans[*OF gds-cross-edge-refine*]
order-trans[*OF gds-discover-refine*]
)
apply (*simp-all split*: *if-split-asm*)

using [[*goals-limit* = *1*]]

apply (*auto simp add*: *pre-defs*; *fail*)
apply (*auto simp add*: *pre-defs gen-rwof′-imp-rwof*; *fail*)
apply (*auto*; *fail*)
apply (*auto dest*: *reachable-invar*; *fail*)
apply (*auto simp add*: *pre-defs gen-rwof′-imp-rwof*; *fail*)
apply (*auto*; *fail*)
apply (*auto*; *fail*)

apply ((*drule pcp-imp-pgp*, *auto simp add*: *pre-defs gen-rwof′-imp-rwof*); *fail*)

apply (*auto simp*: *post-choose-pending-def*; *fail*)
apply (*auto simp*: *post-choose-pending-def*; *fail*)
apply (*auto simp*: *post-choose-pending-def*; *fail*)

apply ((*drule pcp-imp-pgp*, *auto simp add*: *pre-defs gen-rwof′-imp-rwof*); *fail*)

apply (*auto simp*: *post-choose-pending-def*; *fail*)
apply (*auto simp*: *post-choose-pending-def*; *fail*)
apply (*auto simp*: *post-choose-pending-def*; *fail*)

apply ((*drule pcp-imp-pgp*, *auto simp add*: *pre-defs gen-rwof′-imp-rwof*); *fail*)

apply (*rule order-trans*)
apply *rprems*
apply (*auto*; *fail*) []
subgoal
  apply (*rule SPEC-rule*)
  apply (*simp add*: *post-choose-pending-def gen-rwof′-imp-rwof*
    split: *if-split-asm*)
  apply (*clarsimp*
    simp: *gen-rwof′-imp-rwof Un-Diff*
    split: *if-split-asm*) []
  apply (*clarsimp simp*: *it-step-insert-iff neq-Nil-conv*)
  apply (*rule conjI*)
  subgoal
    apply (*rule rec-impl-aux*)
    apply (*drule reachable-invar*)+
    apply (*metis Domain.cases SigmaD1 mem-Collect-eq rev-subsetD*)
  done
  subgoal
    apply (*rule conjI*)
    apply *auto* []
    apply (*metis order-trans*)
  done
done

apply (*auto simp add*: *pre-defs gen-rwof′-imp-rwof*; *fail*)
apply (*auto*; *fail*)
apply (*auto dest*: *reachable-invar*; *fail*)

apply ((*drule pcp-imp-pgp*, *auto simp add*: *pre-defs gen-rwof′-imp-rwof*); *fail*)

apply (*auto simp*: *post-choose-pending-def*; *fail*)
apply (*auto simp*: *post-choose-pending-def*; *fail*)
apply (*auto simp*: *post-choose-pending-def*; *fail*)

apply (*auto*; *fail*)

apply (*auto simp*: *gen-cond-def*; *fail*)

apply (*auto simp*: *gen-cond-def*; *fail*)
done

end

end

1.7 Simple Data Structures

theory *Simple-Impl*
imports
  ../Structural/Rec-Impl
  ../Structural/Tailrec-Impl
begin

We provide some very basic data structures to implement the DFS state

1.7.1 Stack, Pending Stack, and Visited Set

record $'v$ *simple-state* =
  ss-stack :: $('v \times 'v\ set)\ list$
  on-stack :: $'v\ set$
  visited :: $'v\ set$

definition [*to-relAPP*]: *simple-state-rel erel* $\equiv$ { (s,s') .
  ss-stack s = *map* $(\lambda u.\ (u, pending\ s'\ ``\ \{u\}))\ (stack\ s')$ $\wedge$
  on-stack s = *set* $(stack\ s')$ $\wedge$
  visited s = *dom* $(discovered\ s')$ $\wedge$
  dom $(finished\ s')$ = *dom* $(discovered\ s') - set\ (stack\ s')$ $\wedge$ — TODO: Hmm, this is an invariant of the abstract
  set $(stack\ s') \subseteq dom\ (discovered\ s')$ $\wedge$
  $(simple\text{-}state.more\ s,\ state.more\ s') \in erel$
}

lemma *simple-state-relI*:
  assumes
  dom $(finished\ s')$ = *dom* $(discovered\ s') - set\ (stack\ s')$
  set $(stack\ s') \subseteq dom\ (discovered\ s')$
  $(m',\ state.more\ s') \in erel$
  shows (⦇
    ss-stack = *map* $(\lambda u.\ (u, pending\ s'\ ``\ \{u\}))\ (stack\ s')$,
    on-stack = *set* $(stack\ s')$,
    visited = *dom* $(discovered\ s')$,
    $\ldots = m'$
  ⦈, s')$\in\langle erel\rangle$*simple-state-rel*
  using *assms*
  unfolding *simple-state-rel-def*
  by *auto*

lemma *simple-state-more-refine*[*param*]:
  (*simple-state.more-update*, *state.more-update*)
    $\in (R \to R) \to \langle R\rangle simple\text{-}state\text{-}rel \to \langle R\rangle simple\text{-}state\text{-}rel$
  apply (*clarsimp simp*: *simple-state-rel-def*)
  apply *parametricity*

done

We outsource the definitions in a separate locale, as we want to re-use them for similar implementations

locale *pre-simple-impl* = *graph-defs*
begin

definition *init-impl e*
$\equiv$ *RETURN* (| *ss-stack* = [], *on-stack* = {}, *visited* = {}, ... = *e* |)

definition *is-empty-stack-impl s* $\equiv$ (*ss-stack s* = [])
definition *is-discovered-impl u s* $\equiv$ ($u \in$ *visited s*)
definition *is-finished-impl u s* $\equiv$ ($u \in$ *visited s* $-$ (*on-stack s*))

definition *finish-impl u s* $\equiv$ *do* {
ASSERT (*ss-stack s* $\neq$ [] $\wedge$ $u \in$ *on-stack s*);
let s = *s*(|*ss-stack* := *tl* (*ss-stack s*)|);
let s = *s*(|*on-stack* := *on-stack s* $-$ {*u*}|);
RETURN s
}

definition *get-pending-impl s* $\equiv$ *do* {
ASSERT (*ss-stack s* $\neq$ []);
let (*u*,*Vs*) = *hd* (*ss-stack s*);
if Vs = {} *then*
RETURN (*u*,*None*,*s*)
else do {
$v \leftarrow$ *SPEC* ($\lambda v.\ v \in Vs$);
let Vs = *Vs* $-$ {*v*};
let s = *s*(| *ss-stack* := (*u*, *Vs*) # *tl* (*ss-stack s*) |);
RETURN (*u*, *Some v*, *s*)
}
}

definition *discover-impl u v s* $\equiv$ *do* {
ASSERT ($v \notin$ *on-stack s* $\wedge$ $v \notin$ *visited s*);
let s = *s*(|*ss-stack* := (*v*,*E*``{*v*}) # *ss-stack s*|);
let s = *s*(|*on-stack* := *insert v* (*on-stack s*)|);
let s = *s*(|*visited* := *insert v* (*visited s*)|);
RETURN s
}

definition *new-root-impl v0 s* $\equiv$ *do* {
ASSERT (*v0* $\notin$ *visited s*);
let s = *s*(|*ss-stack* := [(*v0*,*E*``{*v0*})]|);
let s = *s*(|*on-stack* := {*v0*}|);
let s = *s*(|*visited* := *insert v0* (*visited s*)|);
RETURN s
}

definition *gbs* ≡ ⦇
gbs-init = *init-impl*,
gbs-is-empty-stack = *is-empty-stack-impl* ,
gbs-new-root = *new-root-impl* ,
gbs-get-pending = *get-pending-impl* ,
gbs-finish = *finish-impl* ,
gbs-is-discovered = *is-discovered-impl* ,
gbs-is-finished = *is-finished-impl* ,
gbs-back-edge = ($\lambda u\ v\ s.\ RETURN\ s$) ,
gbs-cross-edge = ($\lambda u\ v\ s.\ RETURN\ s$) ,
gbs-discover = *discover-impl*
⦈

lemmas *gbs-simps*[*simp*, *DFS-code-unfold*] = *gen-basic-dfs-struct.simps*[*mk-record-simp*, *OF gbs-def*]

lemmas *impl-defs*[*DFS-code-unfold*]
= *init-impl-def is-empty-stack-impl-def new-root-impl-def*
get-pending-impl-def finish-impl-def is-discovered-impl-def
is-finished-impl-def discover-impl-def

end

Simple implementation of a DFS. This locale assumes a refinement of the parameters, and provides an implementation via a stack and a visited set.

locale *simple-impl-defs* =
a: *param-DFS-defs G param*
\+ *c*: *pre-simple-impl*
\+ *gen-param-dfs-refine-defs*
where *gbsi* = *c.gbs*
and *gbs* = *a.gbs*
and *upd-exti* = *simple-state.more-update*
and *upd-ext* = *state.more-update*
and *V0i* = *a.V0*
and *V0* = *a.V0*
begin

sublocale *tailrec-impl-defs G c.gds* **.**

definition *get-pending s* ≡ $\bigcup$(*set* (*map* ($\lambda(u,Vs).\ \{u\}\times Vs$) (*ss-stack s*)))
definition *get-stack s* ≡ *map fst* (*ss-stack s*)
definition *choose-pending*
:: $'v \Rightarrow\ 'v\ option \Rightarrow ('v,'d)$ *simple-state-scheme* ⇒ $('v,'d)$ *simple-state-scheme nres*
where [*DFS-code-unfold*]:
choose-pending u vo s ≡
case vo of

```
      None ⇒ RETURN s
    | Some v ⇒ do {
        ASSERT (ss-stack s ≠ []);
        let (u,Vs) = hd (ss-stack s);
        RETURN (s(| ss-stack := (u,Vs−{v})#tl (ss-stack s)|))
      }

  sublocale rec-impl-defs G c.gds get-pending get-stack choose-pending .
end


locale simple-impl =
  a: param-DFS
  + simple-impl-defs
  + param-refinement
    where gbsi = c.gbs
    and gbs = a.gbs
    and upd-exti = simple-state.more-update
    and upd-ext = state.more-update
    and V0i = a.V0
    and V0 = a.V0
    and V=Id
    and S = ⟨ES⟩simple-state-rel
begin

  lemma init-impl: (ei, e) ∈ ES ⟹
    c.init-impl ei ≤⇓(⟨ES⟩simple-state-rel) (RETURN (a.empty-state e))
    unfolding c.init-impl-def a.empty-state-def simple-state-rel-def
    by (auto)

  lemma new-root-impl:
    ⟦a.gen-dfs.pre-new-root v0 s;
      (v0i, v0)∈Id; (si, s) ∈ ⟨ES⟩simple-state-rel⟧
     ⟹ c.new-root-impl v0 si ≤⇓(⟨ES⟩simple-state-rel) (RETURN (a.new-root v0
s))
      unfolding simple-state-rel-def a.gen-dfs.pre-new-root-def c.new-root-impl-def
      by (auto simp add: a.pred-defs)

  lemma get-pending-impl:
    ⟦a.gen-dfs.pre-get-pending s; (si, s) ∈ ⟨ES⟩simple-state-rel⟧
      ⟹ c.get-pending-impl si
          ≤ ⇓ (Id ×r Id ×r ⟨ES⟩simple-state-rel) (a.get-pending s)
    apply (unfold a.get-pending-def c.get-pending-impl-def) []
    apply (refine-rcg bind-refine' Let-refine' IdI)
    apply (refine-dref-type)
    apply (auto
        simp: simple-state-rel-def a.gen-dfs.pre-defs a.pred-defs neq-Nil-conv
        dest: DFS-invar.stack-distinct
      )
```

done

lemma *inres-get-pending-None-conv*: *inres* (*a.get-pending s0*) (*v*, *None*, *s*)
$\longleftrightarrow$ *s=s0* $\wedge$ *v=hd* (*stack s0*) $\wedge$ *pending s0*``{*hd* (*stack s0*)} = {}
unfolding *a.get-pending-def*
by (*auto simp add*: *refine-pw-simps*)

lemma *inres-get-pending-Some-conv*: *inres* (*a.get-pending s0*) (*v*,*Some Vs*,*s*)
$\longleftrightarrow$ *v* = *hd* (*stack s*) $\wedge$ *s* = *s0*⦇*pending* := *pending s0* − {(*hd* (*stack s0*), *Vs*)}⦈
$\wedge$ (*hd* (*stack s0*), *Vs*) $\in$ *pending s0*
unfolding *a.get-pending-def*
by (*auto simp add*: *refine-pw-simps*)

lemma *finish-impl*:
⟦*a.gen-dfs.pre-finish v s0 s*; (*vi*, *v*)$\in$*Id*; (*si*, *s*) $\in$ ⟨*ES*⟩*simple-state-rel*⟧
$\Longrightarrow$ *c.finish-impl v si* $\le\Downarrow$(⟨*ES*⟩*simple-state-rel*) (*RETURN* (*a.finish v s*))
unfolding *simple-state-rel-def a.gen-dfs.pre-defs c.finish-impl-def*

apply (*clarsimp simp*: *inres-get-pending-None-conv*)
apply (*frule DFS-invar.stack-distinct*)
apply (*simp add*: *a.pred-defs map-tl*)
apply (*clarsimp simp*: *neq-Nil-conv*)
apply *blast*
done

lemma *cross-edge-impl*:
⟦*a.gen-dfs.pre-cross-edge u v s0 s*;
(*ui*, *u*)$\in$*Id*; (*vi*, *v*)$\in$*Id*; (*si*, *s*) $\in$ ⟨*ES*⟩*simple-state-rel*⟧
$\Longrightarrow$ (*si*, *a.cross-edge u v s*) $\in$ ⟨*ES*⟩*simple-state-rel*
unfolding *simple-state-rel-def a.gen-dfs.pre-defs*
by *simp*

lemma *back-edge-impl*:
⟦*a.gen-dfs.pre-back-edge u v s0 s*;
(*ui*, *u*)$\in$*Id*; (*vi*, *v*)$\in$*Id*; (*si*, *s*) $\in$ ⟨*ES*⟩*simple-state-rel*⟧
$\Longrightarrow$ (*si*, *a.back-edge u v s*) $\in$ ⟨*ES*⟩*simple-state-rel*
unfolding *simple-state-rel-def a.gen-dfs.pre-defs*
by *simp*

lemma *discover-impl*:
⟦*a.gen-dfs.pre-discover u v s0 s*; (*ui*, *u*)$\in$*Id*; (*vi*, *v*)$\in$*Id*; (*si*, *s*) $\in$ ⟨*ES*⟩*simple-state-rel*⟧
$\Longrightarrow$ *c.discover-impl ui vi si* $\le\Downarrow$(⟨*ES*⟩*simple-state-rel*) (*RETURN* (*a.discover u v s*))
unfolding *simple-state-rel-def a.gen-dfs.pre-defs c.discover-impl-def*
apply (*rule ASSERT-leI*)
apply (*clarsimp simp*: *inres-get-pending-Some-conv*)
apply (*frule DFS-invar.stack-discovered*)

apply (*auto simp*: *a.pred-defs*) []

apply (*clarsimp simp*: *inres-get-pending-Some-conv*)
apply (*frule DFS-invar.stack-discovered*)
apply (*frule DFS-invar.pending-ssE*)
apply (*clarsimp simp*: *a.pred-defs*)
apply *blast*
**done*

sublocale *gen-param-dfs-refine*
where *gbsi* = *c.gbs*
and *gbs* = *a.gbs*
and *upd-exti* = *simple-state.more-update*
and *upd-ext* = *state.more-update*
and *V0i* = *a.V0*
and *V0* = *a.V0*
and *V* = *Id*
and *S* = ⟨*ES*⟩*simple-state-rel*
apply *unfold-locales*
apply (*simp-all add*: *is-break-param*)

apply (*auto simp*: *a.is-discovered-def c.is-discovered-impl-def simple-state-rel-def*)
[]

apply (*auto simp*: *a.is-finished-def c.is-finished-impl-def simple-state-rel-def*) []

apply (*auto simp*: *a.is-empty-stack-def c.is-empty-stack-impl-def simple-state-rel-def*)
[]

apply (*refine-rcg init-impl*)

apply (*refine-rcg new-root-impl*, *simp-all*) []

apply (*refine-rcg get-pending-impl*) []

apply (*refine-rcg finish-impl*, *simp-all*) []

apply (*refine-rcg cross-edge-impl*, *simp-all*) []

apply (*refine-rcg back-edge-impl*, *simp-all*) []

apply (*refine-rcg discover-impl*, *simp-all*) []
done

Main outcome of this locale: The simple DFS-Algorithm, which is a general DFS scheme itself (and thus open to further refinements), and a refinement theorem that states correct refinement of the original DFS

lemma *simple-refine*[*refine*]: *c.gen-dfs* ≤ ⇓(⟨*ES*⟩*simple-state-rel*) *a.it-dfs*
using *gen-dfs-refine*

by *simp*

lemma *simple-refineT*[*refine*]: *c.gen-dfsT* $\leq$ $\Downarrow$($\langle ES\rangle$*simple-state-rel*) *a.it-dfsT*
using *gen-dfsT-refine*
by *simp*

Link with tail-recursive implementation

sublocale *tailrec-impl G c.gds*
apply *unfold-locales*
apply (*simp-all add*: *c.do-action-defs c.impl-defs*[*abs-def*])
apply (*auto simp*: *pw-leof-iff refine-pw-simps split*: *prod.splits*)
done

lemma *simple-tailrec-refine*[*refine*]: *tailrec-impl* $\leq$ $\Downarrow$($\langle ES\rangle$*simple-state-rel*) *a.it-dfs*
proof −
note *tailrec-impl* **also note** *simple-refine* **finally show** *?thesis* **.**
qed

lemma *simple-tailrecT-refine*[*refine*]: *tailrec-implT* $\leq$ $\Downarrow$($\langle ES\rangle$*simple-state-rel*) *a.it-dfsT*
proof −
note *tailrecT-impl* **also note** *simple-refineT* **finally show** *?thesis* **.**
qed

Link to recursive implementation

lemma *reachable-invar*:
assumes *c.gen-rwof s*
shows *set* (*map fst* (*ss-stack s*)) $\subseteq$ *visited s*
$\wedge$ *distinct* (*map fst* (*ss-stack s*))
using *assms*
apply (*induct rule*: *establish-rwof-invar*[*rotated* −*1*, *consumes 1*])
apply (*simp add*: *c.do-action-defs c.impl-defs*[*abs-def*])
apply (*refine-rcg refine-vcg*)
apply *simp*

unfolding *c.gen-step-def c.do-action-defs c.impl-defs*[*abs-def*] *c.gds-simps c.gbs-simps*
apply (*refine-rcg refine-vcg*)
apply *simp-all*
apply (*fastforce simp*: *neq-Nil-conv*) []
apply (*fastforce simp*: *neq-Nil-conv*) []
apply (*fastforce simp*: *neq-Nil-conv*) []
apply (*fastforce simp*: *neq-Nil-conv*) []
done

sublocale *rec-impl G c.gds get-pending get-stack choose-pending*
apply *unfold-locales*
unfolding *get-pending-def get-stack-def choose-pending-def*
apply (*simp-all add*: *c.do-action-defs c.impl-defs*[*abs-def*])
apply (*auto simp*: *pw-leof-iff refine-pw-simps pw-le-iff select-def*

split: *prod.split*) []
apply (*auto simp*: *pw-leof-iff refine-pw-simps pw-le-iff select-def*
split: *prod.split*) []

apply (*rule le-ASSERTI*)
apply (*unfold c.pre-defs*, *clarify*) []
apply (*frule reachable-invar*)

apply (*fastforce simp add*: *pw-leof-iff refine-pw-simps pw-le-iff neq-Nil-conv*
split: *prod.split option.split*) []

apply (*unfold c.pre-defs*, *clarify*) []
apply (*frule reachable-invar*)
apply (*auto simp*: *pw-leof-iff refine-pw-simps pw-le-iff select-def c.impl-defs neq-Nil-conv*
split: *prod.split option.split*) []

apply (*auto simp*: *pw-leof-iff refine-pw-simps pw-le-iff select-def neq-Nil-conv c.pre-defs c.impl-defs*
split: *prod.split if-split-asm*) []

apply (*auto simp*: *pw-leof-iff refine-pw-simps pw-le-iff split*: *prod.split*) []

apply (*auto simp*: *pw-leof-iff refine-pw-simps pw-le-iff split*: *prod.split*) []

apply (*auto simp*: *pw-leof-iff refine-pw-simps pw-le-iff split*: *prod.split*) []
done

lemma *simple-rec-refine*[*refine*]: *rec-impl* $\leq$ $\Downarrow$($\langle$*ES*$\rangle$*simple-state-rel*) *a.it-dfs*
proof −
note *rec-impl* **also note** *simple-refine* **finally show** *?thesis* **.**
qed

end

Autoref Setup

record (*'si*,*'nsi*)*simple-state-impl* =
ss-stack-impl :: *'si*
ss-on-stack-impl :: *'nsi*
ss-visited-impl :: *'nsi*

definition [*to-relAPP*]: *ss-impl-rel s-rel vis-rel erel* ≡
{((|*ss-stack-impl* = *si*, *ss-on-stack-impl* = *osi*, *ss-visited-impl* = *visi*, ... = *mi*|),
(|*ss-stack* = *s*, *on-stack* = *os*, *visited* = *vis*, ... = *m*|)) |
si osi visi mi s os vis m.
(*si*, *s*) ∈ *s-rel* ∧
(*osi*, *os*) ∈ *vis-rel* ∧
(*visi*, *vis*) ∈ *vis-rel* ∧
(*mi*, *m*) ∈ *erel*

}

consts
i-simple-state :: *interface* ⇒ *interface* ⇒ *interface* ⇒ *interface*

lemmas [*autoref-rel-intf*] = *REL-INTFI*[*of ss-impl-rel i-simple-state*]

term *simple-state-ext*

lemma [*autoref-rules*, *param*]:
fixes *s-rel ps-rel vis-rel erel*
defines $R \equiv \langle$*s-rel*,*vis-rel*,*erel*$\rangle$*ss-impl-rel*
shows
(*ss-stack-impl*, *ss-stack*) ∈ $R \rightarrow$ *s-rel*
(*ss-on-stack-impl*, *on-stack*) ∈ $R \rightarrow$ *vis-rel*
(*ss-visited-impl*, *visited*) ∈ $R \rightarrow$ *vis-rel*
(*simple-state-impl.more*, *simple-state.more*) ∈ $R \rightarrow$ *erel*
(*ss-stack-impl-update*, *ss-stack-update*) ∈ (*s-rel* → *s-rel*) → $R \rightarrow R$
(*ss-on-stack-impl-update*, *on-stack-update*) ∈ (*vis-rel* → *vis-rel*) → $R \rightarrow R$
(*ss-visited-impl-update*, *visited-update*) ∈ (*vis-rel* → *vis-rel*) → $R \rightarrow R$
(*simple-state-impl.more-update*, *simple-state.more-update*) ∈ (*erel* → *erel*) → R
→ R
(*simple-state-impl-ext*, *simple-state-ext*) ∈ *s-rel* → *vis-rel* → *vis-rel* → *erel* → R
unfolding *ss-impl-rel-def R-def*
apply *auto*
apply *parametricity+*
done

1.7.2 Simple state without on-stack

We can further refine the simple implementation and drop the on-stack set

record (*'si*,*'nsi*)*simple-state-nos-impl* =
ssnos-stack-impl :: *'si*
ssnos-visited-impl :: *'nsi*

definition [*to-relAPP*]: *ssnos-impl-rel s-rel vis-rel erel* ≡
{((|*ssnos-stack-impl* = *si*, *ssnos-visited-impl* = *visi*, ... = *mi*|),
(|*ss-stack* = *s*, *on-stack* = *os*, *visited* = *vis*, ... = *m*|)) |
si visi mi s os vis m.
(*si*, *s*) ∈ *s-rel* ∧
(*visi*, *vis*) ∈ *vis-rel* ∧
(*mi*, *m*) ∈ *erel*
}

lemmas [*autoref-rel-intf*] = *REL-INTFI*[*of ssnos-impl-rel i-simple-state*]

definition *op-nos-on-stack-update*
:: (- *set* ⇒ - *set*) ⇒ (-,-)*simple-state-scheme* ⇒ -
where *op-nos-on-stack-update* ≡ *on-stack-update*

context begin interpretation *autoref-syn* **.**
lemma [*autoref-op-pat-def*]: *op-nos-on-stack-update f s*
  $\equiv$ *OP* (*op-nos-on-stack-update f*)\$*s* **by** *simp*

end

lemmas *ssnos-unfolds* — To be unfolded before autoref when using *ssnos-impl-rel*
  = *op-nos-on-stack-update-def*[*symmetric*]

lemma [*autoref-rules*, *param*]:
  fixes *s-rel vis-rel erel*
  defines $R \equiv \langle$*s-rel*,*vis-rel*,*erel*$\rangle$*ssnos-impl-rel*
  shows
  (*ssnos-stack-impl*, *ss-stack*) $\in$ $R \to$ *s-rel*
  (*ssnos-visited-impl*, *visited*) $\in R \to$ *vis-rel*
  (*simple-state-nos-impl.more*, *simple-state.more*) $\in R \to$ *erel*
  (*ssnos-stack-impl-update*, *ss-stack-update*) $\in$ (*s-rel* $\to$ *s-rel*) $\to R \to R$
  ($\lambda x.\ x$, *op-nos-on-stack-update f*) $\in R \to R$
  (*ssnos-visited-impl-update*, *visited-update*) $\in$ (*vis-rel* $\to$ *vis-rel*) $\to R \to R$
  (*simple-state-nos-impl.more-update*, *simple-state.more-update*) $\in$ (*erel* $\to$ *erel*) $\to$ $R \to R$
  (λ*ns - ps vs. simple-state-nos-impl-ext ns ps vs*, *simple-state-ext*)
    $\in$ *s-rel* $\to$ *ANY-rel* $\to$ *vis-rel* $\to$ *erel* $\to R$
  unfolding *ssnos-impl-rel-def R-def op-nos-on-stack-update-def*
  apply *auto*
  apply *parametricity+*
  done

1.7.3 Simple state without stack and on-stack

Even further refinement yields an implementation without a stack. Note that this only works for structural implementations that provide their own stack (e.g., recursive)!

record ('*si*,'*nsi*)*simple-state-ns-impl* =
  ssns-visited-impl :: '*nsi*

definition [*to-relAPP*]: *ssns-impl-rel* (*R*::('*a*$\times$'*b*) *set*) *vis-rel erel* $\equiv$
  {(⦇*ssns-visited-impl* = *visi*, ... = *mi*⦈,
    ⦇*ss-stack* = *s*, *on-stack* = *os*, *visited* = *vis*, ... = *m*⦈) |
    visi mi s os vis m.
    (*visi*, *vis*) $\in$ *vis-rel* $\wedge$
    (*mi*, *m*) $\in$ *erel*
  }

lemmas [*autoref-rel-intf*] = *REL-INTFI*[*of ssns-impl-rel i-simple-state*]

definition *op-ns-on-stack-update*
  :: (- *set* $\Rightarrow$ - *set*) $\Rightarrow$ (-,-)*simple-state-scheme* $\Rightarrow$ -

where *op-ns-on-stack-update* ≡ *on-stack-update*

definition *op-ns-stack-update*
:: (- *list* ⇒ - *list*) ⇒ (-,-)*simple-state-scheme* ⇒ -
where *op-ns-stack-update* ≡ *ss-stack-update*

context begin interpretation *autoref-syn* **.**
lemma [*autoref-op-pat-def*]: *op-ns-on-stack-update f s*
≡ *OP* (*op-ns-on-stack-update f*)$*s* **by** *simp*

lemma [*autoref-op-pat-def*]: *op-ns-stack-update f s*
≡ *OP* (*op-ns-stack-update f*)$*s* **by** *simp*

end

context *simple-impl-defs* **begin**
thm *choose-pending-def*[*unfolded op-ns-stack-update-def*[*symmetric*], *no-vars*]

lemma *choose-pending-ns-unfold*: *choose-pending u vo s* = (
case vo of None ⇒ *RETURN s*
| *Some v* ⇒ *do* {
- ← *ASSERT* (*ss-stack s* ≠ []);
RETURN
(*op-ns-stack-update*
(*let*
(*u*, *Vs*) = *hd* (*ss-stack s*)
in (λ-. (*u*, *Vs* − {*v*}) # *tl* (*ss-stack s*))
)
s
)
})
unfolding *choose-pending-def op-ns-stack-update-def*
by (*auto split*: *option.split prod.split*)

lemmas *ssns-unfolds* — To be unfolded before autoref when using *ssns-impl-rel*. Attention: This lemma conflicts with the standard unfolding lemma in *DFS-code-unfold*, so has to be placed first in an unfold-statement!
= *op-ns-on-stack-update-def*[*symmetric*] *op-ns-stack-update-def*[*symmetric*]
choose-pending-ns-unfold

end

lemma [*autoref-rules*, *param*]:
fixes *s-rel vis-rel erel ANY-rel*
defines *R* ≡ ⟨*ANY-rel*,*vis-rel*,*erel*⟩*ssns-impl-rel*
shows
(*ssns-visited-impl*, *visited*) ∈ *R* → *vis-rel*
(*simple-state-ns-impl.more*, *simple-state.more*) ∈ *R* → *erel*

$\bigwedge f.\ (\lambda x.\ x,\ op\text{-}ns\text{-}stack\text{-}update\ f) \in R \to R$
$\bigwedge f.\ (\lambda x.\ x,\ op\text{-}ns\text{-}on\text{-}stack\text{-}update\ f) \in R \to R$
$(ssns\text{-}visited\text{-}impl\text{-}update,\ visited\text{-}update) \in (vis\text{-}rel \to vis\text{-}rel) \to R \to R$
$(simple\text{-}state\text{-}ns\text{-}impl.more\text{-}update,\ simple\text{-}state.more\text{-}update) \in (erel \to erel) \to R \to R$
$(\lambda\text{-}\ \text{-}\ ps\ vs.\ simple\text{-}state\text{-}ns\text{-}impl\text{-}ext\ ps\ vs,\ simple\text{-}state\text{-}ext)$
$\in ANY1\text{-}rel \to ANY2\text{-}rel \to vis\text{-}rel \to erel \to R$
unfolding *ssns-impl-rel-def R-def op-ns-on-stack-update-def op-ns-stack-update-def*
apply *auto*
apply *parametricity+*
done

lemma [*refine-transfer-post-simp*]:
$\bigwedge a\ m.\ a(\!|simple\text{-}state\text{-}nos\text{-}impl.more := m{::}unit|\!) = a$
$\bigwedge a\ m.\ a(\!|simple\text{-}state\text{-}impl.more := m{::}unit|\!) = a$
$\bigwedge a\ m.\ a(\!|simple\text{-}state\text{-}ns\text{-}impl.more := m{::}unit|\!) = a$
by *auto*

end

1.8 Restricting Nodes by Pre-Initializing Visited Set

theory *Restr-Impl*
imports *Simple-Impl*
begin

Implementation of node and edge restriction via pre-initialized visited set.

We now further refine the simple implementation in case that the graph has the form $G'=(rel\text{-}restrict\ E\ R,\ V0-R)$ for some *fb-graph* $G=(E,V0)$. If, additionally, the parameterization is not "too sensitive" to the visited set, we can pre-initialize the visited set with R, and use the $V0$ and E of G. This may be a more efficient implementation than explicitely restricting $V0$ and E, as it saves additional membership queries in R on each successor function call.

Moreover, in applications where the restriction is updated between multiple calls, we can use one linearly accessed restriction set.

definition *restr-rel* $R \equiv \{\ (s,s').$
$(ss\text{-}stack\ s,\ ss\text{-}stack\ s')\in\langle Id \times_r \{(U,U').\ U-R = U'\}\rangle list\text{-}rel$
$\wedge\ on\text{-}stack\ s = on\text{-}stack\ s'$
$\wedge\ visited\ s = visited\ s' \cup R \wedge visited\ s' \cap R = \{\}$
$\wedge\ simple\text{-}state.more\ s = simple\text{-}state.more\ s'\ \}$

lemma *restr-rel-simps*:
assumes $(s,s')\in restr\text{-}rel\ R$

shows *visited s* = *visited s'* ∪ *R*
and *simple-state.more s* = *simple-state.more s'*
using *assms* **unfolding** *restr-rel-def* **by** *auto*

lemma
assumes $(s,s')\in$*restr-rel R*
shows *restr-rel-stackD*: (*ss-stack s*, *ss-stack s'*) ∈ ⟨*Id* $\times_r$ {(*U*,*U'*). *U*−*R* = *U'*}⟩*list-rel*
and *restr-rel-vis-djD*: *visited s'* ∩ *R* = {}
using *assms* **unfolding** *restr-rel-def* **by** *auto*

context fixes *R* :: *'v set* **begin**
definition [*to-relAPP*]: *restr-simple-state-rel ES* ≡ { (*s*,*s'*) .
(*ss-stack s*, *map* (λ*u*. (*u*,*pending s'* ‘‘ {*u*})) (*stack s'*))
∈ ⟨*Id* $\times_r$ {(*U*,*U'*). *U*−*R* = *U'*}⟩*list-rel* ∧
on-stack s = *set* (*stack s'*) ∧
visited s = *dom* (*discovered s'*) ∪ *R* ∧ *dom* (*discovered s'*) ∩ *R* = {} ∧
dom (*finished s'*) = *dom* (*discovered s'*) − *set* (*stack s'*) ∧
set (*stack s'*) ⊆ *dom* (*discovered s'*) ∧
(*simple-state.more s*, *state.more s'*) ∈ *ES*
}
end

lemma *restr-simple-state-rel-combine*:
⟨*ES*⟩*restr-simple-state-rel R* = *restr-rel R O* ⟨*ES*⟩*simple-state-rel*
unfolding *restr-simple-state-rel-def*
apply (*intro equalityI subsetI*)
apply *clarify*
apply (*rule relcompI*[*OF - simple-state-relI*], *auto simp*: *restr-rel-def*) []

apply (*auto simp*: *restr-rel-def simple-state-rel-def*) []
done

Locale that assumes a simple implementation, makes some additional assumptions on the parameterization (intuitively, that it is not too sensitive to adding nodes from R to the visited set), and then provides a new implementation with pre-initialized visited set.

locale *restricted-impl-defs* =
graph-defs G +
a: *simple-impl-defs graph-restrict G R*
for *G* :: (*'v*, *'more*) *graph-rec-scheme*
and *R*
begin
sublocale *pre-simple-impl G* **.**

abbreviation *rel* ≡ *restr-rel R*

definition *gbs'* ≡ *gbs* (|
gbs-init := λ*e*. *RETURN*

(| *ss-stack*=[], *on-stack*={}, *visited* = *R*, ...=*e* |) |)

lemmas *gbs′-simps*[*simp*, *DFS-code-unfold*]
= *gen-basic-dfs-struct.simps*[*mk-record-simp*, *OF gbs′-def*[*unfolded gbs-simps*]]

sublocale *gen-param-dfs-defs gbs′ parami simple-state.more-update V0* **.**

sublocale *tailrec-impl-defs G gds* **.**
end

locale *restricted-impl* =
fb-graph +
a: *simple-impl graph-restrict G R* +
restricted-impl-defs +

assumes [*simp*]: *on-cross-edge parami* = (λ*u v s*. *RETURN* (*simple-state.more* *s*))
assumes [*simp*]: *on-back-edge parami* = (λ*u v s*. *RETURN* (*simple-state.more* *s*))

assumes *is-break-refine*:
⟦ (*s*,*s′*)∈*restr-rel R* ⟧
⟹ *is-break parami s* ⟷ *is-break parami s′*

assumes *on-new-root-refine*:
⟦ (*s*,*s′*)∈*restr-rel R* ⟧
⟹ *on-new-root parami v0 s* ≤ *on-new-root parami v0 s′*

assumes *on-finish-refine*:
⟦ (*s*,*s′*)∈*restr-rel R* ⟧
⟹ *on-finish parami u s* ≤ *on-finish parami u s′*

assumes *on-discover-refine*:
⟦ (*s*,*s′*)∈*restr-rel R* ⟧
⟹ *on-discover parami u v s* ≤ *on-discover parami u v s′*

begin

lemmas *rel-def* = *restr-rel-def*[**where** *R*=*R*]
sublocale *gen-param-dfs gbs′ parami simple-state.more-update V0* **.**

lemma *is-break-param′*[*param*]: (*is-break parami*, *is-break parami*)∈*rel* → *bool-rel*
using *is-break-refine* **unfolding** *rel-def* **by** *auto*

lemma *do-init-refine*[*refine*]: *do-init* ≤ ⇓ *rel* (*a.c.do-init*)
  unfolding *do-action-defs a.c.do-action-defs*
  apply (*simp add*: *rel-def a.c.init-impl-def*)
  apply *refine-rcg*
  apply *simp*
  done

lemma *gen-cond-param*: (*gen-cond*,*a.c.gen-cond*)∈*rel* → *bool-rel*
  apply (*clarsimp simp del*: *graph-restrict-simps*)
  apply (*frule is-break-param′*[*param-fo*])
  unfolding *gen-cond-def a.c.gen-cond-def rel-def*
  apply *simp*
  unfolding *a.c.is-discovered-impl-def a.c.is-empty-stack-impl-def*
  by *auto*

lemma *cross-back-id*[*simp*]:
  do-cross-edge u v s = *RETURN s*
  do-back-edge u v s = *RETURN s*
  a.c.do-cross-edge u v s = *RETURN s*
  a.c.do-back-edge u v s = *RETURN s*
  unfolding *do-action-defs a.c.do-action-defs*
  by *simp-all*

lemma *pred-rel-simps*:
  assumes (*s*,*s′*)∈*rel*
  shows *a.c.is-discovered-impl u s* ⟷ *a.c.is-discovered-impl u s′* ∨ *u*∈*R*
  and *a.c.is-empty-stack-impl s* ⟷ *a.c.is-empty-stack-impl s′*
  using *assms*
  unfolding *a.c.is-discovered-impl-def a.c.is-empty-stack-impl-def*
  unfolding *rel-def*
  by *auto*

lemma *no-pending-refine*:
  assumes (*s*,*s′*)∈*rel* ¬*a.c.is-empty-stack-impl s′*
  shows (*hd* (*ss-stack s*) = (*u*,{})) ⟹ *hd* (*ss-stack s′*) = (*u*,{})
  using *assms*
  unfolding *a.c.is-empty-stack-impl-def rel-def*
  apply (*cases ss-stack s′*, *simp*)
  apply (*auto elim*: *list-relE*)
  done

lemma *do-new-root-refine*[*refine*]:
  ⟦ (*v0i*,*v0*)∈*Id*; (*si*,*s*)∈*rel*; *v0*∉*R* ⟧
    ⟹ *do-new-root v0i si* ≤ ⇓ *rel* (*a.c.do-new-root v0 s*)
  unfolding *do-action-defs a.c.do-action-defs*
  apply *refine-rcg*
  apply (*rule intro-prgR*[**where** *R*=*rel*])
  apply (*simp add*: *a.c.new-root-impl-def new-root-impl-def*)

apply (*refine-rcg*,*auto simp*: *rel-def rel-restrict-def*) []

apply (*rule intro-prgR*[**where** R=*Id*])
apply (*simp add*: *on-new-root-refine*)
apply (*simp add*: *rel-def*)
done

lemma *do-finish-refine*[*refine*]:
$[\![(s, s') \in rel; (u,u') \in Id]\!]$
$\Longrightarrow$ *do-finish u s* $\leq \Downarrow$ *rel* (*a.c.do-finish u′ s′*)
unfolding *do-action-defs a.c.do-action-defs*
apply *refine-rcg*
apply (*rule intro-prgR*[**where** R=*rel*])
apply (*simp add*: *finish-impl-def is-empty-stack-impl-def*)
apply (*refine-rcg*,*auto simp*: *rel-def rel-restrict-def*) []
apply *parametricity*

apply (*rule intro-prgR*[**where** R=*Id*])
apply (*simp add*: *on-finish-refine*)
apply (*simp add*: *rel-def*)
done

lemma *aux-cnv-pending*:
$[\![$ $(s, s') \in rel$;
$\neg$ *is-empty-stack-impl s*; $vs \in Vs$; $vs \notin R$;
hd (*ss-stack s*) = (u,*Vs*) $]\!] \Longrightarrow$
hd (*ss-stack s′*) = (u,*insert vs* (*Vs*−R))

unfolding *rel-def is-empty-stack-impl-def*
apply (*cases ss-stack s′*, *simp*)
apply (*auto elim*: *list-relE*)
done

lemma *get-pending-refine*:
assumes $(s, s') \in rel$ *gen-cond s* $\neg$ *is-empty-stack-impl s*
shows
get-pending-impl s $\leq$ (*sup*
($\Downarrow$(*Id* $\times_r$ $\langle Id \rangle$*option-rel* $\times_r$ *rel*) (*inf*
(*get-pending-impl s′*)
(*SPEC* (λ(-,*Vs*,-). *case Vs of None* $\Rightarrow$ *True* | *Some v* $\Rightarrow$ $v \notin R$))))
($\Downarrow$(*Id* $\times_r$ $\langle Id \rangle$*option-rel* $\times_r$ *rel*) (
SPEC (λ(u,*Vs*,s''). $\exists\, v$. *Vs*=*Some v* $\wedge$ $v \in R$ $\wedge$ $s''=s'$)
)))
proof −
from *assms* **have**
[*simp*]: *ss-stack s′* $\neq$ []
and [*simp*]: *ss-stack s* $\neq$ []
unfolding *rel-def impl-defs*

apply (*auto*)
done

from *assms* **show** *?thesis*
unfolding *get-pending-impl-def*
apply (*subst Let-def*, *subst Let-def*)
apply (*rule ASSERT-leI*)
apply (*auto simp*: *impl-defs gen-cond-def rel-def*) []

apply (*split prod.split*, *intro allI impI*)
apply (*rule lhs-step-If*)

apply (*rule le-supI1*)
apply (*simp add*: *pred-rel-simps no-pending-refine restr-rel-simps RETURN-RES-refine-iff*)

apply (*rule lhs-step-bind*, *simp*)
apply (*simp split del*: *if-split*)
apply (*rename-tac v*)
apply (*case-tac v*∈*R*)

apply (*rule le-supI2*)
apply (*rule RETURN-SPEC-refine*)
apply (*auto simp*: *rel-def is-empty-stack-impl-def neq-Nil-conv*) []
apply (*cases ss-stack s*′, *simp*) **apply** (*auto elim*!: *list-relE*) []

apply (*rule le-supI1*)
apply (*frule* (*4*) *aux-cnv-pending*)
apply (*simp add*: *no-pending-refine pred-rel-simps memb-imp-not-empty*)
apply (*subst nofail-inf-serialize*,
(*simp-all add*: *refine-pw-simps split*: *prod.splits*) [*2*])
apply *simp*
apply (*rule rhs-step-bind-RES*, *blast*)
apply (*simp add*: *rel-def is-empty-stack-impl-def*) []
apply (*cases ss-stack s*′, *simp*)
apply (*auto elim*: *list-relE*) []
done
qed

lemma *do-discover-refine*[*refine*]:
$\llbracket$ $(s, s') \in rel$; $(u,u')\in Id$; $(v,v')\in Id$; $v' \notin R$ $\rrbracket$
$\Longrightarrow$ *do-discover u v s* $\leq \Downarrow$ *rel* (*a.c.do-discover u*′ *v*′ *s*′)
unfolding *do-action-defs a.c.do-action-defs*
apply *refine-rcg*
apply (*rule intro-prgR*[**where** *R*=*rel*])
apply (*simp add*: *discover-impl-def a.c.discover-impl-def*)
apply (*refine-rcg*,*auto simp*: *rel-def rel-restrict-def*) []

apply (*rule intro-prgR*[**where** *R*=*Id*])
apply (*simp add*: *on-discover-refine*)

apply (*auto simp*: *rel-def*) []
done

lemma *aux-R-node-discovered*: $\llbracket(s,s')\in rel;\ v\in R\rrbracket \Longrightarrow$ *is-discovered-impl v s*
by (*auto simp*: *pred-rel-simps*)

lemma *re-refine-aux*: *gen-dfs* $\leq \Downarrow$*rel a.c.gen-dfs*
unfolding *a.c.gen-dfs-def gen-dfs-def*
apply (*simp del*: *graph-restrict-simps*)

apply (*rule bind-refine*)
apply (*refine-rcg*)
apply (*erule WHILE-invisible-refine*)

apply (*frule gen-cond-param*[*param-fo*], *fastforce*)

apply (*frule* (*1*) *gen-cond-param*[*param-fo*, *THEN IdD*, *THEN iffD1*])
apply (*simp del*: *graph-restrict-simps*)
unfolding *gen-step-def*
apply (*simp del*: *graph-restrict-simps cong*: *if-cong option.case-cong split del*: *if-split*)
apply (*rule lhs-step-If*)

apply (*frule* (*1*) *pred-rel-simps*[*THEN iffD1*])
apply (*rule le-supI1*)
apply (*simp add*: *a.c.gen-step-def del*: *graph-restrict-simps*)
apply *refine-rcg*
apply (*auto simp*: *pred-rel-simps*) [*2*]

apply (*frule* (*1*) *pred-rel-simps*[*THEN Not-eq-iff*[*symmetric*, *THEN iffD1*], *THEN iffD1*])
thm *order-trans*[*OF bind-mono*(*1*)[*OF get-pending-refine order-refl*]]
apply (*rule order-trans*[*OF bind-mono*(*1*)[*OF get-pending-refine order-refl*]])
apply *assumption+*

apply (*unfold bind-distrib-sup1*)
apply (*rule sup-least*)

apply (*rule le-supI1*)
apply (*simp add*: *a.c.gen-step-def del*: *graph-restrict-simps cong*: *option.case-cong if-cong*)
apply (*rule bind-refine'*[*OF conc-fun-mono*[*THEN monoD*]], *simp*)

apply (*clarsimp simp*: *refine-pw-simps*)
apply (*refine-rcg*, *refine-dref-type*, *simp-all add*: *pred-rel-simps*) []

apply (*rule le-supI2*)
apply (*rule RETURN-as-SPEC-refine*)
apply (*simp add*: *conc-fun-SPEC*)
apply (*refine-rcg refine-vcg bind-refine′*, *simp-all*) []
apply (*clarsimp*)
apply (*frule* (*1*) *aux-R-node-discovered*, *blast*)
done

theorem *re-refine-aux2*: *gen-dfs* $\leq\Downarrow$(*rel O* ⟨*ES*⟩*simple-state-rel*) *a.a.it-dfs*
proof −
note *re-refine-aux*
also note *a.gen-dfs-refine*
finally show *?thesis* **by** (*simp add*: *conc-fun-chain del*: *graph-restrict-simps*)
qed

theorem *re-refine*: *gen-dfs* $\leq\Downarrow$(⟨*ES*⟩*restr-simple-state-rel R*) *a.a.it-dfs*
unfolding *restr-simple-state-rel-combine*
by (*rule re-refine-aux2*)

sublocale *tailrec-impl G gds*
apply *unfold-locales*
apply (*simp-all add*: *do-action-defs impl-defs*[*abs-def*])
apply (*auto simp*: *pw-leof-iff refine-pw-simps split*: *prod.split*)
done

lemma *tailrec-refine*: *tailrec-impl* $\leq$ $\Downarrow$(⟨*ES*⟩*restr-simple-state-rel R*) *a.a.it-dfs*
proof −
note *tailrec-impl* **also note** *re-refine* **finally show** *?thesis* **.**
qed

end

end

1.9 Basic DFS Framework

theory *DFS-Framework*
imports
Param-DFS
Invars/DFS-Invars-Basic
Impl/Structural/Tailrec-Impl
Impl/Structural/Rec-Impl

Impl/Data/Simple-Impl
Impl/Data/Restr-Impl
begin

Entry point for the DFS framework, with basic invariants, tail-recursive and recursive implementation, and basic state data structures.

end

Chapter 2

Examples

This chapter contains examples of using the DFS Framework. Most examples are re-usable algorithms, that can easily be integrated into other (refinement framework based) developments.

The cyclicity checker example contains a detailed description of how to use the DFS framework, and can be used as a guideline for own DFS-framework based developments.

2.1 Simple Cyclicity Checker

theory *Cyc-Check*
imports *../DFS-Framework*
  CAVA-Automata.Digraph-Impl
  ../Misc/Impl-Rev-Array-Stack
begin

This example presents a simple cyclicity checker: Given a directed graph with start nodes, decide whether it's reachable part is cyclic.

The example tries to be a tutorial on using the DFS framework, explaining every required step in detail.

We define two versions of the algorithm, a partial correct one assuming only a finitely branching graph, and a total correct one assuming finitely many reachable nodes.

2.1.1 Framework Instantiation

Define a state, based on the DFS-state. In our case, we just add a break-flag.

record *'v cycc-state = 'v state +*
  break :: bool

Some utility lemmas for the simplifier, to handle idiosyncrasies of the record package.

lemma *break-more-cong*: *state.more s* = *state.more s'* $\Longrightarrow$ *break s* = *break s'*
by (*cases s*, *cases s'*, *simp*)

lemma [*simp*]: *s*⦇ *state.more* := ⦇ *break* = *foo* ⦈ ⦈ = *s* ⦇ *break* := *foo* ⦈
by (*cases s*) *simp*

Define the parameterization. We start at a default parameterization, where all operations default to skip, and just add the operations we are interested in: Initially, the break flag is false, it is set if we encounter a back-edge, and once set, the algorithm shall terminate immediately.

definition *cycc-params* :: (*'v*,*unit cycc-state-ext*) *parameterization*
where *cycc-params* ≡ *dflt-parametrization state.more*
(*RETURN* ⦇ *break* = *False* ⦈) ⦇
on-back-edge := λ- - -. *RETURN* ⦇ *break* = *True* ⦈,
is-break := *break* ⦈
lemmas *cycc-params-simp*[*simp*] =
gen-parameterization.simps[*mk-record-simp*, *OF cycc-params-def*[*simplified*]]

interpretation *cycc*: *param-DFS-defs* **where** *param*=*cycc-params* **for** *G* **.**

We now can define our cyclicity checker. The partially correct version asserts a finitely branching graph:

definition *cyc-checker G* ≡ *do* {
ASSERT (*fb-graph G*);
s ← *cycc.it-dfs TYPE*(*'a*) *G*;
RETURN (*break s*)
}

The total correct variant asserts finitely many reachable nodes.

definition *cyc-checkerT G* ≡ *do* {
ASSERT (*graph G* ∧ *finite* (*graph-defs.reachable G*));
s ← *cycc.it-dfsT TYPE*(*'a*) *G*;
RETURN (*break s*)
}

Next, we define a locale for the cyclicity checker's precondition and invariant, by specializing the *param-DFS* locale.

locale *cycc* = *param-DFS G cycc-params* **for** *G* :: (*'v*, *'more*) *graph-rec-scheme*
begin

We can easily show that our parametrization does not fail, thus we also get the DFS-locale, which gives us the correctness theorem for the DFS-scheme

sublocale *DFS G cycc-params*
apply *unfold-locales*
apply (*simp-all add*: *cycc-params-def*)
done

thm *it-dfs-correct* — Partial correctness
thm *it-dfsT-correct* — Total correctness if set of reachable states is finite
end

lemma *cyccI*:
assumes *fb-graph G*
shows *cycc G*
proof −
interpret *fb-graph G* **by** *fact*
show *?thesis* **by** *unfold-locales*
qed

lemma *cyccI'*:
assumes *graph G*
and *FR*: *finite* (*graph-defs.reachable G*)
shows *cycc G*
proof −
interpret *graph G* **by** *fact*
from *FR* **interpret** *fb-graph G* **by** (*rule fb-graphI-fr*)
show *?thesis* **by** *unfold-locales*
qed

Next, we specialize the *DFS-invar* locale to our parameterization. This locale contains all proven invariants. When proving new invariants, this locale is available as assumption, thus allowing us to re-use already proven invariants.

locale *cycc-invar* = *DFS-invar* **where** *param* = *cycc-params* + *cycc*

The lemmas to establish invariants only provide the *DFS-invar* locale. This lemma is used to convert it into the *cycc-invar* locale.

lemma *cycc-invar-eq*[*simp*]:
shows *DFS-invar G cycc-params s* ⟷ *cycc-invar G s*
proof
assume *DFS-invar G cycc-params s*
interpret *DFS-invar G cycc-params s* **by** *fact*
show *cycc-invar G s* **by** *unfold-locales*
next
assume *cycc-invar G s*
then interpret *cycc-invar G s* **.**
show *DFS-invar G cycc-params s* **by** *unfold-locales*
qed

2.1.2 Correctness Proof

We now enter the *cycc-invar* locale, and show correctness of our cyclicity checker.

context *cycc-invar* **begin**

We show that we break if and only if there are back edges. This is straightforward from our parameterization, and we can use the *establish-invarI* rule provided by the DFS framework.

We use this example to illustrate the general proof scheme:

lemma (**in** *cycc*) *i-brk-eq-back*: *is-invar* ($\lambda s.$ *break s* $\longleftrightarrow$ *back-edges s* $\neq$ {})
proof (*induct rule*: *establish-invarI*)

The $\llbracket$*on-init cycc-params* $\le_n$ *SPEC* ($\lambda x.$ *?I* (*empty-state x*)); $\bigwedge s\ s'\ v0.$ $\llbracket$*DFS-invar G cycc-params s*; *?I s*; *cond s*; $\neg$ *is-break cycc-params s*; *stack s* = []; *v0* $\in$ *V0*; *v0* $\notin$ *dom* (*discovered s*); *s'* = *new-root v0 s*$\rrbracket$ $\Longrightarrow$ *on-new-root cycc-params v0 s'* $\le_n$ *SPEC* ($\lambda x.$ *DFS-invar G cycc-params* (*s'*(|*state.more* := *x*|)) $\longrightarrow$ *?I* (*s'*(|*state.more* := *x*|))); $\bigwedge s\ s'\ u.$ $\llbracket$*DFS-invar G cycc-params s*; *?I s*; *cond s*; $\neg$ *is-break cycc-params s*; *stack s* $\neq$ []; *u* = *hd* (*stack s*); *pending s* `` {*u*} = {}; *s'* = *finish u s*$\rrbracket$ $\Longrightarrow$ *on-finish cycc-params u s'* $\le_n$ *SPEC* ($\lambda x.$ *DFS-invar G cycc-params* (*s'*(|*state.more* := *x*|)) $\longrightarrow$ *?I* (*s'*(|*state.more* := *x*|))); $\bigwedge s\ s'\ u\ v.$ $\llbracket$*DFS-invar G cycc-params s*; *?I s*; *cond s*; $\neg$ *is-break cycc-params s*; *stack s* $\neq$ []; (*u*, *v*) $\in$ *pending s*; *u* = *hd* (*stack s*); *v* $\in$ *dom* (*discovered s*); *v* $\in$ *dom* (*finished s*); *s'* = *cross-edge u v* (*s*(|*pending* := *pending s* $-$ {(*u*, *v*)}|))$\rrbracket$ $\Longrightarrow$ *on-cross-edge cycc-params u v s'* $\le_n$ *SPEC* ($\lambda x.$ *DFS-invar G cycc-params* (*s'*(|*state.more* := *x*|)) $\longrightarrow$ *?I* (*s'*(|*state.more* := *x*|))); $\bigwedge s\ s'\ u\ v.$ $\llbracket$*DFS-invar G cycc-params s*; *?I s*; *cond s*; $\neg$ *is-break cycc-params s*; *stack s* $\neq$ []; (*u*, *v*) $\in$ *pending s*; *u* = *hd* (*stack s*); *v* $\in$ *dom* (*discovered s*); *v* $\notin$ *dom* (*finished s*); *s'* = *back-edge u v* (*s*(|*pending* := *pending s* $-$ {(*u*, *v*)}|))$\rrbracket$ $\Longrightarrow$ *on-back-edge cycc-params u v s'* $\le_n$ *SPEC* ($\lambda x.$ *DFS-invar G cycc-params* (*s'*(|*state.more* := *x*|)) $\longrightarrow$ *?I* (*s'*(|*state.more* := *x*|))); $\bigwedge s\ s'\ u\ v.$ $\llbracket$*DFS-invar G cycc-params s*; *?I s*; *cond s*; $\neg$ *is-break cycc-params s*; *stack s* $\neq$ []; (*u*, *v*) $\in$ *pending s*; *u* = *hd* (*stack s*); *v* $\notin$ *dom* (*discovered s*); *s'* = *discover u v* (*s*(|*pending* := *pending s* $-$ {(*u*, *v*)}|))$\rrbracket$ $\Longrightarrow$ *on-discover cycc-params u v s'* $\le_n$ *SPEC* ($\lambda x.$ *DFS-invar G cycc-params* (*s'*(|*state.more* := *x*|)) $\longrightarrow$ *?I* (*s'*(|*state.more* := *x*|)))$\rrbracket$ $\Longrightarrow$ *is-invar ?I* rule is used with the induction method, and yields cases

print-cases

Our parameterization has only hooked into initialization and back-edges, so only these two cases are non-trivial

case *init* **thus** *?case* **by** (*simp add*: *empty-state-def*)
next
case (*back-edge s s' u v*)

For proving invariant preservation, we may assume that the invariant holds on the previous state. Interpreting the invariant locale makes available all invariants ever proved into this locale (i.e., the invariants from all loaded libraries, and the ones you proved yourself.).

then interpret *cycc-invar G s* **by** *simp*

However, here we do not need them:

from *back-edge* **show** *?case* **by** *simp*
qed (*simp-all cong*: *break-more-cong*)

For technical reasons, invariants are proved in the basic locale, and then transferred to the invariant locale:

lemmas *brk-eq-back* = *i-brk-eq-back*[*THEN make-invar-thm*]

The above lemma is simple enough to have a short apply-style proof:

lemma (**in** *cycc*) *i-brk-eq-back-short-proof*:
  is-invar (λ*s. break s* $\longleftrightarrow$ *back-edges s* $\neq$ {})
  apply (*induct rule*: *establish-invarI*)
  apply (*simp-all add*: *cond-def cong*: *break-more-cong*)
  apply (*simp add*: *empty-state-def*)
  done

Now, when we know that the break flag indicates back-edges, we can easily prove correctness, using a lemma from the invariant library:

thm *cycle-iff-back-edges*
lemma *cycc-correct-aux*:
  assumes *NC*: $\neg$*cond s*
  shows *break s* $\longleftrightarrow$ $\neg$*acyclic* (*E* $\cap$ *reachable* $\times$ *UNIV*)
proof (*cases break s*, *simp-all*)
  assume *break s*
  with *brk-eq-back* **have** *back-edges s* $\neq$ {} **by** *simp*
  with *cycle-iff-back-edges* **have** $\neg$*acyclic* (*edges s*) **by** *simp*
  with *acyclic-subset*[*OF - edges-ss-reachable-edges*]
  show $\neg$*acyclic* (*E* $\cap$ *reachable* $\times$ *UNIV*) **by** *blast*
next
  assume *A*: $\neg$*break s*
  from *A brk-eq-back* **have** *back-edges s* = {} **by** *simp*
  with *cycle-iff-back-edges* **have** *acyclic* (*edges s*) **by** *simp*
  also from *A nc-edges-covered*[*OF NC*] **have** *edges s* = *E* $\cap$ *reachable* $\times$ *UNIV*

    by *simp*
  finally show *acyclic* (*E* $\cap$ *reachable* $\times$ *UNIV*) **.**
qed

Again, we have a short two-line proof:

lemma *cycc-correct-aux-short-proof*:
  assumes *NC*: $\neg$*cond s*
  shows *break s* $\longleftrightarrow$ $\neg$*acyclic* (*E* $\cap$ *reachable* $\times$ *UNIV*)
  using *nc-edges-covered*[*OF NC*] *brk-eq-back cycle-iff-back-edges*
  by (*auto dest*: *acyclic-subset*[*OF - edges-ss-reachable-edges*])

end

Finally, we define a specification for cyclicity checking, and prove that our cyclicity checker satisfies the specification:

definition *cyc-checker-spec G* $\equiv$ *do* {
  ASSERT (*fb-graph G*);
  SPEC (λ*r. r* $\longleftrightarrow$ $\neg$*acyclic* (*g-E G* $\cap$ ((*g-E G*)* `` *g-V0 G*) $\times$ *UNIV*))}

theorem *cyc-checker-correct*: *cyc-checker* $G \leq$ *cyc-checker-spec* G
  unfolding *cyc-checker-def cyc-checker-spec-def*
proof (*refine-vcg le-ASSERTI order-trans*[*OF DFS.it-dfs-correct*], *clarsimp-all*)
  assume *fb-graph G*
  then interpret *fb-graph G* **.**
  interpret *cycc* **by** *unfold-locales*
  show *DFS G cycc-params* **by** *unfold-locales*
next
  fix *s*
  assume *cycc-invar G s*
  then interpret *cycc-invar G s* **.**
  assume $\neg$*cycc.cond TYPE('b) G s*
  thus *break* $s = (\neg$ *acyclic* (*g-E G* $\cap$ *cycc.reachable TYPE('b) G* $\times$ *UNIV*))
    by (*rule cycc-correct-aux*)
qed

The same for the total correct variant:

definition *cyc-checkerT-spec* $G \equiv$ *do* {
  ASSERT (*graph G* $\wedge$ *finite* (*graph-defs.reachable G*));
  SPEC ($\lambda r.\ r \longleftrightarrow \neg$*acyclic* (*g-E G* $\cap$ ((*g-E G*)* `` *g-V0 G*) $\times$ *UNIV*))}

theorem *cyc-checkerT-correct*: *cyc-checkerT* $G \leq$ *cyc-checkerT-spec* G
  unfolding *cyc-checkerT-def cyc-checkerT-spec-def*
proof (*refine-vcg le-ASSERTI order-trans*[*OF DFS.it-dfsT-correct*], *clarsimp-all*)
  assume *graph G* **then interpret** *graph G* **.**
  assume *finite* (*graph-defs.reachable G*)
  then interpret *fb-graph G* **by** (*rule fb-graphI-fr*)
  interpret *cycc* **by** *unfold-locales*
  show *DFS G cycc-params* **by** *unfold-locales*
next
  fix *s*
  assume *cycc-invar G s*
  then interpret *cycc-invar G s* **.**
  assume $\neg$*cycc.cond TYPE('b) G s*
  thus *break* $s = (\neg$ *acyclic* (*g-E G* $\cap$ *cycc.reachable TYPE('b) G* $\times$ *UNIV*))
    by (*rule cycc-correct-aux*)
qed

2.1.3 Implementation

The implementation has two aspects: Structural implementation and data implementation. The framework provides recursive and tail-recursive implementations, as well as a variety of data structures for the state.

We will choose the *simple-state* implementation, which provides a stack, an on-stack and a visited set, but no timing information.

Note that it is common for state implementations to omit details from the very detailed abstract state. This means, that the algorithm's operations must not access these details (e.g. timing). However, the algorithm's correctness proofs may still use them.

We extend the state template to add a break flag

record *'v cycc-state-impl* = *'v simple-state* +
  break :: *bool*

Definition of refinement relation: The break-flag is refined by identity.

definition *cycc-erel* ≡ {
  (⦇ *cycc-state-impl.break* = *b* ⦈, ⦇ *cycc-state.break* = *b*⦈) | *b*. *True* }
abbreviation *cycc-rel* ≡ ⟨*cycc-erel*⟩*simple-state-rel*

Implementation of the parameters

definition *cycc-params-impl*
  :: (*'v*,*'v cycc-state-impl*,*unit cycc-state-impl-ext*) *gen-parameterization*
where *cycc-params-impl*
  ≡ *dflt-parametrization simple-state.more* (*RETURN* ⦇ *break* = *False* ⦈) ⦇
  on-back-edge := λ*u v s*. *RETURN* ⦇ *break* = *True* ⦈,
  is-break := *break* ⦈
lemmas *cycc-params-impl-simp*[*simp*,*DFS-code-unfold*] =
  gen-parameterization.simps[*mk-record-simp*, *OF cycc-params-impl-def*[*simplified*]]

Note: In this simple case, the reformulation of the extension state and parameterization is just redundant, However, in general the refinement will also affect the parameterization.

lemma *break-impl*: (*si*,*s*)∈*cycc-rel*
  ⟹ *cycc-state-impl.break si* = *cycc-state.break s*
  by (*cases si*, *cases s*, *simp add*: *simple-state-rel-def cycc-erel-def*)

interpretation *cycc-impl*: *simple-impl-defs G cycc-params-impl cycc-params*
  for *G* **.**

The above interpretation creates an iterative and a recursive implementation

term *cycc-impl.tailrec-impl* **term** *cycc-impl.rec-impl*
term *cycc-impl.tailrec-implT* — Note, for total correctness we currently only support tail-recursive implementations.

We use both to derive a tail-recursive and a recursive cyclicity checker:

definition [*DFS-code-unfold*]: *cyc-checker-impl G* ≡ *do* {
  ASSERT (*fb-graph G*);
  s ← *cycc-impl.tailrec-impl TYPE*(*'a*) *G*;
  RETURN (*break s*)
}

definition [*DFS-code-unfold*]: *cyc-checker-rec-impl G* ≡ *do* {
  ASSERT (*fb-graph G*);
  s ← *cycc-impl.rec-impl TYPE*(*'a*) *G*;

```
  RETURN (break s)
}

definition [DFS-code-unfold]: cyc-checker-implT G ≡ do {
  ASSERT (graph G ∧ finite (graph-defs.reachable G));
  s ← cycc-impl.tailrec-implT TYPE('a) G;
  RETURN (break s)
}
```

To show correctness of the implementation, we integrate the locale of the simple implementation into our cyclicity checker's locale:

```
context cycc begin
  sublocale simple-impl G cycc-params cycc-params-impl cycc-erel
    apply unfold-locales
    apply (intro fun-relI, clarsimp simp: simple-state-rel-def, parametricity) []
    apply (auto simp: cycc-erel-def break-impl simple-state-rel-def)
    done
```

We get that our implementation refines the abstrct DFS algorithm.

```
  lemmas impl-refine = simple-tailrec-refine simple-rec-refine simple-tailrecT-refine
```

Unfortunately, the combination of locales and abbreviations gets to its limits here, so we state the above lemma a bit more readable:

```
  lemma
    cycc-impl.tailrec-impl TYPE('more) G ≤ ⇓ cycc-rel it-dfs
    cycc-impl.rec-impl TYPE('more) G ≤ ⇓ cycc-rel it-dfs
    cycc-impl.tailrec-implT TYPE('more) G ≤ ⇓ cycc-rel it-dfsT
    using impl-refine .

end
```

Finally, we get correctness of our cyclicity checker implementations

```
lemma cyc-checker-impl-refine: cyc-checker-impl G ≤ ⇓Id (cyc-checker G)
  unfolding cyc-checker-impl-def cyc-checker-def
  apply (refine-vcg cycc.impl-refine)
  apply (simp-all add: break-impl cyccI)
  done

lemma cyc-checker-rec-impl-refine:
  cyc-checker-rec-impl G ≤ ⇓Id (cyc-checker G)
  unfolding cyc-checker-rec-impl-def cyc-checker-def
  apply (refine-vcg cycc.impl-refine)
  apply (simp-all add: break-impl cyccI)
  done

lemma cyc-checker-implT-refine: cyc-checker-implT G ≤ ⇓Id (cyc-checkerT G)
  unfolding cyc-checker-implT-def cyc-checkerT-def
  apply (refine-vcg cycc.impl-refine)
```

apply (*simp-all add*: *break-impl cyccI'*)
done

2.1.4 Synthesizing Executable Code

Our algorithm's implementation is still abstract, as it uses abstract data structures like sets and relations. In a last step, we use the Autoref tool to derive an implementation with efficient data structures.

Again, we derive our state implementation from the template provided by the framework. The break-flag is implemented by a Boolean flag. Note that, in general, the user-defined state extensions may be data-refined in this step.

record (*'si,'nsi,'psi*)*cycc-state-impl'* = (*'si,'nsi*)*simple-state-impl* +
break-impl :: *bool*

We define the refinement relation for the state extension

definition [*to-relAPP*]: *cycc-state-erel erel* ≡ {
$((\!|break\text{-}impl = bi, \ldots = mi|\!),(\!|break = b, \ldots = m|\!)) \mid bi\ mi\ b\ m.$
$(bi,b){\in}bool\text{-}rel \wedge (mi,m){\in}erel\}$

And register it with the Autoref tool:

consts
i-cycc-state-ext :: *interface* ⇒ *interface*

lemmas [*autoref-rel-intf*] = *REL-INTFI*[*of cycc-state-erel i-cycc-state-ext*]

We show that the record operations on our extended state are parametric, and declare these facts to Autoref:

lemma [*autoref-rules*]:
fixes *ns-rel vis-rel erel*
defines $R \equiv \langle ns\text{-}rel,vis\text{-}rel,\langle erel\rangle cycc\text{-}state\text{-}erel\rangle ss\text{-}impl\text{-}rel$
shows
$(cycc\text{-}state\text{-}impl'\text{-}ext,\ cycc\text{-}state\text{-}impl\text{-}ext) \in bool\text{-}rel \rightarrow erel \rightarrow \langle erel\rangle cycc\text{-}state\text{-}erel$
$(break\text{-}impl,\ cycc\text{-}state\text{-}impl.break) \in R \rightarrow bool\text{-}rel$
unfolding *cycc-state-erel-def ss-impl-rel-def R-def*
by *auto*

Finally, we can synthesize an implementation for our cyclicity checker, using the standard Autoref-approach:

schematic-goal *cyc-checker-impl*:
defines $V \equiv Id :: ('v \times 'v{::}hashable)\ set$
assumes [*unfolded V-def,autoref-rules*]:
$(Gi,\ G) \in \langle Rm,\ V\rangle g\text{-}impl\text{-}rel\text{-}ext$
notes [*unfolded V-def,autoref-tyrel*] =
TYRELI[**where** $R{=}\langle V\rangle dflt\text{-}ahs\text{-}rel$]
TYRELI[**where** $R{=}\langle V \times_r \langle V\rangle list\text{-}set\text{-}rel\rangle ras\text{-}rel$]
shows $nres\text{-}of\ (?c{::}?'c\ dres) \le\Downarrow ?R\ (cyc\text{-}checker\text{-}impl\ G)$

unfolding *DFS-code-unfold*
using [[*autoref-trace-failed-id*, *goals-limit=1*]]
apply (*autoref-monadic* (*trace*))
done
concrete-definition *cyc-checker-code* **uses** *cyc-checker-impl*
export-code *cyc-checker-code* **checking** *SML*

Combining the refinement steps yields a correctness theorem for the cyclicity checker implementation:

theorem *cyc-checker-code-correct*:
assumes *1*: *fb-graph G*
assumes *2*: $(Gi,\ G) \in \langle Rm,\ Id\rangle g\text{-}impl\text{-}rel\text{-}ext$
assumes *4*: *cyc-checker-code Gi* = *dRETURN x*
shows $x \longleftrightarrow (\neg acyclic\ (g\text{-}E\ G \cap ((g\text{-}E\ G)^{*}\ ``\ g\text{-}V0\ G) \times UNIV))$
proof −
note *cyc-checker-code.refine*[*OF 2*]
also note *cyc-checker-impl-refine*
also note *cyc-checker-correct*
finally show *?thesis* **using** *1 4*
unfolding *cyc-checker-spec-def* **by** *auto*
qed

We can repeat the same boilerplate for the recursive version of the algorithm:

schematic-goal *cyc-checker-rec-impl*:
defines $V \equiv Id :: ('v \times 'v::hashable)\ set$
assumes [*unfolded V-def*,*autoref-rules*]:
$(Gi,\ G) \in \langle Rm,\ V\rangle g\text{-}impl\text{-}rel\text{-}ext$
notes [*unfolded V-def*,*autoref-tyrel*] =
TYRELI[**where** $R=\langle V\rangle dflt\text{-}ahs\text{-}rel$]
TYRELI[**where** $R=\langle V \times_r \langle V\rangle list\text{-}set\text{-}rel\rangle ras\text{-}rel$]
shows $nres\text{-}of\ (?c::?'c\ dres) \le\Downarrow ?R\ (cyc\text{-}checker\text{-}rec\text{-}impl\ G)$
unfolding *DFS-code-unfold*
using [[*autoref-trace-failed-id*, *goals-limit=1*]]
apply (*autoref-monadic* (*trace*))
done
concrete-definition *cyc-checker-rec-code* **uses** *cyc-checker-rec-impl*
prepare-code-thms *cyc-checker-rec-code-def*
export-code *cyc-checker-rec-code* **checking** *SML*

lemma *cyc-checker-rec-code-correct*:
assumes *1*: *fb-graph G*
assumes *2*: $(Gi,\ G) \in \langle Rm,\ Id\rangle g\text{-}impl\text{-}rel\text{-}ext$
assumes *4*: *cyc-checker-rec-code Gi* = *dRETURN x*
shows $x \longleftrightarrow (\neg acyclic\ (g\text{-}E\ G \cap ((g\text{-}E\ G)^{*}\ ``\ g\text{-}V0\ G) \times UNIV))$
proof −
note *cyc-checker-rec-code.refine*[*OF 2*]
also note *cyc-checker-rec-impl-refine*
also note *cyc-checker-correct*
finally show *?thesis* **using** *1 4*

unfolding *cyc-checker-spec-def* **by** *auto*
qed

And, again, for the total correct version. Note that we generate a plain implementation, not inside a monad:

schematic-goal *cyc-checker-implT*:
  defines $V \equiv Id :: ('v \times 'v::hashable)\ set$
  assumes [*unfolded V-def*,*autoref-rules*]:
    $(Gi,\ G) \in \langle Rm,\ V\rangle g\text{-}impl\text{-}rel\text{-}ext$
  notes [*unfolded V-def*,*autoref-tyrel*] =
    TYRELI[**where** $R=\langle V\rangle dflt\text{-}ahs\text{-}rel$]
    TYRELI[**where** $R=\langle V \times_r \langle V\rangle list\text{-}set\text{-}rel\rangle ras\text{-}rel$]
  shows $RETURN\ (?c::?'c) \le\Downarrow ?R\ (cyc\text{-}checker\text{-}implT\ G)$
  unfolding *DFS-code-unfold*
  using [[*autoref-trace-failed-id*, *goals-limit=1*]]
  apply (*autoref-monadic* (*trace*,*plain*))
  done
concrete-definition *cyc-checker-codeT* **uses** *cyc-checker-implT*
export-code *cyc-checker-codeT* **checking** *SML*

theorem *cyc-checker-codeT-correct*:
  assumes *1*: *graph G finite* (*graph-defs.reachable G*)
  assumes *2*: $(Gi,\ G) \in \langle Rm,\ Id\rangle g\text{-}impl\text{-}rel\text{-}ext$
  shows $cyc\text{-}checker\text{-}codeT\ Gi \longleftrightarrow (\neg acyclic\ (g\text{-}E\ G \cap ((g\text{-}E\ G)^*\ ``\ g\text{-}V0\ G) \times UNIV))$
proof −
  note *cyc-checker-codeT.refine*[*OF 2*]
  also note *cyc-checker-implT-refine*
  also note *cyc-checkerT-correct*
  finally show *?thesis* **using** *1*
  unfolding *cyc-checkerT-spec-def* **by** *auto*
qed

end

2.2 Finding a Path between Nodes

theory *DFS-Find-Path*
imports
  ../DFS-Framework
  CAVA-Automata.Digraph-Impl
  ../Misc/Impl-Rev-Array-Stack
begin

We instantiate the DFS framework to find a path to some reachable node that satisfies a given predicate. We present four variants of the algorithm: Finding any path, and finding path of at least length one, combined with searching the whole graph, and searching the graph restricted to a given set of nodes. The restricted variants are efficiently implemented by pre-initializing the visited set (cf. *DFS-Framework.Restr-Impl*).

The restricted variants can be used for incremental search, ignoring already searched nodes in further searches. This is required, e.g., for the inner search of nested DFS (Buchi automaton emptiness check).

2.2.1 Including empty Path

record $'v$ *fp0-state* = $'v$ *state* +
  ppath :: $('v$ *list* $\times$ $'v)$ *option*

type-synonym $'v$ *fp0-param* = $('v$, $('v$,*unit*) *fp0-state-ext*) *parameterization*

lemma [*simp*]: *s*⦇ *state.more* := ⦇ *ppath* = *foo* ⦈ ⦈ = *s* ⦇ *ppath* := *foo* ⦈
  by (*cases s*) *simp*

abbreviation *no-path* ≡ ⦇ *ppath* = *None* ⦈
abbreviation *a-path p v* ≡ ⦇ *ppath* = *Some* (*p*,*v*) ⦈

definition *fp0-params* :: $('v \Rightarrow$ *bool*) $\Rightarrow$ $'v$ *fp0-param*
  where *fp0-params P* ≡ ⦇
  on-init = *RETURN no-path*,
  on-new-root = λ*v0 s. if P v0 then RETURN* (*a-path* [] *v0*) *else RETURN no-path*,
  on-discover = λ*u v s. if P v*
    then — *v* is already on the stack, so we need to pop it again
      RETURN (*a-path* (*rev* (*tl* (*stack s*))) *v*)
    else RETURN no-path,
  on-finish = λ*u s. RETURN* (*state.more s*),
  on-back-edge = λ*u v s. RETURN* (*state.more s*),
  on-cross-edge = λ*u v s. RETURN* (*state.more s*),
  is-break = λ*s. ppath s* $\neq$ *None* ⦈

lemmas *fp0-params-simps*[*simp*]
  = *gen-parameterization.simps*[*mk-record-simp*, *OF fp0-params-def*]

interpretation *fp0*: *param-DFS-defs* **where** *param* = *fp0-params P*
  for *G P* **.**

locale *fp0* = *param-DFS G fp0-params P*
  for *G* **and** *P* :: $'v \Rightarrow$ *bool*
begin

  lemma [*simp*]:
    ppath (*empty-state* ⦇*ppath* = *e*⦈) = *e*
    by (*simp add*: *empty-state-def*)

  lemma [*simp*]:
    ppath (*s*⦇*state.more* := *state.more s'*⦈) = *ppath s'*

by (*cases s*, *cases s′*) *auto*

sublocale *DFS* **where** *param* = *fp0-params P*
by *unfold-locales simp-all*

end

lemma *fp0I*: **assumes** *fb-graph G* **shows** *fp0 G*
proof − **interpret** *fb-graph G* **by** *fact* **show** *?thesis* **by** *unfold-locales* **qed**

locale *fp0-invar* = *fp0* +
DFS-invar **where** *param* = *fp0-params P*

lemma *fp0-invar-eq*[*simp*]:
DFS-invar G (*fp0-params P*) = *fp0-invar G P*
proof (*intro ext iffI*)
fix *s*
assume *DFS-invar G* (*fp0-params P*) *s*
interpret *DFS-invar G fp0-params P s* **by** *fact*
show *fp0-invar G P s* **by** *unfold-locales*
next
fix *s*
assume *fp0-invar G P s*
interpret *fp0-invar G P s* **by** *fact*
show *DFS-invar G* (*fp0-params P*) *s* **by** *unfold-locales*
qed

context *fp0* **begin**

lemma *i-no-path-no-P-discovered*:
is-invar (λ*s. ppath s* = *None* $\longrightarrow$ *dom* (*discovered s*) $\cap$ *Collect P* = {})
by (*rule establish-invarI*) *simp-all*

lemma *i-path-to-P*:
is-invar (λ*s. ppath s* = *Some* (*vs*,*v*) $\longrightarrow$ *P v*)
by (*rule establish-invarI*) *auto*

lemma *i-path-invar*:
is-invar (λ*s. ppath s* = *Some* (*vs*,*v*) $\longrightarrow$
  (*vs* $\neq$ [] $\longrightarrow$ *hd vs* $\in$ *V0* $\wedge$ *path E* (*hd vs*) *vs v*)
  $\wedge$ (*vs* = [] $\longrightarrow$ *v* $\in$ *V0* $\wedge$ *path E v vs v*)
  $\wedge$ (*distinct* (*vs*@[*v*]))
)
proof (*induct rule*: *establish-invarI*)
case (*discover s s′ u v*) **then interpret** *fp0-invar* **where** *s*=*s*
by *simp*

from *discover* **have** *ne*: *stack s* $\neq$ [] **by** *simp*
from *discover* **have** *vnis*: *v*$\notin$*set* (*stack s*) **using** *stack-discovered* **by** *auto*

from *pendingD discover* **have** $v \in succ\ (hd\ (stack\ s))$ **by** *simp*
with *hd-succ-stack-is-path*[*OF ne*] **have** $\exists\, v0 \in V0.\ path\ E\ v0\ (rev\ (stack\ s))\ v$ **.**
moreover from *last-stack-in-V0 ne* **have** $last\ (stack\ s) \in V0$ **by** *simp*
ultimately have $path\ E\ (hd\ (rev\ (stack\ s)))\ \ (rev\ (stack\ s))\ v\ hd\ (rev\ (stack\ s)) \in V0$
using *hd-rev path-hd*[**where** $p = rev\ (stack\ s)$] *ne*
by *auto*
with *ne discover vnis* **show** *?case* **by** (*auto simp*: *stack-distinct*)
qed *auto*
end

context *fp0-invar*
begin
lemmas *no-path-no-P-discovered*
= *i-no-path-no-P-discovered*[*THEN make-invar-thm*, *rule-format*]

lemmas *path-to-P*
= *i-path-to-P*[*THEN make-invar-thm*, *rule-format*]

lemmas *path-invar*
= *i-path-invar*[*THEN make-invar-thm*, *rule-format*]

lemma *path-invar-nonempty*:
assumes $ppath\ s = Some\ (vs,v)$
and $vs \neq []$
shows $hd\ vs \in V0\ path\ E\ (hd\ vs)\ vs\ v$
using *assms path-invar*
by *auto*

lemma *path-invar-empty*:
assumes $ppath\ s = Some\ (vs,v)$
and $vs = []$
shows $v \in V0\ path\ E\ v\ vs\ v$
using *assms path-invar*
by *auto*

lemma *fp0-correct*:
assumes $\neg cond\ s$
shows *case ppath s of*
$None \Rightarrow \neg(\exists\, v0 \in V0.\ \exists\, v.\ (v0,v) \in E^* \wedge P\ v)$
$\mid Some\ (p,v) \Rightarrow (\exists\, v0 \in V0.\ path\ E\ v0\ p\ v \wedge P\ v \wedge distinct\ (p@[v]))$
proof (*cases ppath s*)
case *None* **with** *assms nc-discovered-eq-reachable no-path-no-P-discovered* **have**
$reachable \cap Collect\ P = \{\}$ **by** *auto*
thus *?thesis* **by** (*auto simp add*: *None*)
next
case (*Some vvs*) **then obtain** *v vs* **where** [*simp*]: $vvs = (vs,v)$
by (*cases vvs*) *auto*

from *Some path-invar*[*of vs v*] *path-to-P*[*of - v*] **show** *?thesis*
by *auto*
qed

end

context *fp0* **begin**
lemma *fp0-correct*: *it-dfs* $\leq$ *SPEC* (λs. *case ppath s of*
None $\Rightarrow \neg(\exists\, v0 \in V0.\ \exists\, v.\ (v0,v) \in E^* \wedge P\ v)$
| *Some* $(p,v) \Rightarrow (\exists\, v0 \in V0.\ path\ E\ v0\ p\ v \wedge P\ v \wedge distinct\ (p@[v])))$
apply (*rule weaken-SPEC*[*OF it-dfs-correct*])
apply *clarsimp*
apply (*simp add*: *fp0-invar.fp0-correct*)
done
end

Basic Interface

Use this interface, rather than the internal stuff above!

type-synonym *'v fp-result* = (*'v list* × *'v*) *option*
definition *find-path0-pred G P* ≡ λr. *case r of*
None $\Rightarrow$ (*g-E G*)* `` *g-V0 G* ∩ *Collect P* = {}
| *Some* (*vs*,*v*) $\Rightarrow$ *P v* ∧ *distinct* (*vs*@[*v*]) ∧ (∃ *v0* ∈ *g-V0 G*. *path* (*g-E G*) *v0 vs v*)

definition *find-path0-spec*
:: (*'v*, -) *graph-rec-scheme* ⇒ (*'v* ⇒ *bool*) ⇒ *'v fp-result nres*
— Searches a path from the root nodes to some target node that satisfies a given predicate. If such a path is found, the path and the target node are returned
where
find-path0-spec G P ≡ *do* {
ASSERT (*fb-graph G*);
SPEC (*find-path0-pred G P*)
}

definition *find-path0*
:: (*'v*, *'more*) *graph-rec-scheme* ⇒ (*'v* ⇒ *bool*) ⇒ *'v fp-result nres*
where *find-path0 G P* ≡ *do* {
ASSERT (*fp0 G*);
s ← *fp0.it-dfs TYPE*(*'more*) *G P*;
RETURN (*ppath s*)
}

lemma *find-path0-correct*:
shows *find-path0 G P* ≤ *find-path0-spec G P*
unfolding *find-path0-def find-path0-spec-def find-path0-pred-def*
apply (*refine-vcg le-ASSERTI order-trans*[*OF fp0.fp0-correct*])
apply (*erule fp0I*)

apply (*auto split*: *option.split*) []
done

lemmas *find-path0-spec-rule*[*refine-vcg*] =
ASSERT-le-defI[*OF find-path0-spec-def*]
ASSERT-leof-defI[*OF find-path0-spec-def*]

2.2.2 Restricting the Graph

Extended interface, propagating set of already searched nodes (restriction)

definition *restr-invar*
— Invariant for a node restriction, i.e., a transition closed set of nodes known to not contain a target node that satisfies a predicate.
where
$restr\text{-}invar\ E\ R\ P \equiv E\ ``\ R \subseteq R \wedge R \cap Collect\ P = \{\}$

lemma *restr-invar-triv*[*simp*, *intro*!]: $restr\text{-}invar\ E\ \{\}\ P$
unfolding *restr-invar-def* **by** *simp*

lemma *restr-invar-imp-not-reachable*: $restr\text{-}invar\ E\ R\ P \Longrightarrow E^*\ ``R \cap Collect\ P = \{\}$
unfolding *restr-invar-def* **by** (*simp add*: *Image-closed-trancl*)

type-synonym $'v\ fpr\text{-}result = 'v\ set + ('v\ list \times 'v)$
definition $find\text{-}path0\text{-}restr\text{-}pred\ G\ P\ R \equiv \lambda r.$
case r of
$Inl\ R' \Rightarrow R' = R \cup (g\text{-}E\ G)^*\ ``\ g\text{-}V0\ G \wedge restr\text{-}invar\ (g\text{-}E\ G)\ R'\ P$
$|\ Inr\ (vs,v) \Rightarrow P\ v \wedge (\exists\ v0 \in g\text{-}V0\ G - R.\ path\ (rel\text{-}restrict\ (g\text{-}E\ G)\ R)\ v0\ vs\ v)$

definition *find-path0-restr-spec*
— Find a path to a target node that satisfies a predicate, not considering nodes from the given node restriction. If no path is found, an extended restriction is returned, that contains the start nodes
where $find\text{-}path0\text{-}restr\text{-}spec\ G\ P\ R \equiv do\ \{$
$ASSERT\ (fb\text{-}graph\ G \wedge restr\text{-}invar\ (g\text{-}E\ G)\ R\ P);$
$SPEC\ (find\text{-}path0\text{-}restr\text{-}pred\ G\ P\ R)\}$

lemmas *find-path0-restr-spec-rule*[*refine-vcg*] =
ASSERT-le-defI[*OF find-path0-restr-spec-def*]
ASSERT-leof-defI[*OF find-path0-restr-spec-def*]

definition *find-path0-restr*
$:: ('v, 'more)\ graph\text{-}rec\text{-}scheme \Rightarrow ('v \Rightarrow bool) \Rightarrow 'v\ set \Rightarrow 'v\ fpr\text{-}result\ nres$
where $find\text{-}path0\text{-}restr\ G\ P\ R \equiv do\ \{$
$ASSERT\ (fb\text{-}graph\ G);$
$ASSERT\ (fp0\ (graph\text{-}restrict\ G\ R));$
$s \leftarrow fp0.it\text{-}dfs\ TYPE('more)\ (graph\text{-}restrict\ G\ R)\ P;$

```
  case ppath s of
    None ⇒ do {
      ASSERT (dom (discovered s) = dom (finished s));
      RETURN (Inl (R ∪ dom (finished s)))
    }
  | Some (vs,v) ⇒ RETURN (Inr (vs,v))
}


lemma find-path0-restr-correct:
  shows find-path0-restr G P R ≤ find-path0-restr-spec G P R
proof (rule le-ASSERT-defI1[OF find-path0-restr-spec-def], clarify)
  assume fb-graph G
  interpret a: fb-graph G by fact
  interpret fb-graph graph-restrict G R by (rule a.fb-graph-restrict)

  assume I: restr-invar (g-E G) R P

  define reachable where reachable = graph-defs.reachable (graph-restrict G R)

  interpret fp0 graph-restrict G R by unfold-locales

  show ?thesis unfolding find-path0-restr-def find-path0-restr-spec-def
    apply (refine-rcg refine-vcg le-ASSERTI order-trans[OF it-dfs-correct])
    apply unfold-locales
    apply (clarsimp-all)
  proof −
    fix s
    assume fp0-invar (graph-restrict G R) P s
      and NC[simp]: ¬fp0.cond TYPE('b) (graph-restrict G R) P s
    then interpret fp0-invar graph-restrict G R P s by simp

    {
      assume [simp]: ppath s = None

      from nc-discovered-eq-finished
      show dom (discovered s) = dom (finished s) by simp

      from nc-finished-eq-reachable
      have DFR[simp]: dom (finished s) = reachable
        by (simp add: reachable-def)

      from I have g-E G `` R ⊆ R unfolding restr-invar-def by auto

      have reachable ⊆ (g-E G)* `` g-V0 G
        unfolding reachable-def
        by (rule Image-mono, rule rtrancl-mono) (auto simp: rel-restrict-def)

      hence R ∪ dom (finished s) = R ∪ (g-E G)* `` g-V0 G
```

apply −
apply (*rule equalityI*)
apply *auto* []
unfolding *DFR reachable-def*
apply (*auto elim*: *E-closed-restr-reach-cases*[*OF* - ‹*g-E G* ‘‘ *R* ⊆ *R*›]) []
done
moreover from *nc-fin-closed I*
have *g-E G* ‘‘ (*R* ∪ *dom* (*finished s*)) ⊆ *R* ∪ *dom* (*finished s*)
unfolding *restr-invar-def* **by** (*simp add*: *rel-restrict-def*) *blast*
moreover from *no-path-no-P-discovered nc-discovered-eq-finished I*
have (*R* ∪ *dom* (*finished s*)) ∩ *Collect P* = {}
unfolding *restr-invar-def* **by** *auto*
ultimately
show *find-path0-restr-pred G P R* (*Inl* (*R* ∪ *dom* (*finished s*)))
unfolding *restr-invar-def find-path0-restr-pred-def* **by** *auto*
}

{
fix *v vs*
assume [*simp*]: *ppath s* = *Some* (*vs*,*v*)
from *fp0-correct*
show *find-path0-restr-pred G P R* (*Inr* (*vs*, *v*))
unfolding *find-path0-restr-pred-def* **by** *auto*
}
qed
qed

2.2.3 Path of Minimal Length One, with Restriction

definition *find-path1-restr-pred G P R* ≡ *λr*.
case r of
Inl R′ ⇒ *R′* = *R* ∪ (*g-E G*)$^+$ ‘‘ *g-V0 G* ∧ *restr-invar* (*g-E G*) *R′ P*
| *Inr* (*vs*,*v*) ⇒ *P v* ∧ *vs* ≠ [] ∧ (∃ *v0* ∈ *g-V0 G*. *path* (*g-E G* ∩ *UNIV* × −*R*) *v0 vs v*)

definition *find-path1-restr-spec*
— Find a path of length at least one to a target node that satisfies P. Takes an initial node restriction, and returns an extended node restriction.
where *find-path1-restr-spec G P R* ≡ *do* {
ASSERT (*fb-graph G* ∧ *restr-invar* (*g-E G*) *R P*);
SPEC (*find-path1-restr-pred G P R*)}

lemmas *find-path1-restr-spec-rule*[*refine-vcg*] =
ASSERT-le-defI[*OF find-path1-restr-spec-def*]
ASSERT-leof-defI[*OF find-path1-restr-spec-def*]

definition *find-path1-restr*
:: (′*v*, ′*more*) *graph-rec-scheme* ⇒ (′*v* ⇒ *bool*) ⇒ ′*v set* ⇒ ′*v fpr-result nres*
where *find-path1-restr G P R* ≡

FOREACHc (*g-V0 G*) *is-Inl* (λ*v0 s*. *do* {
ASSERT (*is-Inl s*); — TODO: Add FOREACH-condition as precondition in autoref!
let R = *projl s*;
f0 ← *find-path0-restr-spec* (*G* ⦇ *g-V0* := *g-E G* ‘‘ {*v0*} ⦈) *P R*;
case f0 of
Inl - ⇒ *RETURN f0*
| *Inr* (*vs*,*v*) ⇒ *RETURN* (*Inr* (*v0*#*vs*,*v*))
}) (*Inl R*)

definition *find-path1-tailrec-invar G P R0 it s* ≡
case s of
Inl R ⇒ *R* = *R0* ∪ (*g-E G*)$^+$ ‘‘ (*g-V0 G* − *it*) ∧ *restr-invar* (*g-E G*) *R P*
| *Inr* (*vs*, *v*) ⇒ *P v* ∧ *vs* ≠ [] ∧ (∃ *v0* ∈ *g-V0 G* − *it*. *path* (*g-E G* ∩ *UNIV* × −*R0*) *v0 vs v*)

lemma *find-path1-restr-correct*:
shows *find-path1-restr G P R* ≤ *find-path1-restr-spec G P R*
proof (*rule le-ASSERT-defI1*[*OF find-path1-restr-spec-def*], *clarify*)
assume *fb-graph G*
interpret *a*: *fb-graph G* **by** *fact*
interpret *fb0*: *fb-graph G* ⦇ *g-E* := *g-E G* ∩ *UNIV* × −*R* ⦈
by (*rule a.fb-graph-subset*, *auto*)

assume *I*: *restr-invar* (*g-E G*) *R P*

have *aux2*: ⋀*v0*. *v0* ∈ *g-V0 G* ⟹ *fb-graph* (*G* ⦇ *g-V0* := *g-E G* ‘‘ {*v0*} ⦈)
by (*rule a.fb-graph-subset*, *auto*)

{
fix *v0 it s*
assume *IT*: *it* ⊆ *g-V0 G v0* ∈ *it*
and *is-Inl s*
and *FPI*: *find-path1-tailrec-invar G P R it s*
and *RI*: *restr-invar* (*g-E G*) (*projl s* ∪ (*g-E G*)$^+$ ‘‘ {*v0*}) *P*

then obtain *R′* **where** [*simp*]: *s* = *Inl R′* **by** (*cases s*) *auto*

from *FPI* **have** [*simp*]: *R′* = *R* ∪ (*g-E G*)$^+$ ‘‘ (*g-V0 G* − *it*)
unfolding *find-path1-tailrec-invar-def* **by** *simp*

have *find-path1-tailrec-invar G P R* (*it* − {*v0*})
(*Inl* (*projl s* ∪ (*g-E G*)$^+$ ‘‘ {*v0*}))
using *RI*
by (*auto simp*: *find-path1-tailrec-invar-def it-step-insert-iff*[*OF IT*])
} **note** *aux4* = *this*

{

fix *v0 u it s v p*
assume *IT*: *it* ⊆ *g-V0 G v0* ∈ *it*
and *is-Inl s*
and *FPI*: *find-path1-tailrec-invar G P R it s*
and *PV*: *P v*
and *PATH*: *path* (*rel-restrict* (*g-E G*) (*projl s*)) *u p v* (*v0*,*u*)∈(*g-E G*)
and *PR*: *u* ∉ *projl s*

then obtain *R′* **where** [*simp*]: *s* = *Inl R′* **by** (*cases s*) *auto*

from *FPI* **have** [*simp*]: *R′* = *R* ∪ (*g-E G*)⁺ '' (*g-V0 G* − *it*)
unfolding *find-path1-tailrec-invar-def* **by** *simp*

have *find-path1-tailrec-invar G P R* (*it* − {*v0*}) (*Inr* (*v0* # *p*, *v*))
apply (*simp add*: *find-path1-tailrec-invar-def PV*)
apply (*rule bexI*[**where** *x*=*v0*])
using *PR PATH*(*2*) *path-mono*[*OF rel-restrict-mono2*[*of R*] *PATH*(*1*)]
apply (*auto simp*: *path1-restr-conv*) []

using *IT* **apply** *blast*
done
} **note** *aux5* = *this*

show *?thesis*
unfolding *find-path1-restr-def find-path1-restr-spec-def find-path1-restr-pred-def*
apply (*refine-rcg le-ASSERTI*
refine-vcg FOREACHc-rule[**where** *I*=*find-path1-tailrec-invar G P R*]

)
apply *simp*
using *I* **apply** (*auto simp add*: *find-path1-tailrec-invar-def restr-invar-def*) []

apply (*blast intro*: *aux2*)
apply (*auto simp add*: *find-path1-tailrec-invar-def split*: *sum.splits*) []
apply (*auto*
simp: *find-path0-restr-pred-def aux4 aux5*
simp: *trancl-Image-unfold-left*[*symmetric*]
split: *sum.splits*) []

apply (*auto simp add*: *find-path1-tailrec-invar-def split*: *sum.splits*) [*2*]
done
qed

definition *find-path1-pred G P* ≡ λ*r*.
case r of
None ⇒ (*g-E G*)⁺ '' *g-V0 G* ∩ *Collect P* = {}
| *Some* (*vs*, *v*) ⇒ *P v* ∧ *vs* ≠ [] ∧ (∃ *v0* ∈ *g-V0 G*. *path* (*g-E G*) *v0 vs v*)
definition *find-path1-spec*
— Find a path of length at least one to a target node that satisfies a given

predicate.
  where *find-path1-spec G P* ≡ *do* {
    ASSERT (*fb-graph G*);
    SPEC (*find-path1-pred G P*)}

lemmas *find-path1-spec-rule*[*refine-vcg*] =
  ASSERT-le-defI[*OF find-path1-spec-def*]
  ASSERT-leof-defI[*OF find-path1-spec-def*]

2.2.4 Path of Minimal Length One, without Restriction

definition *find-path1*
  :: ('*v*, '*more*) *graph-rec-scheme* ⇒ ('*v* ⇒ *bool*) ⇒ '*v fp-result nres*
  where *find-path1 G P* ≡ *do* {
  r ← *find-path1-restr-spec G P* {};
  case r of
    Inl - ⇒ *RETURN None*
  | *Inr vsv* ⇒ *RETURN* (*Some vsv*)
}

lemma *find-path1-correct*:
  shows *find-path1 G P* ≤ *find-path1-spec G P*
  unfolding *find-path1-def find-path1-spec-def find-path1-pred-def*
  apply (*refine-rcg refine-vcg le-ASSERTI order-trans*[*OF find-path1-restr-correct*])
  apply *simp*
  apply (*fastforce*
    simp: *find-path1-restr-spec-def find-path1-restr-pred-def*
    split: *sum.splits*
    dest!: *restr-invar-imp-not-reachable tranclD*)
  done

2.2.5 Implementation

record '*v fp0-state-impl* = '*v simple-state* +
  ppath :: ('*v list* × '*v*) *option*

definition *fp0-erel* ≡ {
  ((| *fp0-state-impl.ppath* = *p* |), (| *fp0-state.ppath* = *p*|)) | *p*. *True* }

abbreviation *fp0-rel R* ≡ ⟨*fp0-erel*⟩*restr-simple-state-rel R*

abbreviation *no-path-impl* ≡ (| *fp0-state-impl.ppath* = *None* |)
abbreviation *a-path-impl p v* ≡ (| *fp0-state-impl.ppath* = *Some* (*p*,*v*) |)

lemma *fp0-rel-ppath-cong*[*simp*]:
  (*s*,*s*')∈*fp0-rel R* ⟹ *fp0-state-impl.ppath s* = *fp0-state.ppath s*'
  unfolding *restr-simple-state-rel-def fp0-erel-def*
  by (*cases s*, *cases s*', *auto*)

lemma *fp0-ss-rel-ppath-cong*[*simp*]:

$(s,s')\in\langle fp0\text{-}erel\rangle simple\text{-}state\text{-}rel \Longrightarrow fp0\text{-}state\text{-}impl.ppath\ s = fp0\text{-}state.ppath\ s'$
unfolding *simple-state-rel-def fp0-erel-def*
by (*cases s*, *cases s'*, *auto*)

lemma *fp0i-cong*[*cong*]: *simple-state.more s = simple-state.more s'*
$\Longrightarrow$ *fp0-state-impl.ppath s = fp0-state-impl.ppath s'*
by (*cases s*, *cases s'*, *auto*)

lemma *fp0-erelI*: *p=p'*
$\Longrightarrow$ ((| *fp0-state-impl.ppath = p* |), (| *fp0-state.ppath = p'*|))∈*fp0-erel*
unfolding *fp0-erel-def* **by** *auto*

definition *fp0-params-impl*
:: - ⇒ ('*v*,'*v fp0-state-impl*,('*v*,*unit*)*fp0-state-impl-ext*) *gen-parameterization*
where *fp0-params-impl P* ≡ (|
on-init = RETURN no-path-impl,
on-new-root = λv0 s.
if P v0 then RETURN (*a-path-impl* [] *v0*) *else RETURN no-path-impl*,
on-discover = λu v s.
if P v then RETURN (*a-path-impl* (*map fst* (*rev* (*tl* (*CAST* (*ss-stack s*))))) *v*)
else RETURN no-path-impl,
on-finish = λu s. RETURN (*simple-state.more s*),
on-back-edge = λu v s. RETURN (*simple-state.more s*),
on-cross-edge = λu v s. RETURN (*simple-state.more s*),
is-break = λs. ppath s ≠ None |)

lemmas *fp0-params-impl-simp*[*simp*, *DFS-code-unfold*]
= *gen-parameterization.simps*[*mk-record-simp*, *OF fp0-params-impl-def*]

interpretation *fp0-impl*:
restricted-impl-defs fp0-params-impl P fp0-params P G R
for *G P R* **.**

locale *fp0-restr = fb-graph*
begin
sublocale *fp0?*: *fp0 graph-restrict G R*
apply (*rule fp0I*)
apply (*rule fb-graph-restrict*)
done

sublocale *impl*: *restricted-impl G fp0-params P fp0-params-impl P*
fp0-erel R
apply *unfold-locales*
apply *parametricity*

apply (*simp add*: *fp0-erel-def*)
apply (*auto*) [*1*]

apply (*auto*

simp: *rev-map*[*symmetric*] *map-tl comp-def*
simp: *fp0-erel-def simple-state-rel-def*) [*7*]

apply (*auto simp*: *restr-rel-def*) [*3*]
apply (*clarsimp simp*: *restr-rel-def*)
apply (*rule IdD*) **apply** (*subst list-rel-id-simp*[*symmetric*])
apply *parametricity*
done
end

definition *find-path0-restr-impl G P R* ≡ *do* {
ASSERT (*fb-graph G*);
ASSERT (*fp0* (*graph-restrict G R*));
s ← *fp0-impl.tailrec-impl TYPE*(*'a*) *G R P*;
case ppath s of
None ⇒ *RETURN* (*Inl* (*visited s*))
| *Some* (*vs*,*v*) ⇒ *RETURN* (*Inr* (*vs*,*v*))
}

lemma *find-path0-restr-impl*[*refine*]:
shows *find-path0-restr-impl G P R*
≤ ⇓(⟨*Id*,$Id \times_r Id$⟩*sum-rel*)
(*find-path0-restr G P R*)
proof (*rule refine-ASSERT-defI2*[*OF find-path0-restr-def*])
assume *fb-graph G*
then interpret *fb-graph G* **.**
interpret *fp0-restr G* **by** *unfold-locales*

show *?thesis*
unfolding *find-path0-restr-impl-def find-path0-restr-def*
apply (*refine-rcg impl.tailrec-refine*)
apply *refine-dref-type*
apply (*auto simp*: *restr-simple-state-rel-def*)
done
qed

definition *find-path0-impl G P* ≡ *do* {
ASSERT (*fp0 G*);
s ← *fp0-impl.tailrec-impl TYPE*(*'a*) *G* {} *P*;
RETURN (*ppath s*)
}

lemma *find-path0-impl*[*refine*]: *find-path0-impl G P*
≤ ⇓ (⟨$Id \times_r Id$⟩*option-rel*) (*find-path0 G P*)
proof (*rule refine-ASSERT-defI1*[*OF find-path0-def*])
assume *fp0 G*
then interpret *fp0 G* **.**
interpret *r*: *fp0-restr G* **by** *unfold-locales*

show *?thesis*
unfolding *find-path0-impl-def find-path0-def*
apply (*refine-rcg r.impl.tailrec-refine*[**where** *R*={}, *simplified*])
apply (*auto*)
done
qed

2.2.6 Synthesis of Executable Code

record $('v,'si,'nsi)$*fp0-state-impl'* = $('si,'nsi)$*simple-state-nos-impl* +
ppath-impl :: $('v\ list \times 'v)$ *option*

definition [*to-relAPP*]: *fp0-state-erel erel* ≡ {
((|*ppath-impl* = *pi*, ... = *mi*|),(|*ppath* = *p*, ... = *m*|)) | *pi mi p m*.
(*pi*,*p*)∈⟨⟨*Id*⟩*list-rel* $\times_r$ *Id*⟩*option-rel* ∧ (*mi*,*m*)∈*erel*}

consts
i-fp0-state-ext :: *interface* ⇒ *interface*

lemmas [*autoref-rel-intf*] = *REL-INTFI*[*of fp0-state-erel i-fp0-state-ext*]

term *fp0-state-impl-ext*
lemma [*autoref-rules*]:
fixes *ns-rel vis-rel erel*
defines *R* ≡ ⟨*ns-rel*,*vis-rel*,⟨*erel*⟩*fp0-state-erel*⟩*ssnos-impl-rel*
shows
(*fp0-state-impl'-ext*, *fp0-state-impl-ext*)
∈ ⟨⟨*Id*⟩*list-rel* $\times_r$ *Id*⟩*option-rel* → *erel* → ⟨*erel*⟩*fp0-state-erel*
(*ppath-impl*, *fp0-state-impl.ppath*) ∈ *R* → ⟨⟨*Id*⟩*list-rel* $\times_r$ *Id*⟩*option-rel*
unfolding *fp0-state-erel-def ssnos-impl-rel-def R-def*
by *auto*

schematic-goal *find-path0-code*:
fixes *G* :: (*'v* :: *hashable*, -) *graph-rec-scheme*
assumes [*autoref-rules*]:
(*Gi*, *G*) ∈ ⟨*Rm*, *Id*⟩*g-impl-rel-ext*
(*Pi*, *P*) ∈ *Id* → *bool-rel*
notes [*autoref-tyrel*] = *TYRELI*[**where** *R*=⟨*Id*::(*'v*×*'v*) *set*⟩*dflt-ahs-rel*]
shows (*nres-of* (*?c*::*?'c dres*), *find-path0-impl G P*) ∈ *?R*
unfolding *find-path0-impl-def*[*abs-def*] *DFS-code-unfold ssnos-unfolds*
unfolding *if-cancel not-not comp-def nres-monad-laws*
using [[*autoref-trace-failed-id*]]
apply (*autoref-monadic* (*trace*))
done

concrete-definition *find-path0-code* **uses** *find-path0-code*
export-code *find-path0-code* **checking** *SML*

lemma *find-path0-autoref-aux*:
assumes *Vid*: *Rv* = (*Id* :: *'a* :: *hashable rel*)
shows (λ*G P*. *nres-of* (*find-path0-code G P*), *find-path0-spec*)
∈ ⟨*Rm*, *Rv*⟩*g-impl-rel-ext* → (*Rv* → *bool-rel*)
→ ⟨⟨⟨*Rv*⟩*list-rel* $\times_r$ *Rv*⟩*option-rel*⟩*nres-rel*
apply (*intro fun-relI nres-relI*)
unfolding *Vid*
apply (*rule*
order-trans[*OF find-path0-code.refine*[*param-fo*, *THEN nres-relD*]],
assumption+
)
using *find-path0-impl find-path0-correct*
apply (*simp add*: *pw-le-iff refine-pw-simps*)
apply *blast*
done
lemmas *find-path0-autoref*[*autoref-rules*] = *find-path0-autoref-aux*[*OF PREFER-id-D*]

schematic-goal *find-path0-restr-code*:
fixes *vis-rel* :: (*'v*×*'v*) *set* ⇒ (*'visi*×*'v set*) *set*
notes [*autoref-rel-intf*] = *REL-INTFI*[*of vis-rel i-set* **for** *I*]
assumes [*autoref-rules*]: (*op-vis-insert*, *insert*)∈*Id* → ⟨*Id*⟩*vis-rel* → ⟨*Id*⟩*vis-rel*
assumes [*autoref-rules*]: (*op-vis-memb*, (∈))∈*Id* → ⟨*Id*⟩*vis-rel* → *bool-rel*
assumes [*autoref-rules*]:
(*Gi*, *G*) ∈ ⟨*Rm*, *Id*⟩*g-impl-rel-ext*
(*Pi*,*P*)∈*Id* → *bool-rel*
(*Ri*,*R*)∈⟨*Id*⟩*vis-rel*
shows (*nres-of* (*?c*::*?'c dres*),
find-path0-restr-impl
G
P
(*R*::$:_r$⟨*Id*⟩*vis-rel*)) ∈ *?R*
unfolding *find-path0-restr-impl-def*[*abs-def*] *DFS-code-unfold ssnos-unfolds*
unfolding *if-cancel not-not comp-def nres-monad-laws*
using [[*autoref-trace-failed-id*]]
apply (*autoref-monadic* (*trace*))
done

concrete-definition *find-path0-restr-code* **uses** *find-path0-restr-code*
export-code *find-path0-restr-code* **checking** *SML*

lemma *find-path0-restr-autoref-aux*:
assumes *1*: (*op-vis-insert*, *insert*)∈*Rv* → ⟨*Rv*⟩*vis-rel* → ⟨*Rv*⟩*vis-rel*
assumes *2*: (*op-vis-memb*, (∈))∈*Rv* → ⟨*Rv*⟩*vis-rel* → *bool-rel*
assumes *Vid*: *Rv* = *Id*
shows (λ *G P R*. *nres-of* (*find-path0-restr-code op-vis-insert op-vis-memb G P*

R),
  find-path0-restr-spec)
  $\in \langle Rm,\ Rv\rangle$*g-impl-rel-ext* $\rightarrow$ ($Rv \rightarrow$ *bool-rel*) $\rightarrow \langle Rv\rangle$*vis-rel* $\rightarrow$
  $\langle\langle\langle Rv\rangle$*vis-rel*, $\langle Rv\rangle$*list-rel* $\times_r$ $Rv\rangle$*sum-rel*$\rangle$*nres-rel*
 apply (*intro fun-relI nres-relI*)
 unfolding *Vid*
 apply (*rule*
  order-trans[*OF find-path0-restr-code.refine*[*OF 1*[*unfolded Vid*] *2*[*unfolded Vid*], *param-fo*, *THEN nres-relD*]]
)
 apply *assumption+*
 using *find-path0-restr-impl find-path0-restr-correct*
 apply (*simp add*: *pw-le-iff refine-pw-simps*)
 apply *blast*
 done
lemmas *find-path0-restr-autoref*[*autoref-rules*] = *find-path0-restr-autoref-aux*[*OF GEN-OP-D GEN-OP-D PREFER-id-D*]

schematic-goal *find-path1-restr-code*:
 fixes *vis-rel* :: ($'v\times'v$) *set* $\Rightarrow$ ($'visi\times'v$ *set*) *set*
 notes [*autoref-rel-intf*] = *REL-INTFI*[*of vis-rel i-set* **for** *I*]
 assumes [*autoref-rules*]: (*op-vis-insert*, *insert*)$\in$*Id* $\rightarrow \langle Id\rangle$*vis-rel* $\rightarrow \langle Id\rangle$*vis-rel*
 assumes [*autoref-rules*]: (*op-vis-memb*, ($\in$))$\in$*Id* $\rightarrow \langle Id\rangle$*vis-rel* $\rightarrow$ *bool-rel*
 assumes [*autoref-rules*]:
  (Gi, G) $\in \langle Rm,\ Id\rangle$*g-impl-rel-ext*
  (Pi,P)$\in$*Id* $\rightarrow$ *bool-rel*
  (Ri,R)$\in\langle Id\rangle$*vis-rel*
 shows (*nres-of ?c*,*find-path1-restr* G P R)
 $\in \langle\langle\langle Id\rangle$*vis-rel*, $\langle Id\rangle$*list-rel* $\times_r$ $Id\rangle$*sum-rel*$\rangle$*nres-rel*
 unfolding *find-path1-restr-def*[*abs-def*]
 using [[*autoref-trace-failed-id*]]
 apply (*autoref-monadic* (*trace*))
 done

concrete-definition *find-path1-restr-code* **uses** *find-path1-restr-code*
export-code *find-path1-restr-code* **checking** *SML*

lemma *find-path1-restr-autoref-aux*:
 assumes G: (*op-vis-insert*, *insert*)$\in V \rightarrow \langle V\rangle$*vis-rel* $\rightarrow \langle V\rangle$*vis-rel*
      (*op-vis-memb*, ($\in$))$\in V \rightarrow \langle V\rangle$*vis-rel* $\rightarrow$ *bool-rel*
 assumes *Vid*[*simp*]: V=*Id*
 shows (λ G P R. *nres-of* (*find-path1-restr-code op-vis-insert op-vis-memb* G P R),*find-path1-restr-spec*)
 $\in \langle Rm,\ V\rangle$*g-impl-rel-ext* $\rightarrow$ ($V \rightarrow$ *bool-rel*) $\rightarrow \langle V\rangle$*vis-rel* $\rightarrow$
  $\langle\langle\langle V\rangle$*vis-rel*, $\langle V\rangle$*list-rel* $\times_r$ $V\rangle$*sum-rel*$\rangle$*nres-rel*

proof −
 note *find-path1-restr-code.refine*[*OF G*[*simplified*], *param-fo*, *THEN nres-relD*]
 also note *find-path1-restr-correct*

finally show *?thesis* **by** (*force intro*!: *nres-relI*)
qed

lemmas *find-path1-restr-autoref*[*autoref-rules*] = *find-path1-restr-autoref-aux*[*OF GEN-OP-D GEN-OP-D PREFER-id-D*]

schematic-goal *find-path1-code*:
assumes *Vid*: $V = (Id :: 'a :: hashable\ rel)$
assumes [*unfolded Vid*,*autoref-rules*]:
$(Gi, G) \in \langle Rm, V\rangle g\text{-}impl\text{-}rel\text{-}ext$
$(Pi,P) \in V \to bool\text{-}rel$
notes [*autoref-tyrel*] = *TYRELI*[**where** $R=\langle(Id::('a\times'a::hashable)set)\rangle dflt\text{-}ahs\text{-}rel$]
shows (*nres-of ?c*,*find-path1 G P*)
$\in \langle\langle\langle V\rangle list\text{-}rel \times_r V\rangle option\text{-}rel\rangle nres\text{-}rel$
unfolding *find-path1-def*[*abs-def*] *Vid*
using [[*autoref-trace-failed-id*]]
apply (*autoref-monadic* (*trace*))
done
concrete-definition *find-path1-code* **uses** *find-path1-code*

export-code *find-path1-code* **checking** *SML*

lemma *find-path1-code-autoref-aux*:
assumes *Vid*: $V = (Id :: 'a :: hashable\ rel)$
shows (λ *G P*. *nres-of* (*find-path1-code G P*), *find-path1-spec*)
$\in \langle Rm, V\rangle g\text{-}impl\text{-}rel\text{-}ext \to (V \to bool\text{-}rel) \to \langle\langle\langle V\rangle list\text{-}rel \times_r V\rangle option\text{-}rel\rangle nres\text{-}rel$
proof −
note *find-path1-code.refine*[*OF Vid*, *param-fo*, *THEN nres-relD*, *simplified*]
also note *find-path1-correct*
finally show *?thesis* **by** (*force intro*!: *nres-relI*)
qed

lemmas *find-path1-autoref*[*autoref-rules*] = *find-path1-code-autoref-aux*[*OF PREFER-id-D*]

2.2.7 Conclusion

We have synthesized an efficient implementation for an algorithm to find a path to a reachable node that satisfies a predicate. The algorithm comes in four variants, with and without empty path, and with and without node restriction.

We have set up the Autoref tool, to insert this algorithms for the following specifications:

- *find-path0-spec G P* — find path to node that satisfies *P*.
- *find-path1-spec G P* — find non-empty path to node that satisfies *P*.

- *find-path0-restr-spec* $G\ P\ R$ — find path, with nodes from R already searched.
- *find-path1-restr-spec* — find non-empty path, with nodes from R already searched.

thm *find-path0-autoref*
thm *find-path1-autoref*
thm *find-path0-restr-autoref*
thm *find-path1-restr-autoref*

end

2.3 Set of Reachable Nodes

theory *Reachable-Nodes*
imports *../DFS-Framework*
CAVA-Automata.Digraph-Impl
../Misc/Impl-Rev-Array-Stack
begin

This theory provides a re-usable algorithm to compute the set of reachable nodes in a graph.

2.3.1 Preliminaries

lemma *gen-obtain-finite-set*:
assumes F: *finite* S
assumes E: $(e,\{\})\in\langle R\rangle Rs$
assumes I: $(i,insert)\in R\to\langle R\rangle Rs\to\langle R\rangle Rs$
assumes EE: $\bigwedge x.\ x\in S \Longrightarrow \exists\, xi.\ (xi,x)\in R$
shows $\exists\, Si.\ (Si,S)\in\langle R\rangle Rs$
proof −
define S' **where** $S' = S$

have $S\subseteq S'$ **by** (*simp add*: S'*-def*)
from F *this* **show** $(\exists\, Si.\ (Si,S)\in\langle R\rangle Rs)$
proof (*induction*)
case *empty* **thus** *?case*
using E **by** (*blast*)
next
case (*insert x S*)
then obtain *xi Si* **where** *1*: $(Si,S)\in\langle R\rangle Rs$ **and** *2*: $(xi,x)\in R$
using *EE* **unfolding** S'*-def* **by** *blast*
from I[*THEN fun-relD*, *OF 2*, *THEN fun-relD*, *OF 1*] **show** *?case* **..**
qed
qed

lemma *obtain-finite-ahs*: *finite S* $\Longrightarrow$ $\exists$ *x*. (*x*,*S*)$\in$$\langle$*Id*$\rangle$*dflt-ahs-rel*
apply (*erule gen-obtain-finite-set*)
apply *autoref*
apply *autoref*
by *blast*

2.3.2 Framework Instantiation

definition *unit-parametrization* $\equiv$ *dflt-parametrization* (λ-. ()) (*RETURN* ())

lemmas *unit-parametrization-simp*[*simp*, *DFS-code-unfold*] =
dflt-parametrization-simp[*mk-record-simp*, *OF*, *OF unit-parametrization-def*]

interpretation *unit-dfs*: *param-DFS-defs* **where** *param*=*unit-parametrization* **for**
G **.**

locale *unit-DFS* = *param-DFS G unit-parametrization* **for** *G* :: ('*v*, '*more*) *graph-rec-scheme*
begin
sublocale *DFS G unit-parametrization*
by *unfold-locales simp-all*
end

lemma *unit-DFSI*[*Pure.intro?*, *intro?*]:
assumes *fb-graph G*
shows *unit-DFS G*
proof −
interpret *fb-graph G* **by** *fact*
show *?thesis* **by** *unfold-locales*
qed

definition *find-reachable G* $\equiv$ *do* {
ASSERT (*fb-graph G*);
s $\leftarrow$ *unit-dfs.it-dfs TYPE*('*a*) *G*;
RETURN (*dom* (*discovered s*))
}

definition *find-reachableT G* $\equiv$ *do* {
ASSERT (*fb-graph G*);
s $\leftarrow$ *unit-dfs.it-dfsT TYPE*('*a*) *G*;
RETURN (*dom* (*discovered s*))
}

2.3.3 Correctness

context *unit-DFS* **begin**
lemma *find-reachable-correct*: *find-reachable G* $\leq$ *SPEC* (λ*r*. *r* = *reachable*)
unfolding *find-reachable-def*

apply (*refine-vcg order-trans*[*OF it-dfs-correct*])
apply *unfold-locales*
apply *clarify*
apply (*drule* (*1*) *DFS-invar.nc-discovered-eq-reachable*)
by *auto*

lemma *find-reachableT-correct*:
finite reachable $\Longrightarrow$ *find-reachableT G* $\leq$ *SPEC* ($\lambda r.\ r = reachable$)
unfolding *find-reachableT-def*
apply (*refine-vcg order-trans*[*OF it-dfsT-correct*])
apply *unfold-locales*
apply *clarify*
apply (*drule* (*1*) *DFS-invar.nc-discovered-eq-reachable*)
by *auto*
end

context *unit-DFS* **begin**

sublocale *simple-impl G unit-parametrization unit-parametrization unit-rel*
apply *unfold-locales*
apply (*clarsimp simp*: *simple-state-rel-def*) []
by *auto*

lemmas *impl-refine* = *simple-tailrecT-refine simple-tailrec-refine simple-rec-refine*
end

interpretation *unit-simple-impl*:
simple-impl-defs G unit-parametrization unit-parametrization
for *G* **.**

term *unit-simple-impl.tailrec-impl* **term** *unit-simple-impl.rec-impl*

definition [*DFS-code-unfold*]: *find-reachable-impl G* ≡ *do* {
ASSERT (*fb-graph G*);
s ← *unit-simple-impl.tailrec-impl TYPE*(*'a*) *G*;
RETURN (*simple-state.visited s*)
}

definition [*DFS-code-unfold*]: *find-reachable-implT G* ≡ *do* {
ASSERT (*fb-graph G*);
s ← *unit-simple-impl.tailrec-implT TYPE*(*'a*) *G*;
RETURN (*simple-state.visited s*)
}

definition [*DFS-code-unfold*]: *find-reachable-rec-impl G* ≡ *do* {
ASSERT (*fb-graph G*);
s ← *unit-simple-impl.rec-impl TYPE*(*'a*) *G*;
RETURN (*visited s*)

}

lemma *find-reachable-impl-refine*:
  find-reachable-impl $G \leq \Downarrow Id$ (*find-reachable* G)
  unfolding *find-reachable-impl-def find-reachable-def*
  apply (*refine-vcg unit-DFS.impl-refine*)
  apply (*simp-all add*: *unit-DFSI simple-state-rel-def*)
  done

lemma *find-reachable-implT-refine*:
  find-reachable-implT $G \leq \Downarrow Id$ (*find-reachableT* G)
  unfolding *find-reachable-implT-def find-reachableT-def*
  apply (*refine-vcg unit-DFS.impl-refine*)
  apply (*simp-all add*: *unit-DFSI simple-state-rel-def*)
  done

lemma *find-reachable-rec-impl-refine*:
  find-reachable-rec-impl $G \leq \Downarrow Id$ (*find-reachable* G)
  unfolding *find-reachable-rec-impl-def find-reachable-def*
  apply (*refine-vcg unit-DFS.impl-refine*)
  apply (*simp-all add*: *unit-DFSI simple-state-rel-def*)
  done

2.3.4 Synthesis of Executable Implementation

schematic-goal *find-reachable-impl*:
  defines $V \equiv Id :: ('v \times 'v::hashable)\ set$
  assumes [*unfolded V-def*,*autoref-rules*]:
    $(Gi,\ G) \in \langle Rm,\ V\rangle g\text{-}impl\text{-}rel\text{-}ext$
  notes [*unfolded V-def*,*autoref-tyrel*] =
    TYRELI[**where** $R{=}\langle V\rangle dflt\text{-}ahs\text{-}rel$]
    TYRELI[**where** $R{=}\langle V \times_r \langle V\rangle list\text{-}set\text{-}rel\rangle ras\text{-}rel$]
  shows *nres-of* $(?c::?'c\ dres) \leq \Downarrow ?R$ (*find-reachable-impl* G)
  unfolding *if-cancel DFS-code-unfold ssnos-unfolds*
  using [[*autoref-trace-failed-id*, *goals-limit=1*]]
  apply (*autoref-monadic* (*trace*))
  done
concrete-definition *find-reachable-code* **uses** *find-reachable-impl*
export-code *find-reachable-code* **checking** *SML*

lemma *find-reachable-code-correct*:
  assumes *1*: *fb-graph* G
  assumes *2*: $(Gi,\ G) \in \langle Rm,\ Id\rangle g\text{-}impl\text{-}rel\text{-}ext$
  assumes *4*: *find-reachable-code* $Gi = dRETURN\ r$
  shows $(r,\ (g\text{-}E\ G)^*\ ``\ g\text{-}V0\ G) \in \langle Id\rangle dflt\text{-}ahs\text{-}rel$
proof −
  from *1* **interpret** *unit-DFS* **by** *rule*
  note *find-reachable-code.refine*[*OF 2*]

also note *find-reachable-impl-refine*
also note *find-reachable-correct*
finally show *?thesis* **using** *1 4* **by** (*auto simp*: *RETURN-RES-refine-iff*)
qed

schematic-goal *find-reachable-implT*:
fixes V :: ('*vi*×'*v*) *set*
assumes [*autoref-ga-rules*]: *is-bounded-hashcode V eq bhc*
assumes [*autoref-rules*]: (*eq*,(=)) ∈ $V \to V \to$ *bool-rel*
assumes [*autoref-ga-rules*]: *is-valid-def-hm-size TYPE* ('*vi*) *sz*
assumes [*autoref-rules*]:
(*Gi*, *G*) ∈ ⟨*Rm*, *V*⟩*g-impl-rel-ext*
notes [*autoref-tyrel*] =
TYRELI[**where** *R*=⟨*V*⟩*ahs-rel bhc*]
TYRELI[**where** *R*=⟨$V \times_r$ ⟨*V*⟩*list-set-rel*⟩*ras-rel*]
shows *RETURN* (*?c*::*?'c*) ≤⇓*?R* (*find-reachable-implT G*)
unfolding *if-cancel DFS-code-unfold ssnos-unfolds*
using [[*autoref-trace-failed-id*, *goals-limit=1*]]
apply (*autoref-monadic* (*plain*,*trace*))
done
concrete-definition *find-reachable-codeT* **for** *eq bhc sz Gi*
uses *find-reachable-implT*
export-code *find-reachable-codeT* **checking** *SML*

lemma *find-reachable-codeT-correct*:
fixes V :: ('*vi*×'*v*) *set*
assumes *G*: *graph G*
assumes *FR*: *finite* ((*g-E G*)* ‘‘ *g-V0 G*)
assumes *BHC*: *is-bounded-hashcode V eq bhc*
assumes *EQ*: (*eq*,(=)) ∈ $V \to V \to$ *bool-rel*
assumes *VDS*: *is-valid-def-hm-size TYPE* ('*vi*) *sz*
assumes *2*: (*Gi*, *G*) ∈ ⟨*Rm*, *V*⟩*g-impl-rel-ext*
shows (*find-reachable-codeT eq bhc sz Gi*, (*g-E G*)* ‘‘ *g-V0 G*)∈⟨*V*⟩*ahs-rel bhc*
proof −
from *G* **interpret** *graph* **by** *this*
from *FR* **interpret** *fb-graph* **using** *fb-graphI-fr* **by** *simp*
interpret *unit-DFS* **by** *unfold-locales*
note *find-reachable-codeT.refine*[*OF BHC EQ VDS 2*]
also note *find-reachable-implT-refine*
also note *find-reachableT-correct*
finally show *?thesis* **using** *FR* **by** (*auto simp*: *RETURN-RES-refine-iff*)
qed

definition *all-unit-rel* :: (*unit* × '*a*) *set* **where** *all-unit-rel* ≡ *UNIV*

lemma *all-unit-refine*[*simp*]:
((),*x*)∈*all-unit-rel* **unfolding** *all-unit-rel-def* **by** *simp*

definition *unit-list-rel* :: ('c×'a) *set* ⇒ (*unit* × 'a *list*) *set*
  where [*to-relAPP*]: *unit-list-rel R* ≡ *UNIV*

lemma *unit-list-rel-refine*[*simp*]: ((),*y*)∈⟨*R*⟩*unit-list-rel*
  unfolding *unit-list-rel-def* **by** *auto*

lemmas [*autoref-rel-intf*] = *REL-INTFI*[*of unit-list-rel i-list*]

lemma [*autoref-rules*]:
  ((),[])∈⟨*R*⟩*unit-list-rel*
  (λ-. (),*tl*)∈⟨*R*⟩*unit-list-rel*→⟨*R*⟩*unit-list-rel*
  (λ- -. (),(#))∈*R* → ⟨*R*⟩*unit-list-rel*→⟨*R*⟩*unit-list-rel*
  by *auto*

schematic-goal *find-reachable-rec-impl*:
  defines *V* ≡ *Id* :: ('v × 'v::*hashable*) *set*
  assumes [*unfolded V-def*,*autoref-rules*]:
    (*Gi*, *G*) ∈ ⟨*Rm*, *V*⟩*g-impl-rel-ext*
  notes [*unfolded V-def*,*autoref-tyrel*] =
    TYRELI[**where** *R*=⟨*V*⟩*dflt-ahs-rel*]
  shows *nres-of* (*?c*::*?'c dres*) ≤⇓*?R* (*find-reachable-rec-impl G*)
  unfolding *unit-simple-impl.ssns-unfolds*
    DFS-code-unfold if-cancel if-False option.case
  using [[*autoref-trace-failed-id*, *goals-limit=1*]]
  apply (*autoref-monadic* (*trace*))
  done
concrete-definition *find-reachable-rec-code* **uses** *find-reachable-rec-impl*
prepare-code-thms *find-reachable-rec-code-def*
export-code *find-reachable-rec-code* **checking** *SML*

lemma *find-reachable-rec-code-correct*:
  assumes *1*: *fb-graph G*
  assumes *2*: (*Gi*, *G*) ∈ ⟨*Rm*, *Id*⟩*g-impl-rel-ext*
  assumes *4*: *find-reachable-rec-code Gi* = *dRETURN r*
  shows (*r*, (*g-E G*)* `` *g-V0 G*)∈⟨*Id*⟩*dflt-ahs-rel*
proof −
  from *1* **interpret** *unit-DFS* **by** *rule*
  note *find-reachable-rec-code.refine*[*OF 2*]
  also note *find-reachable-rec-impl-refine*
  also note *find-reachable-correct*
  finally show *?thesis* **using** *1 4* **by** (*auto simp*: *RETURN-RES-refine-iff*)
qed

definition [*simp*]: *op-reachable G* ≡ (*g-E G*)* `` *g-V0 G*
lemmas [*autoref-op-pat*] = *op-reachable-def*[*symmetric*]

context begin interpretation *autoref-syn* **.**

lemma *autoref-op-reachable*[*autoref-rules*]:
  fixes V :: $('vi \times 'v)$ *set*
  assumes *G*: *SIDE-PRECOND* (*graph G*)
  assumes *FR*: *SIDE-PRECOND* (*finite* ((*g-E G*)* `` *g-V0 G*))
  assumes *BHC*: *SIDE-GEN-ALGO* (*is-bounded-hashcode V eq bhc*)
  assumes *EQ*: *GEN-OP eq* (=) ($V \to V \to$ *bool-rel*)
  assumes *VDS*: *SIDE-GEN-ALGO* (*is-valid-def-hm-size TYPE* (*'vi*) *sz*)
  assumes *2*: (*Gi*, *G*) $\in$ ⟨*Rm*, *V*⟩*g-impl-rel-ext*
  shows (*find-reachable-codeT eq bhc sz Gi*,
    (*OP op-reachable* ::: ⟨*Rm*, *V*⟩*g-impl-rel-ext* $\to$ ⟨*V*⟩*ahs-rel bhc*)$*G*)$\in$⟨*V*⟩*ahs-rel bhc*
  using *assms*
  by (*simp add*: *find-reachable-codeT-correct*)

end

2.3.5 Conclusions

We have defined an efficient DFS-based implementation for *op-reachable*, and declared it to Autoref.

end

2.4 Find a Feedback Arc Set

theory *Feedback-Arcs*
imports
  ../DFS-Framework
  CAVA-Automata.Digraph-Impl
  Reachable-Nodes
begin

A feedback arc set is a set of edges that breaks all reachable cycles. In this theory, we define an algorithm to find a feedback arc set.

definition *is-fas* :: (*'v*, *'more*) *graph-rec-scheme* $\Rightarrow$ *'v rel* $\Rightarrow$ *bool* **where**
  is-fas G EC $\equiv \neg(\exists$ $u \in$ (*g-E G*)* `` *g-V0 G*. $(u, u) \in$ (*g-E G* $-$ *EC*)$^+$)

lemma *is-fas-alt*:
  is-fas G EC = *acyclic* ((*g-E G* $\cap$ ((*g-E G*)* `` *g-V0 G* $\times$ *UNIV*) $-$ *EC*))
  unfolding *is-fas-def acyclic-def*
proof (*clarsimp*, *safe*)
  fix *u*
  assume *A*: $(u,u) \in$ (*g-E G* $\cap$ (*g-E G*)* `` *g-V0 G* $\times$ *UNIV* $-$ *EC*)$^+$
  hence $(u,u)\in$(*g-E G* $-$ *EC*)$^+$ **by** (*rule trancl-mono*) *blast*
  moreover from *A* **have** $u \in$ (*g-E G*)* `` *g-V0 G* **by** (*cases rule*: *converse-tranclE*) *auto*
  moreover assume $\forall u\in$(*g-E G*)* `` *g-V0 G*. $(u, u) \notin$ (*g-E G* $-$ *EC*)$^+$

ultimately show *False* **by** *blast*
next
fix *u v0*
assume *1*: $v0 \in g\text{-}V0\ G$ **and** *2*: $(v0,u) \in (g\text{-}E\ G)^*$ **and** *3*: $(u,u) \in (g\text{-}E\ G - EC)^+$
have $(u, u) \in (Restr\ (g\text{-}E\ G - EC)\ ((g\text{-}E\ G)^*\ ``\ g\text{-}V0\ G))^+$
apply (*rule trancl-restrict-reachable*[*OF 3*, **where** $S=(g\text{-}E\ G)^*\ ``\ g\text{-}V0\ G$])
apply (*rule order-trans*[*OF - rtrancl-image-unfold-right*])
using *1 2* **by** *auto*
hence $(u, u) \in (g\text{-}E\ G \cap (g\text{-}E\ G)^*\ ``\ g\text{-}V0\ G \times UNIV - EC)^+$
by (*rule trancl-mono*) *auto*
moreover assume $\forall x.\ (x, x) \notin (g\text{-}E\ G \cap (g\text{-}E\ G)^*\ ``\ g\text{-}V0\ G \times UNIV - EC)^+$
ultimately show *False* **by** *blast*
qed

2.4.1 Instantiation of the DFS-Framework

record *'v fas-state* = *'v state* +
fas :: (*'v*×*'v*) *set*

lemma *fas-more-cong*: *state.more s* = *state.more s'* ⟹ *fas s* = *fas s'*
by (*cases s*, *cases s'*, *simp*)

lemma [*simp*]: *s*⦇ *state.more* := ⦇ *fas* = *foo* ⦈ ⦈ = *s* ⦇ *fas* := *foo* ⦈
by (*cases s*) *simp*

definition *fas-params* :: (*'v*,(*'v*,*unit*) *fas-state-ext*) *parameterization*
where *fas-params* ≡ *dflt-parametrization state.more*
(*RETURN* ⦇ *fas* = {} ⦈) ⦇
on-back-edge := λ*u v s*. *RETURN* ⦇ *fas* = *insert* (*u*,*v*) (*fas s*) ⦈
⦈
lemmas *fas-params-simp*[*simp*] =
gen-parameterization.simps[*mk-record-simp*, *OF fas-params-def*[*simplified*]]

interpretation *fas*: *param-DFS-defs* **where** *param*=*fas-params* **for** *G* **.**

Find feedback arc set

definition *find-fas G* ≡ *do* {
ASSERT (*graph G*);
ASSERT (*finite* $((g\text{-}E\ G)^*\ ``\ g\text{-}V0\ G)$);
s ← *fas.it-dfsT TYPE*(*'a*) *G*;
RETURN (*fas-state.fas s*)
}

locale *fas* =
param-DFS G fas-params
for *G* :: (*'v*, *'more*) *graph-rec-scheme*
+
assumes *finite-reachable*[*simp*, *intro*!]: *finite* $((g\text{-}E\ G)^*\ ``\ g\text{-}V0\ G)$

begin

  sublocale *DFS G fas-params*
    apply *unfold-locales*
    apply (*simp-all add*: *fas-params-def*)
    done

end

lemma *fasI*:
  assumes *graph G*
  assumes *finite* ((*g-E G*)* `` *g-V0 G*)
  shows *fas G*
proof −
  interpret *graph G* **by** *fact*
  interpret *fb-graph G* **by** (*rule fb-graphI-fr*[*OF assms*(*2*)])
  show *?thesis* **by** *unfold-locales fact*
qed

2.4.2 Correctness Proof

locale *fas-invar* = *DFS-invar* **where** *param* = *fas-params* + *fas*
begin

  lemma (**in** *fas*) *i-fas-eq-back*: *is-invar* (λ*s*. *fas-state.fas s* = *back-edges s*)
    apply (*induct rule*: *establish-invarI*)
    apply (*simp-all add*: *cond-def cong*: *fas-more-cong*)
    apply (*simp add*: *empty-state-def*)
    done
  lemmas *fas-eq-back* = *i-fas-eq-back*[*THEN make-invar-thm*]

  lemma *find-fas-correct-aux*:
    assumes *NC*: ¬*cond s*
    shows *is-fas G* (*fas-state.fas s*)
  proof −
    note [*simp*] = *fas-eq-back*

    from *nc-edges-covered*[*OF NC*] *edges-disjoint* **have**
      E ∩ *reachable* × *UNIV* − *back-edges s* = *tree-edges s* ∪ *cross-edges s*
      by *auto*
    with *tree-cross-acyclic* **show** *is-fas G* (*fas-state.fas s*)
      unfolding *is-fas-alt* **by** *simp*
  qed

end

lemma *find-fas-correct*:
  assumes *graph G*
  assumes *finite* ((*g-E G*)* `` *g-V0 G*)

shows *find-fas G* $\leq$ *SPEC* (*is-fas G*)
unfolding *find-fas-def*
proof (*refine-vcg le-ASSERTI order-trans*[*OF DFS.it-dfsT-correct*], *clarsimp-all*)
interpret *graph G* **by** *fact*
assume *finite* ((*g-E G*)* `` *g-V0 G*)
then interpret *fb-graph G* **by** (*rule fb-graphI-fr*)
interpret *fas* **by** *unfold-locales fact*
show *DFS G fas-params* **by** *unfold-locales*
next
fix *s*
assume *DFS-invar G fas-params s*
then interpret *DFS-invar G fas-params s* **.**
interpret *fas-invar G s* **by** *unfold-locales fact*
assume $\neg$*fas.cond TYPE*($'b$) *G s*
thus *is-fas G* (*fas-state.fas s*)
by (*rule find-fas-correct-aux*)
qed (*rule assms*)+

2.4.3 Implementation

record $'v$ *fas-state-impl* = $'v$ *simple-state* +
fas :: ($'v\times'v$) *set*

definition *fas-erel* $\equiv$ {
(⦇ *fas-state-impl.fas* = *f* ⦈, ⦇ *fas-state.fas* = *f*⦈) | *f*. *True* }
abbreviation *fas-rel* $\equiv$ ⟨*fas-erel*⟩*simple-state-rel*

definition *fas-params-impl*
:: ($'v$,$'v$ *fas-state-impl*,($'v$,*unit*) *fas-state-impl-ext*) *gen-parameterization*
where *fas-params-impl*
$\equiv$ *dflt-parametrization simple-state.more* (*RETURN* ⦇ *fas* = {} ⦈) ⦇
on-back-edge := $\lambda u\ v\ s$. *RETURN* ⦇ *fas* = *insert* (*u*,*v*) (*fas s*) ⦈⦈
lemmas *fas-params-impl-simp*[*simp*,*DFS-code-unfold*] =
gen-parameterization.simps[*mk-record-simp*, *OF fas-params-impl-def*[*simplified*]]

lemma *fas-impl*: (*si*,*s*)$\in$*fas-rel*
$\Longrightarrow$ *fas-state-impl.fas si* = *fas-state.fas s*
by (*cases si*, *cases s*, *simp add*: *simple-state-rel-def fas-erel-def*)

interpretation *fas-impl*: *simple-impl-defs G fas-params-impl fas-params*
for *G* **.**

term *fas-impl.tailrec-impl* **term** *fas-impl.tailrec-implT* **term** *fas-impl.rec-impl*

definition [*DFS-code-unfold*]: *find-fas-impl G* $\equiv$ *do* {
ASSERT (*graph G*);

$ASSERT\ (finite\ ((g\text{-}E\ G)^*\ ``\ g\text{-}V0\ G));$
$s \leftarrow fas\text{-}impl.tailrec\text{-}implT\ TYPE('a)\ G;$
$RETURN\ (fas\ s)$
}

context *fas* **begin**

sublocale *simple-impl G fas-params fas-params-impl fas-erel*
apply *unfold-locales*
apply (*intro fun-relI*, *clarsimp simp*: *simple-state-rel-def*, *parametricity*) []
apply (*auto simp*: *fas-erel-def fas-impl simple-state-rel-def*)
done

lemmas *impl-refine* = *simple-tailrec-refine simple-tailrecT-refine simple-rec-refine*
thm *simple-refine*
end

lemma *find-fas-impl-refine*: $find\text{-}fas\text{-}impl\ G \le \Downarrow Id\ (find\text{-}fas\ G)$
unfolding *find-fas-impl-def find-fas-def*
apply (*refine-vcg fas.impl-refine*)
apply (*simp-all add*: *fas-impl fasI*)
done

2.4.4 Synthesis of Executable Code

record ('si,'nsi,'fsi)*fas-state-impl'* = ('si,'nsi)*simple-state-impl* +
fas-impl :: 'fsi

definition [*to-relAPP*]: *fas-state-erel frel erel* ≡ {
$((\!|fas\text{-}impl = fi,\ \ldots = mi|\!),(\!|fas = f,\ \ldots = m|\!))\ |\ fi\ mi\ f\ m.$
$(fi,f)\in frel \wedge (mi,m)\in erel\}$

consts
$i\text{-}fas\text{-}state\text{-}ext :: interface \Rightarrow interface \Rightarrow interface$

lemmas [*autoref-rel-intf*] = *REL-INTFI*[*of fas-state-erel i-fas-state-ext*]

term *fas-update*
term *fas-state-impl'.fas-impl-update*
lemma [*autoref-rules*]:
fixes *ns-rel vis-rel frel erel*
defines $R \equiv \langle ns\text{-}rel,vis\text{-}rel,\langle frel,erel\rangle fas\text{-}state\text{-}erel\rangle ss\text{-}impl\text{-}rel$
shows
$(fas\text{-}state\text{-}impl'\text{-}ext,\ fas\text{-}state\text{-}impl\text{-}ext) \in frel \rightarrow erel \rightarrow \langle frel,erel\rangle fas\text{-}state\text{-}erel$
$(fas\text{-}impl,\ fas\text{-}state\text{-}impl.fas) \in R \rightarrow frel$
$(fas\text{-}state\text{-}impl'.fas\text{-}impl\text{-}update,\ fas\text{-}update) \in (frel \rightarrow frel) \rightarrow R \rightarrow R$
unfolding *fas-state-erel-def ss-impl-rel-def R-def*

by (*auto*, *parametricity*)

schematic-goal *find-fas-impl*:
fixes V :: $('vi \times 'v)$ *set*
assumes [*autoref-ga-rules*]: *is-bounded-hashcode* V *eq bhc*
assumes [*autoref-rules*]: $(eq,(=)) \in V \to V \to$ *bool-rel*
assumes [*autoref-ga-rules*]: *is-valid-def-hm-size* $TYPE$ $('vi)$ *sz*
assumes [*autoref-rules*]:
$(Gi,\ G) \in \langle Rm,\ V\rangle$*g-impl-rel-ext*
notes [*autoref-tyrel*] =
TYRELI[**where** $R=\langle V\rangle$*ahs-rel bhc*]
TYRELI[**where** $R=\langle V \times_r V\rangle$*ahs-rel* (*prod-bhc bhc bhc*)]
TYRELI[**where** $R=\langle V \times_r \langle V\rangle$*list-set-rel*$\rangle$*ras-rel*]
shows *RETURN* (*?c*::*?'c*) $\leq\Downarrow$*?R* (*find-fas-impl G*)
unfolding *DFS-code-unfold*
using [[*autoref-trace-failed-id*, *goals-limit=1*]]
apply (*autoref-monadic* (*trace*))
done
concrete-definition *find-fas-code* **for** *eq bhc sz Gi* **uses** *find-fas-impl*
export-code *find-fas-code* **checking** *SML*

thm *find-fas-code.refine*

lemma *find-fas-code-refine*[*refine*]:
fixes V :: $('vi \times 'v)$ *set*
assumes *is-bounded-hashcode* V *eq bhc*
assumes $(eq,(=)) \in V \to V \to$ *bool-rel*
assumes *is-valid-def-hm-size* $TYPE$ $('vi)$ *sz*
assumes *2*: $(Gi,\ G) \in \langle Rm,\ V\rangle$*g-impl-rel-ext*
shows *RETURN* (*find-fas-code eq bhc sz Gi*) $\leq \Downarrow(\langle V \times_r V\rangle$*ahs-rel* (*prod-bhc bhc bhc*)) (*find-fas G*)
proof −
note *find-fas-code.refine*[*OF assms*]
also note *find-fas-impl-refine*
finally show *?thesis* **.**
qed

context begin interpretation *autoref-syn* **.**

Declare this algorithm to Autoref:

theorem *find-fas-code-autoref*[*autoref-rules*]:
fixes V :: $('vi \times 'v)$ *set* **and** *bhc*
defines $RR \equiv \langle\langle V \times_r V\rangle$*ahs-rel* (*prod-bhc bhc bhc*)$\rangle$*nres-rel*
assumes *BHC*: *SIDE-GEN-ALGO* (*is-bounded-hashcode* V *eq bhc*)
assumes *EQ*: *GEN-OP eq* $(=)$ $(V \to V \to$ *bool-rel*)
assumes *VDS*: *SIDE-GEN-ALGO* (*is-valid-def-hm-size* $TYPE$ $('vi)$ *sz*)
assumes *2*: $(Gi,\ G) \in \langle Rm,\ V\rangle$*g-impl-rel-ext*
shows (*RETURN* (*find-fas-code eq bhc sz Gi*),
(*OP find-fas*

::: $\langle Rm,\ V\rangle$*g-impl-rel-ext* $\rightarrow$ *RR*)\$*G*)$\in$*RR*
unfolding *RR-def*
apply (*rule nres-relI*)
using *assms*
by (*simp add*: *find-fas-code-refine*)

end

2.4.5 Feedback Arc Set with Initialization

This algorithm extends a given set to a feedback arc set. It works in two steps:

1. Determine set of reachable nodes
2. Construct feedback arc set for graph without initial set

definition *find-fas-init* **where**
find-fas-init G FI $\equiv$ *do* {
ASSERT (*graph G*);
ASSERT (*finite* ((*g-E G*)* “ *g-V0 G*));
let nodes = (*g-E G*)* “ *g-V0 G*;
fas $\leftarrow$ *find-fas* (| *g-V* = *g-V G*, *g-E* = *g-E G* $-$ *FI*, *g-V0* = *nodes* |);
RETURN (*FI* $\cup$ *fas*)
}

The abstract idea: To find a feedback arc set that contains some set F2, we can find a feedback arc set for the graph with F2 removed, and then join with F2.

lemma *is-fas-join*: *is-fas G* (*F1* $\cup$ *F2*) $\longleftrightarrow$
is-fas (| *g-V* = *g-V G*, *g-E* = *g-E G* $-$ *F2*, *g-V0* = (*g-E G*)* “ *g-V0 G* |) *F1*
unfolding *is-fas-def*
apply (*auto simp*: *set-diff-diff-left Un-commute*)
by (*metis ImageI rtrancl-trans subsetCE rtrancl-mono*[*of g-E G* $-$ *F2 g-E G*, *OF Diff-subset*])

lemma *graphI-init*:
assumes *graph G*
shows *graph* (| *g-V* = *g-V G*, *g-E* = *g-E G* $-$ *FI*, *g-V0* = (*g-E G*)* “ *g-V0 G* |)
proof $-$
interpret *graph G* **by** *fact*
show *?thesis*
apply *unfold-locales*
using *reachable-V* **apply** *simp*
using *E-ss* **apply** *force*
done
qed

lemma *find-fas-init-correct*:
  assumes [*simp*, *intro*!]: *graph G*
  assumes [*simp*, *intro*!]: *finite* ((*g-E G*)* ‘‘ *g-V0 G*)
  shows *find-fas-init G FI* $\leq$ *SPEC* (λ*fas*. *is-fas G fas* $\wedge$ *FI* $\subseteq$ *fas*)
  unfolding *find-fas-init-def*
  apply (*refine-vcg order-trans*[*OF find-fas-correct*])
  apply *clarsimp-all*
  apply (*rule graphI-init*)
  apply *simp*
  apply (*rule finite-subset*[*rotated*], *rule assms*)
  apply (*metis Diff-subset Image-closed-trancl reachable-mono*
    rtrancl-image-unfold-right rtrancl-reflcl rtrancl-trancl-reflcl
    trancl-rtrancl-absorb)
  apply (*simp add*: *is-fas-join*[**where** *?F2.0=FI*] *Un-commute*)
  done

lemma *gen-cast-set*[*autoref-rules-raw*]:
  assumes *PRIO-TAG-GEN-ALGO*
  assumes *INS*: *GEN-OP ins Set.insert* (*Rk*$\rightarrow\langle$*Rk*$\rangle$*Rs2*$\rightarrow\langle$*Rk*$\rangle$*Rs2*)
  assumes *EM*: *GEN-OP emp* {} ($\langle$*Rk*$\rangle$*Rs2*)
  assumes *IT*: *SIDE-GEN-ALGO* (*is-set-to-list Rk Rs1 tsl*)
  shows (λ*s*. *gen-union* (λ*x*. *foldli* (*tsl x*)) *ins s emp*,*CAST*)
    $\in$ ($\langle$*Rk*$\rangle$*Rs1*) $\rightarrow$ ($\langle$*Rk*$\rangle$*Rs2*)
proof –
  note [*autoref-rules*] = *GEN-OP-D*[*OF INS*]
  note [*autoref-rules*] = *GEN-OP-D*[*OF EM*]
  note [*autoref-ga-rules*] = *SIDE-GEN-ALGO-D*[*OF IT*]
  have *1*: *CAST* = (λ*s*. *s* $\cup$ {}) **by** *auto*
  show *?thesis*
    unfolding *1*
    by *autoref*
qed

lemma *gen-cast-fun-set-rel*[*autoref-rules-raw*]:
  assumes *INS*: *GEN-OP mem* ($\in$) (*Rk*$\rightarrow\langle$*Rk*$\rangle$*Rs*$\rightarrow$*bool-rel*)
  shows (λ*s x*. *mem x s*,*CAST*) $\in$ ($\langle$*Rk*$\rangle$*Rs*) $\rightarrow$ ($\langle$*Rk*$\rangle$*fun-set-rel*)
proof –
  have *A*: $\bigwedge$*s*. (λ*x*. *x*$\in$*s*,*CAST s*) $\in$ *br Collect* (λ-. *True*)
    by (*auto simp*: *br-def*)

  show *?thesis*
    unfolding *fun-set-rel-def*
    apply *rule*
    apply *rule*
    defer
    apply (*rule A*)
    using *INS*[*simplified*]
    apply *parametricity*

done
qed

lemma *find-fas-init-impl-aux-unfolds*:
Let $(E^* `` V0) =$ *Let* $($*CAST* $(E^* `` V0))$
$(\lambda S.$ *RETURN* $(FI \cup S)) = (\lambda S.$ *RETURN* $(FI \cup$ *CAST* $S))$
by *simp-all*

schematic-goal *find-fas-init-impl*:
fixes $V :: ('vi \times 'v)$ *set* **and** *bhc*
assumes [*autoref-ga-rules*]: *is-bounded-hashcode V eq bhc*
assumes [*autoref-rules*]: $(eq,(=)) \in V \to V \to$ *bool-rel*
assumes [*autoref-ga-rules*]: *is-valid-def-hm-size TYPE* $('vi)$ *sz*
assumes [*autoref-rules*]:
$(Gi,\ G) \in \langle Rm,\ V\rangle$*g-impl-rel-ext*
$(FIi,FI) \in \langle V \times_r V\rangle$*fun-set-rel*
shows *RETURN* $(?c::?'c) \leq\Downarrow ?R$ (*find-fas-init G FI*)
unfolding *find-fas-init-def*
unfolding *find-fas-init-impl-aux-unfolds*
by (*autoref-monadic* (*plain*,*trace*))

concrete-definition *find-fas-init-code* **for** *eq bhc sz Gi FIi*
uses *find-fas-init-impl*
export-code *find-fas-init-code* **checking** *SML*

context begin interpretation *autoref-syn* **.**

The following theorem declares our implementation to Autoref:

theorem *find-fas-init-code-autoref*[*autoref-rules*]:
fixes $V :: ('vi \times 'v)$ *set* **and** *bhc*
defines $RR \equiv \langle V \times_r V\rangle$*fun-set-rel*
assumes *SIDE-GEN-ALGO* (*is-bounded-hashcode V eq bhc*)
assumes *GEN-OP eq* $(=)$ $(V \to V \to$ *bool-rel*$)$
assumes *SIDE-GEN-ALGO* (*is-valid-def-hm-size TYPE* $('vi)$ *sz*)
shows $(\lambda Gi\ FIi.$ *RETURN* (*find-fas-init-code eq bhc sz Gi FIi*),*find-fas-init*)
$\in \langle Rm,\ V\rangle$*g-impl-rel-ext* $\to RR \to \langle RR\rangle$*nres-rel*
unfolding *RR-def*
apply (*intro fun-relI nres-relI*)
using *assms*
by (*simp add*: *find-fas-init-code.refine*)

end

2.4.6 Conclusion

We have defined an algorithm to find a feedback arc set, and one to extend a given set to a feedback arc set. We have registered them to Autoref as implementations for *find-fas* and *find-fas-init*.

For preliminary refinement steps, you need the theorems *find-fas-correct* and *find-fas-init-correct*.

thm *find-fas-code-autoref find-fas-init-code-autoref*
thm *find-fas-correct* **thm** *find-fas-init-correct*

end

2.5 Nested DFS

theory *Nested-DFS*
imports *DFS-Find-Path*
begin

Nested DFS is a standard method for Buchi-Automaton emptiness check.

2.5.1 Auxiliary Lemmas

lemma *closed-restrict-aux*:
  assumes *CL*: $E``F \subseteq F \cup S$
  assumes *NR*: $E^* `` U \cap S = \{\}$
  assumes *SS*: $U \subseteq F$
  shows $E^* `` U \subseteq F$
  — Auxiliary lemma to show that nodes reachable from a finished node must be finished if, additionally, no stack node is reachable
proof *clarify*
  fix *u v*
  assume *A*: $(u,v) \in E^*$ $u \in U$
  hence *M*: $E^* ``\{u\} \cap S = \{\}$ $u \in F$ **using** *NR SS* **by** *blast+*

  from *A(1) M* **show** $v \in F$
    apply (*induct rule*: *converse-rtrancl-induct*)
    using *CL* **apply** (*auto dest*: *rtrancl-Image-advance-ss*)
    done

qed

2.5.2 Instantiation of the Framework

record *'v blue-dfs-state = 'v state +*
  lasso :: (*'v list* × *'v list*) *option*
  red :: *'v set*

type-synonym *'v blue-dfs-param* = (*'v*, (*'v,unit*) *blue-dfs-state-ext*) *parameterization*

lemma *lasso-more-cong*[*cong*]:*state.more s = state.more s'* ⟹ *lasso s = lasso s'*

by (*cases s*, *cases s'*) *simp*
lemma *red-more-cong*[*cong*]: *state.more s* = *state.more s'* ⟹ *red s* = *red s'*
by (*cases s*, *cases s'*) *simp*

lemma [*simp*]: *s*⦇ *state.more* := ⦇ *lasso* = *foo*, *red* = *bar* ⦈ ⦈ = *s* ⦇ *lasso* := *foo*, *red* := *bar* ⦈
by (*cases s*) *simp*

abbreviation *dropWhileNot v* ≡ *dropWhile* ((≠) *v*)
abbreviation *takeWhileNot v* ≡ *takeWhile* ((≠) *v*)

locale *BlueDFS-defs* = *graph-defs G*
for *G* :: (*'v*, *'more*) *graph-rec-scheme* +
fixes *accpt* :: *'v* ⇒ *bool*
begin

definition *blue s* ≡ *dom* (*finished s*) − *red s*
definition *cyan s* ≡ *set* (*stack s*)
definition *white s* ≡ − *dom* (*discovered s*)

abbreviation *red-dfs R ss x* ≡ *find-path1-restr-spec* (*G* ⦇ *g-V0* := {*x*} ⦈) *ss R*

definition *mk-blue-witness*
:: *'v blue-dfs-state* ⇒ *'v fpr-result* ⇒ (*'v*,*unit*) *blue-dfs-state-ext*
where
mk-blue-witness s redS ≡ *case redS of*
Inl R' ⇒ ⦇ *lasso* = *None*, *red* = (*R'* $\cancel{\cup\ red\ s}$) ⦈
| *Inr* (*vs*, *v*) ⇒ *let rs* = *rev* (*stack s*) *in*
⦇ *lasso* = *Some* (*rs*, *vs*@*dropWhileNot v rs*), *red* = *red s*⦈

definition *run-red-dfs*
:: *'v* ⇒ *'v blue-dfs-state* ⇒ (*'v*,*unit*) *blue-dfs-state-ext nres*
where
run-red-dfs u s ≡ *case lasso s of None* ⇒ *do* {
redS ← *red-dfs* (*red s*) (λ*x*. *x* = *u* ∨ *x* ∈ *cyan s*) *u*;
RETURN (*mk-blue-witness s redS*)
}
| - ⇒ *NOOP s*

Schwoon-Esparza extension

definition *se-back-edge u v s* ≡ *case lasso s of*
None ⇒
— it's a back edge, so *u* and *v* are both on stack
— we differentiate whether *u* or *v* is the 'culprit'
— to generate a better counter example
if accpt u then
let rs = *rev* (*tl* (*stack s*));
ur = *rs*;

```
          ul = u#dropWhileNot v rs
        in RETURN (|lasso = Some (ur,ul), red = red s|)
      else if accpt v then
        let rs = rev (stack s);
            vr = takeWhileNot v rs;
            vl = dropWhileNot v rs
        in RETURN (|lasso = Some (vr,vl), red = red s|)
      else NOOP s
  | - ⇒ NOOP s

  definition blue-dfs-params :: 'v blue-dfs-param
    where blue-dfs-params = (|
    on-init = RETURN (| lasso = None, red = {} |),
    on-new-root = λv0 s. NOOP s,
    on-discover = λu v s. NOOP s,
    on-finish = λu s. if accpt u then run-red-dfs u s else NOOP s,
    on-back-edge = se-back-edge,
    on-cross-edge = λu v s. NOOP s,
    is-break = λs. lasso s ≠ None |)

  schematic-goal blue-dfs-params-simps[simp]:
    on-init blue-dfs-params = ?OI
    on-new-root blue-dfs-params = ?ONR
    on-discover blue-dfs-params = ?OD
    on-finish blue-dfs-params = ?OF
    on-back-edge blue-dfs-params = ?OBE
    on-cross-edge blue-dfs-params = ?OCE
    is-break blue-dfs-params = ?IB
    unfolding blue-dfs-params-def gen-parameterization.simps
    by (rule refl)+


  sublocale param-DFS-defs G blue-dfs-params
    by unfold-locales

end

locale BlueDFS = BlueDFS-defs G accpt + param-DFS G blue-dfs-params
  for G :: ('v, 'more) graph-rec-scheme and accpt :: 'v ⇒ bool

lemma BlueDFSI:
  assumes fb-graph G
  shows BlueDFS G
proof −
  interpret fb-graph G by fact
  show ?thesis by unfold-locales
qed

locale BlueDFS-invar = BlueDFS +
```

DFS-invar **where** *param* = *blue-dfs-params*

context *BlueDFS-defs* **begin**

lemma *BlueDFS-invar-eq*[*simp*]:
shows *DFS-invar G blue-dfs-params s* $\longleftrightarrow$ *BlueDFS-invar G accpt s*
proof
assume *DFS-invar G blue-dfs-params s*
interpret *DFS-invar G blue-dfs-params s* **by** *fact*
show *BlueDFS-invar G accpt s* **by** *unfold-locales*
next
assume *BlueDFS-invar G accpt s*
then interpret *BlueDFS-invar G accpt s* **.**
show *DFS-invar G blue-dfs-params s* **by** *unfold-locales*
qed

end

2.5.3 Correctness Proof

context *BlueDFS* **begin**

definition *blue-basic-invar s* ≡
case lasso s of
None ⇒ *restr-invar E* (*red s*) (λ*x*. *x*∈*set* (*stack s*))
∧ *red s* ⊆ *dom* (*finished s*)
| *Some l* ⇒ *True*

lemma (**in** *BlueDFS-invar*) *red-DFS-precond-aux*:
assumes *BI*: *blue-basic-invar s*
assumes [*simp*]: *lasso s* = *None*
assumes *SNE*: *stack s* ≠ []
shows
fb-graph (*G* ⦇ *g-V0* := {*hd* (*stack s*)} ⦈)
and *fb-graph* (*G* ⦇ *g-E* := *E* ∩ *UNIV* × − *red s*, *g-V0* := {*hd* (*stack s*)} ⦈)
and *restr-invar E* (*red s*) (λ*x*. *x* ∈ *set* (*stack s*))
using *stack-reachable* ‹*stack s* ≠ []›
apply (*rule-tac fb-graph-subset*, *auto*) []
using *stack-reachable* ‹*stack s* ≠ []›
apply (*rule-tac fb-graph-subset*, *auto*) []

using *BI* **apply** (*simp add*: *blue-basic-invar-def*)
done

lemma (**in** *BlueDFS-invar*) *red-dfs-pres-bbi*:
assumes *BI*: *blue-basic-invar s*
assumes [*simp*]: *lasso s* = *None* **and** *SNE*: *stack s* ≠ []
assumes *pending s* ‘‘ {*hd* (*stack s*)} = {}
shows *run-red-dfs* (*hd* (*stack s*)) (*finish* (*hd* (*stack s*)) *s*) $\le_n$

SPEC (*λe*.
DFS-invar G blue-dfs-params (*finish* (*hd* (*stack s*)) *s*(|*state.more* := *e*|))
⟶ *blue-basic-invar* (*finish* (*hd* (*stack s*)) *s*(|*state.more* := *e*|)))
proof −
have [*simp*]: (*λx*. *x* = *hd* (*stack s*) ∨ *x* ∈ *cyan* (*finish* (*hd* (*stack s*)) *s*)) =
(*λx*. *x*∈*set* (*stack s*))
using ‹*stack s* ≠ []›
unfolding *finish-def cyan-def* **by** (*auto simp*: *neq-Nil-conv*)

show *?thesis*
unfolding *run-red-dfs-def*
apply *simp*
apply (*refine-vcg*)
apply *simp*
proof −
fix *fp1*
define *s′* **where** *s′* = *finish* (*hd* (*stack s*)) *s*
assume *FP-spec*:
find-path1-restr-pred (*G* (| *g-V0* := {*hd* (*stack s*)} |)) (*λx*. *x* ∈ *set* (*stack s*)) (*red s*) *fp1*
assume *BlueDFS-invar G accpt* (*s′*(|*state.more* := *mk-blue-witness s′ fp1* |))

then interpret *i*: *BlueDFS-invar G accpt* (*s′*(|*state.more* := *mk-blue-witness s′ fp1* |))
by *simp*

have [*simp*]:
red s′ = *red s*
discovered s′ = *discovered s*
dom (*finished s′*) = *insert* (*hd* (*stack s*)) (*dom* (*finished s*))
unfolding *s′-def finish-def* **by** *auto*

{
fix *R′*
assume [*simp*]: *fp1* = *Inl R′*
from *FP-spec*[*unfolded find-path1-restr-pred-def*, *simplified*]
have
R′FMT: *R′* = *red s* ∪ E^+ ‘‘ {*hd* (*stack s*)}
and *RI*: *restr-invar E R′* (*λx*. *x* ∈ *set* (*stack s*))
by *auto*

from *BI* **have** *red s* ⊆ *dom* (*finished s*)
unfolding *blue-basic-invar-def* **by** *auto*
also have E^+ ‘‘ {*hd* (*stack s*)} ⊆ *dom* (*finished s*)
proof (*intro subsetI*, *elim ImageE*, *simp*)
fix *v*
assume (*hd* (*stack s*),*v*)∈E^+

then obtain *u* **where** (*hd* (*stack s*),*u*)∈*E* **and** (*u*,*v*)∈E^*

by (*auto simp*: *trancl-unfold-left*)

from *RI* **have** *NR*: E^+ `` $\{hd\ (stack\ s)\} \cap set\ (stack\ s) = \{\}$
unfolding *restr-invar-def* **by** (*auto simp*: *R′FMT*)

with ‹$(hd\ (stack\ s),u) \in E$› **have** $u \notin set\ (stack\ s)$ **by** *auto*
with *i.finished-closed*[*simplified*] ‹$(hd\ (stack\ s),u) \in E$›
have *UID*: $u \in dom\ (finished\ s)$ **by** (*auto simp*: *stack-set-def*)

from *NR* ‹$(hd\ (stack\ s),u) \in E$› **have** *NR′*: E^* ``$\{u\} \cap set\ (stack\ s) = \{\}$
by (*auto simp*: *trancl-unfold-left*)

have *CL*: E `` $dom\ (finished\ s) \subseteq dom\ (finished\ s) \cup set\ (stack\ s)$
using *finished-closed discovered-eq-finished-un-stack*
by *simp*

from *closed-restrict-aux*[*OF CL NR′*] *UID*
have E^* `` $\{u\} \subseteq dom\ (finished\ s)$ **by** *simp*
with ‹$(u,v) \in E^*$› **show** $v \in dom\ (finished\ s)$ **by** *auto*
qed
finally (*sup-least*)
have $R' \subseteq dom\ (finished\ s) \wedge red\ s \subseteq dom\ (finished\ s)$
by (*simp add*: *R′FMT*)
} **note** *aux1* = *this*

show *blue-basic-invar* (s'⦇*state.more* := *mk-blue-witness* s' *fp1*⦈)
unfolding *blue-basic-invar-def mk-blue-witness-def*
apply (*simp split*: *option.splits sum.splits*)
apply (*intro allI conjI impI*)

using *FP-spec SNE*
apply (*auto*
simp: *s′-def blue-basic-invar-def find-path1-restr-pred-def*
simp: *restr-invar-def*
simp: *neq-Nil-conv*) []

apply (*auto dest*!: *aux1*) []
done
qed
qed

lemma *blue-basic-invar*: *is-invar blue-basic-invar*
proof (*induct rule*: *establish-invarI*)
case (*finish s*) **then interpret** *BlueDFS-invar* **where** *s*=*s* **by** *simp*

have [*simp*]: $(\lambda x.\ x = hd\ (stack\ s) \vee x \in cyan\ (finish\ (hd\ (stack\ s))\ s)) =$
$(\lambda x.\ x \in set\ (stack\ s))$
using ‹$stack\ s \neq []$›
unfolding *finish-def cyan-def* **by** (*auto simp*: *neq-Nil-conv*)

from *finish* **show** *?case*
  apply (*simp*)
  apply (*intro conjI impI*)
  apply (*rule leof-trans*[*OF red-dfs-pres-bbi*], *assumption+*, *simp*)

  apply (*auto simp*: *restr-invar-def blue-basic-invar-def neq-Nil-conv*) []
  done
qed (*auto simp*: *blue-basic-invar-def cond-def se-back-edge-def*
  simp: *restr-invar-def empty-state-def pred-defs*
  simp: *DFS-invar.discovered-eq-finished-un-stack*
  simp del: *BlueDFS-invar-eq*
  split: *option.splits*)

lemmas (**in** *BlueDFS-invar*) *s-blue-basic-invar*
  = *blue-basic-invar*[*THEN make-invar-thm*]

lemmas (**in** *BlueDFS-invar*) *red-DFS-precond*
  = *red-DFS-precond-aux*[*OF s-blue-basic-invar*]

sublocale *DFS G blue-dfs-params*
  apply *unfold-locales*

  apply (*clarsimp-all*
    simp: *se-back-edge-def run-red-dfs-def refine-pw-simps pre-on-defs*
    split: *option.splits*)

  unfolding *nofail-SPEC-iff*
  apply (*refine-vcg*)
    apply (*erule BlueDFS-invar.red-DFS-precond*, *auto*) []

    apply (*simp add*: *cyan-def finish-def*)
    apply (*erule BlueDFS-invar.red-DFS-precond*, *auto*) []

    apply (*rule TrueI*)
  done

end

context *BlueDFS-invar*
begin

context assumes [*simp*]: *lasso s* = *None*
begin
  lemma *red-closed*:
    E ‘‘ *red s* ⊆ *red s*
    using *s-blue-basic-invar*
    unfolding *blue-basic-invar-def restr-invar-def*
    by *simp*

lemma *red-stack-disjoint*:
  set (*stack s*) ∩ *red s* = {}
  using *s-blue-basic-invar*
  unfolding *blue-basic-invar-def restr-invar-def*
  by *auto*

lemma *red-finished*: *red s* ⊆ *dom* (*finished s*)
  using *s-blue-basic-invar*
  unfolding *blue-basic-invar-def*
  by *auto*

lemma *all-nodes-colored*: *white s* ∪ *blue s* ∪ *cyan s* ∪ *red s* = *UNIV*
  unfolding *white-def blue-def cyan-def*
  by (*auto simp*: *stack-set-def*)

lemma *colors-disjoint*:
  white s ∩ (*blue s* ∪ *cyan s* ∪ *red s*) = {}
  blue s ∩ (*white s* ∪ *cyan s* ∪ *red s*) = {}
  cyan s ∩ (*white s* ∪ *blue s* ∪ *red s*) = {}
  red s ∩ (*white s* ∪ *blue s* ∪ *cyan s*) = {}
  unfolding *white-def blue-def cyan-def*
  using *finished-discovered red-finished*
  unfolding *stack-set-def*
  by *blast+*

end

lemma (**in** *BlueDFS*) *i-no-accpt-cyle-in-finish*:
  is-invar (λ*s*. *lasso s* = *None* ⟶ (∀ *x*. *accpt x* ∧ *x* ∈ *dom* (*finished s*) ⟶ (*x*,*x*) ∉ E^+))
proof (*induct rule*: *establish-invarI*)
  case (*finish s s′ u*) **then interpret** *BlueDFS-invar* **where** *s*=*s* **by** *simp*

  let *?onstack* = λ*x*. *x*∈*set* (*stack s*)
  let *?rE* = *rel-restrict E* (*red s*)

  from *finish* **obtain** *sh st* **where** [*simp*]: *stack s* = *sh*#*st*
    by (*auto simp*: *neq-Nil-conv*)

  have *1*: *g-E* (*G* ⦇ *g-V0* := {*hd* (*stack s*)} ⦈) = *E* **by** *simp*

  {
    fix *R′*::′*v set*
    let *?R′* = *R′* ~~∪ *red s*~~
    let *?s* = *s′*⦇ *lasso* := *None*, *red* := *?R′*⦈

    assume ⋀*v*. (*hd* (*stack s*), *v*) ∈ *?rE*$^+$ ⟹ ¬ *?onstack v*

```
    and accpt: accpt u
    and NL[simp]: lasso s = None
    hence no-hd-cycle: (hd (stack s), hd (stack s)) ∉ ?rE⁺
      by auto

    from finish have stack s ≠ [] by simp
    from hd-in-set[OF this] have hd (stack s) ∉ red s
      using red-stack-disjoint
      by auto
    hence (hd (stack s),hd (stack s)) ∉ E⁺
      using no-hd-cycle rel-restrict-tranclI red-closed[OF NL]
      by metis
    with accpt finish have
      ∀ x. accpt x ∧ x ∈ dom (finished ?s) ⟶ (x,x) ∉ E⁺
      by auto
  } with finish have
    red-dfs (red s) ?onstack (hd (stack s))
      ≤ SPEC (λx. ∀ R. x = Inl R ⟶
            DFS-invar G blue-dfs-params (lasso-update Map.empty s'⦇red := R ~~∪
red s~~⦈) ⟶
          (∀ x. accpt x ∧ x∈dom (finished s') ⟶ (x, x) ∉ E⁺))
    apply −
    apply (rule find-path1-restr-spec-rule, intro conjI)

    apply (rule red-DFS-precond, simp-all) []
    unfolding 1
    apply (rule red-DFS-precond, simp-all) []

    apply (auto simp: find-path1-restr-pred-def restr-invar-def)
    done
  note aux = leof-trans[OF this[simplified,THEN leof-lift]]

  note [refine-vcg del] = find-path1-restr-spec-rule

  from finish show ?case
    apply simp
    apply (intro conjI impI)
      unfolding run-red-dfs-def mk-blue-witness-def cyan-def
      apply clarsimp
      apply (refine-vcg aux)
      apply (auto split: sum.splits)
    done
next
  case back-edge thus ?case
    by (simp add: se-back-edge-def split: option.split)
qed simp-all

lemma no-accpt-cycle-in-finish:
  ⟦lasso s = None; accpt v; v ∈ dom (finished s)⟧ ⟹ (v,v) ∉ E⁺
```

using *i-no-accpt-cyle-in-finish*[*THEN make-invar-thm*]
by *blast*

end

context *BlueDFS*
begin
definition *lasso-inv* **where**
$lasso\text{-}inv\ s \equiv \forall\, pr\ pl.\ lasso\ s = Some\ (pr,pl) \longrightarrow$
$pl \neq []$
$\wedge\ (\exists\, v0{\in}V0.\ path\ E\ v0\ pr\ (hd\ pl))$
$\wedge\ accpt\ (hd\ pl)$
$\wedge\ path\ E\ (hd\ pl)\ pl\ (hd\ pl)$

lemma (**in** *BlueDFS-invar*) *se-back-edge-lasso-inv*:
assumes *b-inv*: *lasso-inv s*
and *ne*: $stack\ s \neq []$
and *R*: $lasso\ s = None$
and *p*:$(hd\ (stack\ s),\ v) \in pending\ s$
and *v*: $v \in dom\ (discovered\ s)\ v \notin dom\ (finished\ s)$
and *s'*: $s' = back\text{-}edge\ (hd\ (stack\ s))\ v\ (s(\!|pending := pending\ s - \{(u,v)\}|\!))$
shows $se\text{-}back\text{-}edge\ (hd\ (stack\ s))\ v\ s'$
$\leq SPEC\ (\lambda e.\ DFS\text{-}invar\ G\ blue\text{-}dfs\text{-}params\ (s'(\!|state.more := e|\!)) \longrightarrow$
$lasso\text{-}inv\ (s'(\!|state.more := e|\!)))$
proof −

from *v stack-set-def* **have** *v-in*: $v \in set\ (stack\ s)$ **by** *simp*
from *p* **have** *uv-edg*: $(hd\ (stack\ s),\ v) \in E$ **by** (*auto dest*: *pendingD*)

{
assume *accpt*: *accpt* (*hd* (*stack s*))
let $?ur = rev\ (tl\ (stack\ s))$
let $?ul = hd\ (stack\ s)\#dropWhileNot\ v\ (rev\ (tl\ (stack\ s)))$
let $?s = s'(\!|lasso := Some\ (?ur,\ ?ul),\ red := red\ s|\!)$

assume *DFS-invar G blue-dfs-params ?s*

have [*simp*]: $stack\ ?s = stack\ s$
by (*simp add*: *s'*)

have *hd-ul*[*simp*]: $hd\ ?ul = hd\ (stack\ s)$ **by** *simp*

have $?ul \neq []$ **by** *simp*

moreover have *P*:$\exists\, v0{\in}V0.\ path\ E\ v0\ ?ur\ (hd\ ?ul)$
using *stack-is-path*[*OF ne*]
by *auto*
moreover
from *accpt* **have** *accpt* (*hd ?ul*) **by** *simp*

```
  moreover have path E (hd ?ul) ?ul (hd ?ul)
  proof (cases v = hd (stack s))
    case True
    with distinct-hd-tl stack-distinct have ul: ?ul = [hd (stack s)]
      by force
    from True uv-edg show ?thesis
      by (subst ul)+ (simp add: path1)
  next
    case False with v-in ne have v ∈ set ?ur
      by (auto simp add: neq-Nil-conv)
    with P show ?thesis
      by (fastforce intro: path-prepend
                    dropWhileNot-path[where p=?ur]
                    uv-edg)
  qed

  ultimately have lasso-inv ?s by (simp add: lasso-inv-def)
}

moreover
{
  assume accpt: accpt v
  let ?vr = takeWhileNot v (rev (stack s))
  let ?vl = dropWhileNot v (rev (stack s))
  let ?s = s′(|lasso := Some(?vr, ?vl), red := red s|)

  assume DFS-invar G blue-dfs-params ?s

  have [simp]: stack ?s = stack s
    by (simp add: s′)

  from ne v-in have hd-vl[simp]: hd ?vl = v
    by (induct (stack s) rule: rev-nonempty-induct) auto

  from v-in have ?vl ≠ [] by simp

  moreover from hd-succ-stack-is-path[OF ne] uv-edg have
    P: ∃ v0∈V0. path E v0 (rev (stack s)) v
    by auto
  with ne v-in have ∃ v0∈V0. path E v0 ?vr (hd ?vl)
    by (force intro: takeWhileNot-path)

  moreover from accpt have accpt (hd ?vl) by simp

  moreover from P ne v-in have path E (hd ?vl) ?vl (hd ?vl)
    by (force intro: dropWhileNot-path)

  ultimately have lasso-inv ?s by (simp add: lasso-inv-def)
```

}

moreover
{
assume ¬ *accpt* (*hd* (*stack s*)) ¬ *accpt v*
let *?s* = *s′*(|*state.more* := *state.more s′*|)

assume *DFS-invar G blue-dfs-params ?s*

from *assms* **have** *lasso-inv ?s*
by (*auto simp add*: *lasso-inv-def*)
}

ultimately show *?thesis*
using *R s′*
unfolding *se-back-edge-def*
by (*auto split*: *option.splits*)
qed

lemma *lasso-inv*:
is-invar lasso-inv
proof (*induct rule*: *establish-invarI*)
case (*finish s s′ u*) **then interpret** *BlueDFS-invar* **where** *s*=*s* **by** *simp*

let *?onstack* = $\lambda x.\ x \in$ *set* (*stack s*)
let *?rE* = *rel-restrict E* (*red s*)
let *?revs* = *rev* (*tl* (*stack s*))

note *ne* = ‹*stack s* ≠ []›

note [*simp*] = ‹*u*=*hd* (*stack s*)›

from *finish* **have** [*simp*]:
⋀*x*. *x* = *hd* (*stack s*) ∨ *x* ∈ *set* (*stack s′*) ⟷ *x*∈*set* (*stack s*)
red s′ = *red s*
lasso s′ = *lasso s*
by (*auto simp*: *neq-Nil-conv*)

{
fix *v vs*
let *?cyc* = *vs* @ *dropWhileNot v ?revs*
let *?s* = *s′*(|*lasso* := *Some* (*?revs*, *?cyc*), *red* := *red s*|)

assume *DFS-invar G blue-dfs-params ?s*
and *vs*: *vs* ≠ [] *path ?rE* (*hd* (*stack s*)) *vs v*
and *v*: *?onstack v*

```
      and accpt: accpt (hd (stack s))
    from vs have P: path E (hd (stack s)) vs v
      by (metis path-mono rel-restrict-sub)

    have hds[simp]: hd vs = hd (stack s) hd ?cyc = hd (stack s)
      using vs path-hd
      by simp-all

    from vs have ?cyc ≠ [] by simp

    moreover have P0: ∃ v0∈V0. path E v0 ?revs (hd ?cyc)
      using stack-is-path[OF ne]
      by auto

    moreover from accpt have accpt (hd ?cyc) by simp

    moreover have path E (hd ?cyc) ?cyc (hd ?cyc)
    proof (cases tl (stack s) = [])
      case True with ne last-stack-in-V0 obtain v0 where v0 ∈ V0
        and [simp]: stack s = [v0]
        by (auto simp: neq-Nil-conv)
      with v True finish have [simp]: v = v0 by simp

      from True P show ?thesis by simp
    next
      case False note tl-ne = this

      show ?thesis
      proof (cases v = hd (stack s))
        case True hence v ∉ set ?revs
          using ne stack-distinct by (auto simp: neq-Nil-conv)
        hence ?cyc = vs by fastforce
        with P True show ?thesis by (simp del: dropWhile-eq-Nil-conv)
      next
        case False with finish v have v ∈ set ?revs
          by (auto simp: neq-Nil-conv)
        with tl-ne False P0  show ?thesis
          by (force intro: path-conc[OF P]
              dropWhileNot-path[where p=?revs])
      qed
    qed

    ultimately have lasso-inv ?s by (simp add: lasso-inv-def)
  }
  hence accpt (hd (stack s)) ⟶ lasso s = None ⟶
        red-dfs (red s) ?onstack (hd (stack s)) ≤ SPEC (λrs. ∀ vs v.
          rs = Inr (vs,v) ⟶
                DFS-invar G blue-dfs-params (s'(|lasso := Some (?revs, vs @
dropWhileNot v ?revs), red:= red s|)) ⟶
```

```
                lasso-inv (s'(|lasso := Some (?revs, vs @ dropWhileNot v ?revs),
red:=red s|)))
      apply clarsimp
      apply (rule find-path1-restr-spec-rule, intro conjI)
      apply (rule red-DFS-precond, simp-all add: ne) []
      apply (simp, rule red-DFS-precond, simp-all add: ne) []

      using red-stack-disjoint ne

      apply clarsimp
      apply rprems
      apply (simp-all add: find-path1-restr-pred-def restr-invar-def)
      apply (fastforce intro: path-restrict-tl rel-restrictI)
      done
    note aux1 = this[rule-format,THEN leof-lift]

    show ?case
      apply simp
      unfolding run-red-dfs-def mk-blue-witness-def cyan-def

      apply (simp
        add: run-red-dfs-def mk-blue-witness-def cyan-def
        split: option.splits)
      apply (intro conjI impI)
      apply (refine-vcg leof-trans[OF aux1])
      using finish
      apply (auto simp add: lasso-inv-def split: sum.split)
      done
  next
    case (back-edge s s' u v) then interpret BlueDFS-invar where s=s by simp

    from back-edge se-back-edge-lasso-inv[THEN leof-lift] show ?case
      by auto
  qed (simp-all add: lasso-inv-def empty-state-def)
end

context BlueDFS-invar
begin
  lemmas s-lasso-inv = lasso-inv[THEN make-invar-thm]

  lemma
    assumes lasso s = Some (pr,pl)
    shows loop-nonempty: pl ≠ []
    and accpt-loop: accpt (hd pl)
    and loop-is-path: path E (hd pl) pl (hd pl)
    and loop-reachable: ∃ v0∈V0. path E v0 pr (hd pl)
    using assms s-lasso-inv
    by (simp-all add: lasso-inv-def)
```

```
lemma blue-dfs-correct:
  assumes NC: ¬ cond s
  shows case lasso s of
    None ⇒ ¬(∃ v0∈V0. ∃ v. (v0,v) ∈ E* ∧ accpt v ∧ (v,v) ∈ E⁺)
  | Some (pr,pl) ⇒ (∃ v0∈V0. ∃ v.
      path E v0 pr v ∧ accpt v ∧ pl≠[] ∧ path E v pl v)
proof (cases lasso s)
  case None
  moreover
  {
    fix v v0
    assume v0 ∈ V0 (v0,v) ∈ E* accpt v (v,v) ∈ E⁺
    moreover
    hence v ∈ reachable by (auto)
    with nc-finished-eq-reachable NC None have v ∈ dom (finished s)
      by simp
    moreover note no-accpt-cycle-in-finish None
    ultimately have False by blast
  }
  ultimately show ?thesis by auto
next
  case (Some prpl) with s-lasso-inv show ?thesis
    by (cases prpl)
       (auto intro: path-is-rtrancl path-is-trancl simp: lasso-inv-def)
qed

end
```

2.5.4 Interface

```
interpretation BlueDFS-defs for G accpt .

definition nested-dfs-spec G accpt ≡ λr. case r of
  None ⇒ ¬(∃ v0∈g-V0 G. ∃ v. (v0,v) ∈ (g-E G)* ∧ accpt v ∧ (v,v) ∈ (g-E G)⁺)
| Some (pr,pl) ⇒ (∃ v0∈g-V0 G. ∃ v.
    path (g-E G) v0 pr v ∧ accpt v ∧ pl≠[] ∧ path (g-E G) v pl v)

definition nested-dfs G accpt ≡ do {
  ASSERT (fb-graph G);
  s ← it-dfs TYPE('a) G accpt;
  RETURN (lasso s)
}

theorem nested-dfs-correct:
  assumes fb-graph G
  shows nested-dfs G accpt ≤ SPEC (nested-dfs-spec G accpt)
proof −
  interpret fb-graph G by fact
  interpret BlueDFS G accpt by unfold-locales
```

show *?thesis*
  unfolding *nested-dfs-def*
  apply (*refine-rcg refine-vcg*)
  apply *fact*
  apply (*rule weaken-SPEC*[*OF it-dfs-correct*])
  apply *clarsimp*
proof −
  fix *s*
  assume *BlueDFS-invar G accpt s*
  then interpret *BlueDFS-invar G accpt s* **.**

  assume ¬*cond TYPE*(′*b*) *G accpt s*
  from *blue-dfs-correct*[*OF this*] **show** *nested-dfs-spec G accpt* (*lasso s*)
    unfolding *nested-dfs-spec-def* **by** *simp*
qed
qed

2.5.5 Implementation

record ′*v bdfs-state-impl* = ′*v simple-state* +
  lasso-impl :: (′*v list* × ′*v list*) *option*
  red-impl :: ′*v set*

definition *bdfs-erel* ≡ {((|*lasso-impl*=*li*,*red-impl*=*ri*|),(|*lasso*=*l*, *red*=*r*|))
  | *li ri l r*. *li*=*l* ∧ *ri*=*r*}

abbreviation *bdfs-rel* ≡ ⟨*bdfs-erel*⟩*simple-state-rel*

definition *mk-blue-witness-impl*
  :: ′*v bdfs-state-impl* ⇒ ′*v fpr-result* ⇒ (′*v*,*unit*) *bdfs-state-impl-ext*
  where
  mk-blue-witness-impl s redS ≡
    case redS of
      Inl R′ ⇒ (| *lasso-impl* = *None*, *red-impl* = (*R*′ ~~∪ *red-impl s*~~) |)
    | *Inr* (*vs*, *v*) ⇒ *let*
        rs = *rev* (*map fst* (*CAST* (*ss-stack s*)))
      in (|
        lasso-impl = *Some* (*rs*, *vs*@*dropWhileNot v rs*),
        red-impl = *red-impl s*|)

lemma *mk-blue-witness-impl*[*refine*]:
  ⟦ (*si*,*s*)∈*bdfs-rel*; (*ri*,*r*)∈⟨*Id*, ⟨*Id*⟩*list-rel* $\times_r$ *Id*⟩*sum-rel* ⟧
  ⟹ (*mk-blue-witness-impl si ri*, *mk-blue-witness s r*)∈*bdfs-erel*
  unfolding *mk-blue-witness-impl-def mk-blue-witness-def*
  apply *parametricity*
  apply (*cases si*, *cases s*)
  apply (*auto simp*: *bdfs-erel-def simple-state-rel-def*) []
  apply (*rule introR*[**where** *R*=⟨*Id*⟩*list-rel*])

apply (*cases si*, *cases s*)
apply (*auto simp*: *bdfs-erel-def simple-state-rel-def comp-def*) []
apply (*cases si*, *cases s*)
apply (*auto simp*: *bdfs-erel-def simple-state-rel-def*) []
done

definition *cyan-impl s* ≡ *on-stack s*
lemma *cyan-impl*[*refine*]: ⟦(*si*,*s*)∈*bdfs-rel*⟧ ⟹ (*cyan-impl si*, *cyan s*)∈*Id*
unfolding *cyan-impl-def cyan-def*
by (*auto simp*: *bdfs-erel-def simple-state-rel-def*)

definition *run-red-dfs-impl*
:: (′*v*, ′*more*) *graph-rec-scheme* ⇒ ′*v* ⇒ ′*v bdfs-state-impl* ⇒ (′*v*,*unit*) *bdfs-state-impl-ext nres*
where
run-red-dfs-impl G u s ≡ *case lasso-impl s of None* ⇒ *do* {
redS ← *red-dfs TYPE*(′*more*) *G* (*red-impl s*) (λ*x*. *x* = *u* ∨ *x* ∈ *cyan-impl s*) *u*;
RETURN (*mk-blue-witness-impl s redS*)
}
| - ⇒ *RETURN* (*simple-state.more s*)

lemma *run-red-dfs-impl*[*refine*]: ⟦(*Gi*,*G*)∈*Id*; (*ui*,*u*)∈*Id*; (*si*,*s*)∈*bdfs-rel*⟧
⟹ *run-red-dfs-impl Gi ui si* ≤⇓*bdfs-erel* (*run-red-dfs TYPE*(′*a*) *G u s*)
unfolding *run-red-dfs-impl-def run-red-dfs-def*
apply *refine-rcg*
apply *refine-dref-type*
apply (*cases si*, *cases s*, *auto simp*: *bdfs-erel-def simple-state-rel-def*) []
apply (*cases si*, *cases s*,
auto simp: *bdfs-erel-def simple-state-rel-def cyan-impl-def cyan-def*) []
apply (*auto simp*: *bdfs-erel-def simple-state-rel-def*) [*2*]
done

definition *se-back-edge-impl accpt u v s* ≡ *case lasso-impl s of*
None ⇒
if accpt u then
let rs = *rev* (*map fst* (*tl* (*CAST* (*ss-stack s*))));
ur = *rs*;
ul = *u*#*dropWhileNot v rs*
in RETURN (|*lasso-impl* = *Some* (*ur*,*ul*), *red-impl* = *red-impl s*|)
else if accpt v then
let rs = *rev* (*map fst* (*CAST* (*ss-stack s*)));
vr = *takeWhileNot v rs*;
vl = *dropWhileNot v rs*
in RETURN (|*lasso-impl* = *Some* (*vr*,*vl*), *red-impl* = *red-impl s*|)
else RETURN (*simple-state.more s*)
| - ⇒ *RETURN* (*simple-state.more s*)

lemma *se-back-edge-impl*[*refine*]: ⟦ (*accpti*,*accpt*)∈*Id*; (*ui*,*u*)∈*Id*; (*vi*,*v*)∈*Id*; (*si*,*s*)∈*bdfs-rel*

⟧
  $\Longrightarrow$ *se-back-edge-impl accpt ui vi si* $\leq\Downarrow$*bdfs-erel* (*se-back-edge accpt u v s*)
  unfolding *se-back-edge-impl-def se-back-edge-def*
  apply *refine-rcg*
  apply *refine-dref-type*
  apply *simp-all*
  apply (*simp-all add*: *bdfs-erel-def simple-state-rel-def*)
  apply (*cases si*, *cases s*, (*auto*) [])
  apply (*cases si*, *cases s*, (*auto simp*: *map-tl comp-def*) [])
  apply (*cases si*, *cases s*, (*auto simp*: *comp-def*) [])
  done

lemma *NOOP-impl*: (*si*, *s*) ∈ *bdfs-rel*
  $\Longrightarrow$ *RETURN* (*simple-state.more si*) ≤ ⇓ *bdfs-erel* (*NOOP s*)
  apply (*simp add*: *pw-le-iff refine-pw-simps*)
  apply (*auto simp*: *simple-state-rel-def*)
  done

definition *bdfs-params-impl*
  :: ('*v*, '*more*) *graph-rec-scheme* ⇒ ('*v* ⇒ *bool*) ⇒ ('*v*,'*v bdfs-state-impl*,('*v*,*unit*)*bdfs-state-impl-ext*) *gen-parameterization*
  where *bdfs-params-impl G accpt* ≡ ⦇
  on-init = *RETURN* ⦇*lasso-impl* = *None*, *red-impl* = {}⦈,
  on-new-root = λ*v0 s*. *RETURN* (*simple-state.more s*),
  on-discover = λ*u v s*. *RETURN* (*simple-state.more s*),
  on-finish = λ*u s*.
    if accpt u then run-red-dfs-impl G u s else RETURN (*simple-state.more s*),
  on-back-edge = *se-back-edge-impl accpt*,
  on-cross-edge = λ*u v s*. *RETURN* (*simple-state.more s*),
  is-break = λ*s*. *lasso-impl s* ≠ *None* ⦈

lemmas *bdfs-params-impl-simps*[*simp*, *DFS-code-unfold*] =
  gen-parameterization.simps[*mk-record-simp*, *OF bdfs-params-impl-def*]

interpretation *impl*: *simple-impl-defs G bdfs-params-impl G accpt blue-dfs-params* *TYPE*('*a*) *G accpt*
  for *G accpt* **.**

context *BlueDFS* **begin**

sublocale *impl*: *simple-impl G blue-dfs-params bdfs-params-impl G accpt bdfs-erel*
  apply *unfold-locales*

  apply (*simp-all*
    add: *bdfs-params-impl-def run-red-dfs-impl se-back-edge-impl NOOP-impl*)
  apply *parametricity*

apply (*clarsimp-all simp*: *pw-le-iff refine-pw-simps bdfs-erel-def simple-state-rel-def*)
apply (*rename-tac si s x y*, *case-tac si*, *case-tac s*)
apply (*auto simp add*: *bdfs-erel-def simple-state-rel-def*) []
done

lemmas *impl* = *impl.simple-tailrec-refine*
end

definition *nested-dfs-impl G accpt* ≡ *do* {
ASSERT (*fb-graph G*);
s ← *impl.tailrec-impl TYPE*(′*a*) *G accpt*;
RETURN (*lasso-impl s*)
}

lemma *nested-dfs-impl*[*refine*]:
assumes (*Gi*,*G*)∈*Id*
assumes (*accpti*,*accpt*)∈*Id*
shows *nested-dfs-impl Gi accpti* ≤⇓(⟨⟨*Id*⟩*list-rel* $\times_r$ ⟨*Id*⟩*list-rel*⟩*option-rel*)
(*nested-dfs G accpt*)
using *assms*
unfolding *nested-dfs-impl-def nested-dfs-def*
apply *refine-rcg*
apply *simp-all*
apply (*rule intro-prgR*[**where** *R*=*bdfs-rel*])
defer
apply (*rename-tac si s*)
apply (*case-tac si*, *case-tac s*)
apply (*auto simp*: *bdfs-erel-def simple-state-rel-def*) []
proof −
assume *fb-graph G*
then interpret *fb-graph G* **.**
interpret *BlueDFS G* **by** *unfold-locales*

from *impl* **show** *impl.tailrec-impl TYPE*(′*b*) *G accpt* ≤⇓*bdfs-rel* (*it-dfs TYPE*(′*b*) *G accpt*) **.**
qed

2.5.6 Synthesis of Executable Code

record (′*v*,′*si*,′*nsi*)*bdfs-state-impl*′ = (′*si*,′*nsi*)*simple-state-impl* +
lasso-impl′ :: (′*v list* × ′*v list*) *option*
red-impl′ :: ′*nsi*

definition [*to-relAPP*]: *bdfs-state-erel*′ *Vi* ≡ {
((|*lasso-impl*′ = *li*, *red-impl*′=*ri*|),(|*lasso-impl* = *l*, *red-impl* = *r*|)) | *li ri l r*.
(*li*,*l*)∈⟨⟨*Vi*⟩*list-rel* $\times_r$ ⟨*Vi*⟩*list-rel*⟩*option-rel* ∧ (*ri*,*r*)∈⟨*Vi*⟩*dflt-ahs-rel*}

consts
i-bdfs-state-ext :: *interface* ⇒ *interface*

lemmas [*autoref-rel-intf*] = *REL-INTFI*[*of bdfs-state-erel′ i-bdfs-state-ext*]

lemma [*autoref-rules*]:
fixes *ns-rel vis-rel Vi*
defines $R \equiv \langle ns\text{-}rel, vis\text{-}rel, \langle Vi\rangle bdfs\text{-}state\text{-}erel'\rangle ss\text{-}impl\text{-}rel$
shows
$(bdfs\text{-}state\text{-}impl'\text{-}ext,\ bdfs\text{-}state\text{-}impl\text{-}ext)$
$\in \langle\langle Vi\rangle list\text{-}rel \times_r \langle Vi\rangle list\text{-}rel\rangle option\text{-}rel \to \langle Vi\rangle dflt\text{-}ahs\text{-}rel \to unit\text{-}rel \to$
$\langle Vi\rangle bdfs\text{-}state\text{-}erel'$
$(lasso\text{-}impl',\ lasso\text{-}impl) \in R \to \langle\langle Vi\rangle list\text{-}rel \times_r \langle Vi\rangle list\text{-}rel\rangle option\text{-}rel$
$(red\text{-}impl',\ red\text{-}impl) \in R \to \langle Vi\rangle dflt\text{-}ahs\text{-}rel$
unfolding *bdfs-state-erel′-def ss-impl-rel-def R-def*
by *auto*

schematic-goal *nested-dfs-code*:
assumes *Vid*: $V = (Id :: ('v::hashable \times 'v)\ set)$
assumes [*unfolded Vid*, *autoref-rules*]:
$(Gi,\ G) \in \langle Rm,\ V\rangle g\text{-}impl\text{-}rel\text{-}ext$
$(accpti, accpt) \in (V \to bool\text{-}rel)$
notes [*unfolded Vid*, *autoref-tyrel*] =
TYRELI[**where** $R = \langle V\rangle dflt\text{-}ahs\text{-}rel$]
TYRELI[**where** $R = \langle V\rangle ras\text{-}rel$]
shows (*nres-of ?c*, *nested-dfs-impl G accpt*)
$\in \langle\langle\langle V\rangle list\text{-}rel \times_r \langle V\rangle list\text{-}rel\rangle option\text{-}rel\rangle nres\text{-}rel$
unfolding *nested-dfs-impl-def*[*abs-def*] *Vid*
se-back-edge-impl-def run-red-dfs-impl-def mk-blue-witness-impl-def
cyan-impl-def
DFS-code-unfold

using [[*autoref-trace-failed-id*]]
apply (*autoref-monadic* (*trace*))
done

concrete-definition *nested-dfs-code* **uses** *nested-dfs-code*

export-code *nested-dfs-code* **checking** *SML*

2.5.7 Conclusion

We have implemented an efficiently executable nested DFS algorithm. The following theorem declares this implementation to the Autoref tool, such that it uses it to synthesize efficient code for *nested-dfs*. Note that you will need the lemma *nested-dfs-correct* to link *nested-dfs* to an abstract specification, which is usually done in a previous refinement step.

theorem *nested-dfs-autoref*[*autoref-rules*]:
assumes *PREFER-id V*
shows (λ *G accpt. nres-of* (*nested-dfs-code G accpt*),*nested-dfs*) $\in$

$\langle Rm,\ V\rangle g\text{-}impl\text{-}rel\text{-}ext \rightarrow (V \rightarrow bool\text{-}rel) \rightarrow$
$\langle\langle\langle V\rangle list\text{-}rel \times_r \langle V\rangle list\text{-}rel\rangle option\text{-}rel\rangle nres\text{-}rel$
proof −
from *assms* **have** *Vid*: *V=Id* **by** *simp*
note *nested-dfs-code.refine*[*OF Vid,param-fo*, *THEN nres-relD*]
also note *nested-dfs-impl*
finally show *?thesis* **by** (*fastforce intro*: *nres-relI*)
qed

end

2.6 Invariants for Tarjan's Algorithm

theory *Tarjan-LowLink*
imports
../DFS-Framework
../Invars/DFS-Invars-SCC
begin

context *param-DFS-defs* **begin**

definition
$lowlink\text{-}path\ s\ v\ p\ w \equiv path\ E\ v\ p\ w \wedge p \neq []$
$\wedge (last\ p,\ w) \in cross\text{-}edges\ s \cup back\text{-}edges\ s$
$\wedge (length\ p > 1 \longrightarrow$
$p!1 \in dom\ (finished\ s)$
$\wedge (\forall k < length\ p - 1.\ (p!k,\ p!Suc\ k) \in tree\text{-}edges\ s))$

definition
$lowlink\text{-}set\ s\ v \equiv \{w \in dom\ (discovered\ s).$
$v = w$
$\vee (v,w) \in E^+ \wedge (w,v) \in E^+$
$\wedge (\exists p.\ lowlink\text{-}path\ s\ v\ p\ w)\}$

context begin interpretation *timing-syntax* **.**
abbreviation *LowLink* **where**
$LowLink\ s\ v \equiv Min\ (\delta\ s\ `\ lowlink\text{-}set\ s\ v)$
end

end

context *DFS-invar* **begin**

lemma *lowlink-setI*:
assumes $lowlink\text{-}path\ s\ v\ p\ w$
and $w \in dom\ (discovered\ s)$
and $(v,w) \in E^*\ (w,v) \in E^*$
shows $w \in lowlink\text{-}set\ s\ v$

proof (*cases* $v = w$)
  case *True* **thus** *?thesis* **by** (*simp add*: *lowlink-set-def assms*)
next
  case *False* **with** *assms* **have** $(v,w) \in E^+$ $(w,v) \in E^+$ **by** (*metis rtrancl-eq-or-trancl*)+
  with *assms* **show** *?thesis* **by** (*auto simp add*: *lowlink-set-def*)
qed

lemma *lowlink-set-discovered*:
  lowlink-set s v $\subseteq$ *dom* (*discovered s*)
  unfolding *lowlink-set-def*
  by *blast*

lemma *lowlink-set-finite*[*simp*, *intro*!]:
  finite (*lowlink-set s v*)
  using *lowlink-set-discovered discovered-finite*
  by (*metis finite-subset*)

lemma *lowlink-set-not-empty*:
  assumes $v \in$ *dom* (*discovered s*)
  shows *lowlink-set s v* $\neq \{\}$
  unfolding *lowlink-set-def*
  using *assms* **by** *auto*

lemma *lowlink-path-single*:
  assumes $(v,w) \in$ *cross-edges s* $\cup$ *back-edges s*
  shows *lowlink-path s v* [*v*] *w*
  unfolding *lowlink-path-def*
  using *assms cross-edges-ssE back-edges-ssE*
  by (*auto simp add*: *path-simps*)

lemma *lowlink-path-Cons*:
  assumes *lowlink-path s v* (*x*#*xs*) *w*
  and *xs* $\neq$ []
  shows $\exists$ *u*. *lowlink-path s u xs w*
proof −
  from *assms* **have** *path*: *path E v* (*x*#*xs*) *w*
    and *cb*: (*last xs*, *w*) $\in$ *cross-edges s* $\cup$ *back-edges s*
    and *f*: (*x*#*xs*)!*1* $\in$ *dom* (*finished s*)
    and *t*: ($\forall$ *k* < *length xs*. ((*x*#*xs*)!*k*, (*x*#*xs*)!*Suc k*) $\in$ *tree-edges s*)
    unfolding *lowlink-path-def*
    by *auto*

  from *path* **obtain** *u* **where** *path E u xs w* **by** (*auto simp add*: *path-simps*)
  moreover note *cb* ‹*xs* $\neq$ []›
  moreover { **fix** *k* **define** k' **where** $k' =$ *Suc k*
    assume *k* < *length xs* − *1*
    with k'*-def* **have** k' < *length xs* **by** *simp*
    with *t* **have** ((*x*#*xs*)!k', (*x*#*xs*)!*Suc* k') $\in$ *tree-edges s* **by** *simp*
    hence (*xs*!*k*,*xs*!*Suc k*) $\in$ *tree-edges s* **by** (*simp add*: k'*-def nth-Cons*$'$)

} **note** $t' = this$
moreover {
assume $*$: *length xs* $>$ *1*
from *f* **have** *xs*!*0* $\in$ *dom* (*finished s*) **by** *auto*
moreover from t'[*of 0*] $*$ **have** (*xs*!*0*,*xs*!*1*) $\in$ *tree-edges s* **by** *simp*
ultimately have *xs*!*1* $\in$ *dom* (*finished s*) **using** *tree-edge-impl-parenthesis* **by** *metis*
}

ultimately have *lowlink-path s u xs w* **by** (*auto simp add*: *lowlink-path-def*)
thus *?thesis* **..**
qed

lemma *lowlink-path-in-tree*:
assumes *p*: *lowlink-path s v p w*
and *j*: $j <$ *length p*
and *k*: $k < j$
shows (*p*!*k*, *p*!*j*) $\in$ (*tree-edges s*)$^+$
proof $-$
from *p* **have** $p \neq []$ **by** (*auto simp add*: *lowlink-path-def*)
thus *?thesis* **using** *p j k*
proof (*induction arbitrary*: *v j k rule*: *list-nonempty-induct*)
case *single* **thus** *?case* **by** *auto*
next
case (*cons x xs*)
define j' **where** $j' = j - 1$
with *cons* **have** j'*-le*: $j' <$ *length xs* **and** $k \leq j'$ **and** *j*: $j = Suc\ j'$ **by** *auto*

from *cons lowlink-path-Cons* **obtain** *u* **where** *p*: *lowlink-path s u xs w* **by** *blast*

show *?case*
proof (*cases k=0*)
case *True*
from *cons* **have** $\bigwedge k.\ k <$ *length* (*x*#*xs*) $-$ *1* $\Longrightarrow$ ((*x*#*xs*)!*k*,(*x*#*xs*)!*Suc k*) $\in$ *tree-edges s*
unfolding *lowlink-path-def*
by *auto*
moreover from *True cons* **have** $k <$ *length* (*x*#*xs*) $-$ *1* **by** *auto*
ultimately have $*$: ((*x*#*xs*)!*k*,(*x*#*xs*)!*Suc k*) $\in$ *tree-edges s* **by** *metis*

show *?thesis*
proof (*cases* $j' = 0$)
case *True* **with** $*$ *j* ‹*k=0*› **show** *?thesis* **by** *simp*
next
case *False* **with** *True* **have** $j' > k$ **by** *simp*
with *cons.IH*[*OF p* j'*-le*] **have** (*xs*!*k*, *xs*!j') $\in$ (*tree-edges s*)$^+$ **.**
with *j* **have** ((*x*#*xs*)!*Suc k*, (*x*#*xs*)!*j*) $\in$ (*tree-edges s*)$^+$ **by** *simp*
with $*$ **show** *?thesis* **by** (*metis trancl-into-trancl2*)

qed
next
case *False*
define k' **where** $k' = k - 1$
with *False* ‹$k \leq j'$› **have** $k' < j'$ **and** k: $k = Suc\ k'$ **by** *simp-all*
with *cons.IH*[*OF p j'-le*] **have** $(xs!k', xs!j') \in (tree\text{-}edges\ s)^+$ **by** *metis*
hence $((x\#xs)!Suc\ k', (x\#xs)!Suc\ j') \in (tree\text{-}edges\ s)^+$ **by** *simp*
with *k j* **show** *?thesis* **by** *simp*
qed
qed
qed

lemma *lowlink-path-finished*:
assumes *p*: *lowlink-path s v p w*
and *j*: $j < length\ p\ j > 0$
shows $p!j \in dom\ (finished\ s)$
proof −
from *j* **have** $length\ p > 1$ **by** *simp*
with *p* **have** *f*: $p!1 \in dom\ (finished\ s)$ **by** (*simp add*: *lowlink-path-def*)
thus *?thesis*
proof (*cases j=1*)
case *False* **with** *j* **have** $j > 1$ **by** *simp*
with *assms lowlink-path-in-tree*[**where** *k=1*] **have** $(p!1,p!j) \in (tree\text{-}edges\ s)^+$
by *simp*
with *f tree-path-impl-parenthesis* **show** *?thesis* **by** *simp*
qed *simp*
qed

lemma *lowlink-path-tree-prepend*:
assumes *p*: *lowlink-path s v p w*
and *tree-edges*: $(u,v) \in (tree\text{-}edges\ s)^+$
and *fin*: $u \in dom\ (finished\ s) \lor (stack\ s \neq [] \land u = hd\ (stack\ s))$
shows $\exists p.\ lowlink\text{-}path\ s\ u\ p\ w$
proof −
note *lowlink-path-def*[*simp*]

from *tree-edges trancl-is-path* **obtain** *tp* **where**
tp: $path\ (tree\text{-}edges\ s)\ u\ tp\ v\ tp \neq []$
by *metis*

from *tree-path-impl-parenthesis assms hd-stack-tree-path-finished* **have**
v-fin: $v \in dom\ (finished\ s)$ **by** *blast*

from *p* **have** $p!0 = hd\ p$ **by** (*simp add*: *hd-conv-nth*)
with *p* **have** *p-0*: $p!0 = v$ **by** (*auto simp add*: *path-hd*)

let $?p = tp\ @\ p$

{

from *tp path-mono*[*OF tree-edges-ssE*] **have** *path E u tp v* **by** *simp*
also from *p* **have** *path E v p w* **by** *simp*
finally have *path E u ?p w* **.**
}

moreover from *p* **have** $?p \neq []$ **by** *simp*

moreover
from *p* **have** $(last\ ?p,\ w) \in cross\text{-}edges\ s \cup back\text{-}edges\ s$ **by** *simp*

moreover {
assume $length\ ?p > 1$

have $?p\ !\ 1 \in dom\ (finished\ s)$
proof $(cases\ length\ tp > 1)$
case *True* **hence** $tp1$: $?p\ !\ 1 = tp\ !\ 1$ **by** (*simp add*: *nth-append*)
from *tp True* **have** $(tp\ !\ 0,\ tp\ !\ 1) \in (tree\text{-}edges\ s)^+$
by (*auto simp add*: *path-nth-conv nth-append elim*: *allE*[**where** $x=0$])
also from *True* **have** $tp\ !\ 0 = hd\ tp$ **by** (*simp add*: *hd-conv-nth*)
also from *tp* **have** $hd\ tp = u$ **by** (*simp add*: *path-hd*)
finally have $tp\ !\ 1 \in dom\ (finished\ s)$
using *tree-path-impl-parenthesis fin hd-stack-tree-path-finished* **by** *blast*
thus *?thesis* **by** (*subst tp1*)
next
case *False* **with** *tp* **have** $length\ tp = 1$ **by** (*cases tp*) *auto*
with *p-0* **have** $?p\ !\ 1 = v$ **by** (*simp add*: *nth-append*)
thus *?thesis* **by** (*simp add*: *v-fin*)
qed

also have $\forall k < length\ ?p - 1.\ (?p!k,\ ?p!Suc\ k) \in tree\text{-}edges\ s$
proof (*safe*)
fix *k*
assume *A*: $k < length\ ?p - 1$
show $(?p!k,\ ?p!Suc\ k) \in tree\text{-}edges\ s$
proof $(cases\ k < length\ tp)$
case *True* **hence** *k*: $?p\ !\ k = tp\ !\ k$ **by** (*simp add*: *nth-append*)
show *?thesis*
proof $(cases\ Suc\ k < length\ tp)$
case *True* **hence** $?p\ !\ Suc\ k = tp\ !\ Suc\ k$ **by** (*simp add*: *nth-append*)
moreover from *True tp* **have** $(tp\ !\ k,\ tp\ !\ Suc\ k) \in tree\text{-}edges\ s$
by (*auto simp add*: *path-nth-conv nth-append*
elim: *allE*[**where** $x=k$])
ultimately show *?thesis* **using** *k* **by** *simp*
next
case *False* **with** *True* **have** $*$: $Suc\ k = length\ tp$ **by** *simp*
with *tp True* **have** $(tp\ !\ k,\ v) \in tree\text{-}edges\ s$
by (*auto simp add*: *path-nth-conv nth-append*
elim: *allE*[**where** $x=k$])
also from $*$ *p-0* **have** $v = ?p\ !\ Suc\ k$ **by** (*simp add*: *nth-append*)

finally show *?thesis* **by** (*simp add*: *k*)
qed
next
case *False* **hence** $*$: *Suc k* $-$ *length tp* $=$ *Suc* (*k* $-$ *length tp*) **by** *simp*
define k' **where** $k' = k -$ *length tp*
with *False* $*$ **have** k': *?p* ! *k* $=$ *p* ! k' *?p* ! *Suc k* $=$ *p* ! *Suc* k'
by (*simp-all add*: *nth-append*)
from k'*-def False A* **have** $k' <$ *length p* $-$ *1* **by** *simp*
with *p* **have** (*p*!k', *p*!*Suc* k') $\in$ *tree-edges s* **by** *simp*
with k' **show** *?thesis* **by** *simp*
qed
qed

also (*conjI*) **note** *calculation*
}

ultimately have *lowlink-path s u ?p w* **by** *simp*
thus *?thesis* **..**
qed

lemma *lowlink-path-complex*:
assumes (*u*,*v*) $\in$ (*tree-edges s*)$^+$
and *u* $\in$ *dom* (*finished s*) $\vee$ (*stack s* $\neq$ [] $\wedge$ *u* $=$ *hd* (*stack s*))
and (*v*,*w*) $\in$ *cross-edges s* $\cup$ *back-edges s*
shows $\exists$ *p*. *lowlink-path s u p w*
proof $-$
from *assms lowlink-path-single* **have** *lowlink-path s v* [*v*] *w* **by** *simp*
with *assms lowlink-path-tree-prepend* **show** *?thesis* **by** *simp*
qed

lemma *no-path-imp-no-lowlink-path*:
assumes *edges s* `` {*v*} $=$ {}
shows $\neg$*lowlink-path s v p w*
proof
assume *p*: *lowlink-path s v p w*
hence [*simp*]: *p*$\neq$[] **by** (*simp add*: *lowlink-path-def*)

from *p* **have** *hd p* $=$ *v* **by** (*auto simp add*: *lowlink-path-def path-hd*)
with *hd-conv-nth*[*OF* ‹*p*$\neq$[]›] **have** *v*: *p*!*0* $=$ *v* **by** *simp*

show *False*
proof (*cases length p* $>$ *1*)
case *True* **with** *p* **have** (*p*!*0*,*p*!*1*) $\in$ *tree-edges s* **by** (*simp add*: *lowlink-path-def*)
with *v assms* **show** *False* **by** *auto*
next
case *False* **with** ‹*p*$\neq$[]› **have** *length p* $=$ *1* **by** (*cases p*) *auto*
hence *last p* $=$ *v* **by** (*simp add*: *last-conv-nth v*)
with *p* **have** (*v*,*w*) $\in$ *edges s* **by** (*simp add*: *lowlink-path-def*)

with *assms* **show** *False* **by** *auto*
qed
qed

context begin interpretation *timing-syntax* **.**

lemma *LowLink-le-disc*:
assumes $v \in dom\ (discovered\ s)$
shows $LowLink\ s\ v \leq \delta\ s\ v$
using *assms*
unfolding *lowlink-set-def*
by *clarsimp*

lemma *LowLink-lessE*:
assumes $LowLink\ s\ v < x$
and $v \in dom\ (discovered\ s)$
obtains w **where** $\delta\ s\ w < x\ w \in lowlink\text{-}set\ s\ v$
proof $-$
let $?L = \delta\ s$ ' $lowlink\text{-}set\ s\ v$

note *assms*
moreover from *lowlink-set-finite* **have** *finite ?L* **by** *auto*
moreover from *lowlink-set-not-empty assms* **have** $?L \neq \{\}$ **by** *auto*
ultimately show *?thesis* **using** *that* **by** (*auto simp*: *Min-less-iff*)
qed

lemma *LowLink-lessI*:
assumes $y \in lowlink\text{-}set\ s\ v$
and $\delta\ s\ y < \delta\ s\ v$
shows $LowLink\ s\ v < \delta\ s\ v$
proof $-$
let $?L = \delta\ s$ ' $lowlink\text{-}set\ s\ v$

from *assms* **have** $\delta\ s\ y \in\ ?L$ **by** *simp*
moreover hence $?L \neq \{\}$ **by** *auto*
moreover from *lowlink-set-finite* **have** *finite ?L* **by** *auto*
ultimately show *?thesis*
by (*metis Min-less-iff assms*(*2*))
qed

lemma *LowLink-eqI*:
assumes $DFS\text{-}invar\ G\ param\ s'$
assumes *sub-m*: $discovered\ s \subseteq_m discovered\ s'$
assumes *sub*: $lowlink\text{-}set\ s\ w \subseteq lowlink\text{-}set\ s'\ w$
and *rev-sub*: $lowlink\text{-}set\ s'\ w \subseteq lowlink\text{-}set\ s\ w \cup X$
and *w-disc*: $w \in dom\ (discovered\ s)$
and X: $\bigwedge x.\ [\![x \in X;\ x \in lowlink\text{-}set\ s'\ w]\!] \Longrightarrow \delta\ s'\ x \geq LowLink\ s\ w$
shows $LowLink\ s\ w = LowLink\ s'\ w$
proof (*rule ccontr*)

interpret *s'*: *DFS-invar* **where** *s*=*s'* **by** *fact*
assume *A*: *LowLink s w* $\neq$ *LowLink s' w*

from *lowlink-set-discovered sub sub-m w-disc* **have**
sub': δ *s* ‘ *lowlink-set s w* $\subseteq$ δ *s'* ‘ *lowlink-set s' w*
and *w-disc'*: *w* $\in$ *dom* (*discovered s'*)
and *eq*: $\bigwedge$*ll*. *ll* $\in$ *lowlink-set s w* $\Longrightarrow$ δ *s' ll* = δ *s ll*
by (*force simp*: *map-le-def*)+

from *lowlink-set-not-empty*[*OF w-disc*] *A Min-antimono*[*OF sub'*] *s'.lowlink-set-finite*
have
LowLink s' w < *LowLink s w* **by** *fastforce*
then obtain *ll* **where** *ll*: *ll* $\in$ *lowlink-set s' w* **and** *ll-le*: δ *s' ll* < *LowLink s w*
by (*metis s'.LowLink-lessE w-disc'*)
with *rev-sub* **have** *ll* $\in$ *lowlink-set s w* $\vee$ *ll* $\in$ *X* **by** *auto*
hence *LowLink s w* $\leq$ δ *s' ll*
proof
assume *ll* $\in$ *lowlink-set s w* **with** *lowlink-set-finite eq* **show** *?thesis* **by** *force*
next
assume *ll* $\in$ *X* **with** *ll* **show** *?thesis* **by** (*metis X*)
qed
with *ll-le* **show** *False* **by** *simp*
qed

lemma *LowLink-eq-disc-iff-scc-root*:
assumes *v* $\in$ *dom* (*finished s*) $\vee$ (*stack s* $\neq$ [] $\wedge$ *v* = *hd* (*stack s*) $\wedge$ *pending s*
‘‘ {*v*} = {})
shows *LowLink s v* = δ *s v* $\longleftrightarrow$ *scc-root s v* (*scc-of E v*)
proof
let *?scc* = *scc-of E v*
assume *scc*: *scc-root s v ?scc*
show *LowLink s v* = δ *s v*
proof (*rule ccontr*)
assume *A*: *LowLink s v* $\neq$ δ *s v*

from *assms finished-discovered stack-discovered hd-in-set* **have** *disc*: *v* $\in$ *dom*
(*discovered s*) **by** *blast*
with *assms LowLink-le-disc A* **have** *LowLink s v* < δ *s v* **by** *force*
with *disc* **obtain** *w* **where**
w: δ *s w* < δ *s v w* $\in$ *lowlink-set s v*
by (*metis LowLink-lessE*)
with *lowlink-set-discovered* **have** *wdisc*: *w* $\in$ *dom* (*discovered s*) **by** *auto*

from *w* **have** (*v*,*w*) $\in$ E^* (*w*,*v*) $\in$ E^* **by** (*auto simp add*: *lowlink-set-def*)
moreover have *is-scc E ?scc v* $\in$ *?scc* **by** *simp-all*
ultimately have *w* $\in$ *?scc* **by** (*metis is-scc-closed*)
with *wdisc scc-root-disc-le*[*OF scc*] **have** δ *s v* $\leq$ δ *s w* **by** *simp*
with *w* **show** *False* **by** *auto*
qed

next
  assume *LL*: $LowLink\ s\ v = \delta\ s\ v$

  from *assms finished-discovered stack-discovered hd-in-set* **have**
    v-disc: $v \in dom\ (discovered\ s)$ **by** *blast*

  from *assms finished-no-pending* **have**
    v-no-p: $pending\ s\ `` \{v\} = \{\}$ **by** *blast*

  let *?scc* = *scc-of E v*
  have *is-scc*: *is-scc E ?scc* **by** *simp*

  {
    fix *r*
    assume $r \neq v$
    and $r \in ?scc\ r \in dom\ (discovered\ s)$

    have $v \in ?scc$ **by** *simp*
    with ‹$r \in ?scc$› *is-scc* **have** $(v,r) \in (Restr\ E\ ?scc)^*$
      by (*simp add*: *is-scc-connected′*)

    hence $(v,r) \in (tree\text{-}edges\ s)^+$ **using** ‹$r\neq v$›
    proof (*induction rule*: *rtrancl-induct*)
      case (*step y z*) **hence** $(v,z) \in (Restr\ E\ ?scc)^*$
        by (*metis rtrancl-into-rtrancl*)
      hence $(v,z) \in E^*$ **by** (*metis Restr-rtrancl-mono*)

      from *step* **have** $(z,v) \in E^*$ **by** (*simp add*: *is-scc-connected*[*OF is-scc*])

      {
        assume *z-disc*: $z \in dom\ (discovered\ s)$
        and $\exists\, p.\ lowlink\text{-}path\ s\ v\ p\ z$
        with ‹$(z,v) \in E^*$› ‹$(v,z) \in E^*$› **have** *ll*: $z \in lowlink\text{-}set\ s\ v$
          by (*metis lowlink-setI*)
        have $\delta\ s\ v < \delta\ s\ z$
        proof (*rule ccontr*)
          presume $\delta\ s\ v \geq \delta\ s\ z$ **with** ‹$z\neq v$› *v-disc z-disc disc-unequal* **have** $\delta\ s$
$z < \delta\ s\ v$ **by** *fastforce*
          with *ll* **have** $LowLink\ s\ v < \delta\ s\ v$ **by** (*metis LowLink-lessI*)
          with *LL* **show** *False* **by** *simp*
        qed *simp*
      } **note** $\delta z = this$

      show *?case*
      proof (*cases y=v*)
        case *True* **note** [*simp*] = *this*
        with *step v-no-p v-disc no-pending-imp-succ-discovered* **have**
          z-disc: $z \in dom\ (discovered\ s)$ **by** *blast*

from *step edges-covered v-no-p v-disc* **have** $(v,z) \in$ *edges s* **by** *auto*
thus *?thesis*
proof (*rule edgesE-CB*)
  assume $(v,z) \in$ *tree-edges s* **thus** *?thesis* **..**
next
  assume *CB*: $(v,z) \in$ *cross-edges s* $\cup$ *back-edges s*
  hence *lowlink-path s v* [*v*] *z*
    by (*simp add*: *lowlink-path-single*)
    with δz[*OF z-disc*] *no-pending-succ-impl-path-in-tree v-disc v-no-p step*
show *?thesis*
    by *auto*
qed
next
case *False* **with** *step.IH* **have** *T*: $(v,y) \in (tree\text{-}edges\ s)^+$ **.**
with *tree-path-impl-parenthesis assms hd-stack-tree-path-finished tree-path-disc*
have
  y-fin: $y \in$ *dom* (*finished s*)
  and *y-δ* : $\delta\ s\ v < \delta\ s\ y$ **by** *blast+*
with *step* **have** *z-disc*: $z \in$ *dom* (*discovered s*)
  using *finished-imp-succ-discovered*
  by *auto*

from *step edges-covered finished-no-pending*[*of y*] *y-fin finished-discovered*
have
  $(y,z) \in$ *edges s*
  by *fast*
thus *?thesis*
proof (*rule edgesE-CB*)
  assume $(y,z) \in$ *tree-edges s* **with** *T* **show** *?thesis* **..**
next
  assume *CB*: $(y,z) \in$ *cross-edges s* $\cup$ *back-edges s*
  with *lowlink-path-complex*[*OF T*] *assms* **have**
    $\exists\, p.$ *lowlink-path s v p z* **by** *blast*
  with δz *z-disc* **have** δz: $\delta\ s\ v < \delta\ s\ z$ **by** *simp*

  show *?thesis*
  proof (*cases* $v \in$ *dom* (*finished s*))
    case *True* **with** *tree-path-impl-parenthesis T* **have** *y-f*: $\varphi\ s\ y < \varphi\ s\ v$
by *blast*

    from *CB* **show** *?thesis*
    proof
      assume *C*: $(y,z) \in$ *cross-edges s*
      with *cross-edges-finished-decr y-fin y-f* **have** $\varphi\ s\ z < \varphi\ s\ v$
        by *force*
      moreover note δz
      moreover from *C cross-edges-target-finished* **have**
        $z \in$ *dom* (*finished s*) **by** *simp*
      ultimately show *?thesis*

```
        using parenthesis-impl-tree-path[OF True] by metis
      next
        assume B: (y,z) ∈ back-edges s
        with back-edge-disc-lt-fin y-fin y-f have δ s z < φ s v
          by force
        moreover note δz z-disc
        ultimately have z ∈ dom (finished s) φ s z < φ s v
          using parenthesis-contained[OF True] by simp-all
        with δz show ?thesis
          using parenthesis-impl-tree-path[OF True] by metis
      qed
    next
      case False — v ∉ dom (finished s)
       with assms have st: stack s ≠ [] v = hd (stack s) pending s `` {v} =
{} by blast+

      have z ∈ dom (finished s)
      proof (rule ccontr)
        assume z ∉ dom (finished s)
        with z-disc have z ∈ set (stack s) by (simp add: stack-set-def)
        with ‹z≠v› st have z ∈ set (tl (stack s)) by (cases stack s) auto
        with st tl-lt-stack-hd-discover δz show False by force
      qed
    with δz parenthesis-impl-tree-path-not-finished v-disc False show ?thesis
by simp
    qed
   qed
  qed
 qed simp
 hence r ∈ (tree-edges s)* `` {v} by auto
}
hence ?scc ∩ dom (discovered s) ⊆ (tree-edges s)* `` {v}
  by fastforce
thus scc-root s v ?scc by (auto intro!: scc-rootI v-disc)
qed
end end
end
```

2.7 Tarjan's Algorithm

```
theory Tarjan
imports
  Tarjan-LowLink
begin
```

We use the DFS Framework to implement Tarjan's algorithm. Note that, currently, we only provide an abstract version, and no refinement to efficient code.

2.7.1 Preliminaries

lemma *tjs-union*:
  fixes *tjs u*
  defines $dw \equiv dropWhile\ ((\neq)\ u)\ tjs$
  defines $tw \equiv takeWhile\ ((\neq)\ u)\ tjs$
  assumes $u \in set\ tjs$
  shows $set\ tjs = set\ (tl\ dw) \cup insert\ u\ (set\ tw)$
proof −
  from *takeWhile-dropWhile-id* **have** $set\ tjs = set\ (tw@dw)$ **by** (*auto simp*: *dw-def tw-def*)
  hence $set\ tjs = set\ tw \cup set\ dw$ **by** (*metis set-append*)
  moreover from ‹$u \in set\ tjs$› *dropWhile-eq-Nil-conv* **have** $dw \neq []$ **by** (*auto simp*: *dw-def*)
  from *hd-dropWhile*[*OF this*[*unfolded dw-def*]] **have** $hd\ dw = u$ **by** (*simp add*: *dw-def*)
  with ‹$dw \neq []$› **have** $set\ dw = insert\ u\ (set\ (tl\ dw))$ **by** (*cases dw*) *auto*
  ultimately show *?thesis* **by** *blast*
qed

2.7.2 Instantiation of the DFS-Framework

record *'v tarjan-state* = *'v state* +
  sccs :: *'v set set*
  lowlink :: $'v \rightharpoonup nat$
  tj-stack :: *'v list*

type-synonym *'v tarjan-param* = (*'v*, (*'v*,*unit*) *tarjan-state-ext*) *parameterization*

abbreviation *the-lowlink s v* ≡ *the* (*lowlink s v*)

context *timing-syntax*
begin
  notation *the-lowlink* (ζ)
end

locale *Tarjan-def* = *graph-defs G*
  for *G* :: (*'v*, *'more*) *graph-rec-scheme*
begin
  context begin interpretation *timing-syntax* **.**

  definition *tarjan-disc* :: $'v \Rightarrow 'v$ *tarjan-state* $\Rightarrow$ (*'v*,*unit*) *tarjan-state-ext nres* **where**
    tarjan-disc v s = *RETURN* ⦇ *sccs* = *sccs s*,
        lowlink = (*lowlink s*)($v \mapsto \delta\ s\ v$),
        tj-stack = *v*#*tj-stack s*⦈

  definition *tj-stack-pop* :: $'v\ list \Rightarrow 'v \Rightarrow ('v\ list \times 'v\ set)$ *nres* **where**
    tj-stack-pop tjs u = *RETURN* (*tl* (*dropWhile* (($\neq$) *u*) *tjs*), *insert u* (*set* (*takeWhile* (($\neq$) *u*) *tjs*)))

lemma *tj-stack-pop-set*:
tj-stack-pop tjs u ≤ *SPEC* (λ(*tjs′*,*scc*). *u* ∈ *set tjs* ⟶ *set tjs* = *set tjs′* ∪ *scc* ∧ *u* ∈ *scc*)
proof −
  from *tjs-union*[*of u tjs*] **show** *?thesis*
    unfolding *tj-stack-pop-def*
    by (*refine-vcg*) *auto*
qed

lemmas *tj-stack-pop-set-leof-rule* = *weaken-SPEC*[*OF tj-stack-pop-set*, *THEN leof-lift*]

definition *tarjan-fin* :: ′*v* ⇒ ′*v tarjan-state* ⇒ (′*v*,*unit*) *tarjan-state-ext nres* **where**
tarjan-fin v s = *do* {
    let ll = (*if stack s* = [] *then lowlink s*
        else let u = *hd* (*stack s*) *in*
            (*lowlink s*)(*u* ↦ *min* (ζ *s u*) (ζ *s v*)));
    let s′ = *s*⦇ *lowlink* := *ll* ⦈;

    ASSERT (*v* ∈ *set* (*tj-stack s*));
    ASSERT (*distinct* (*tj-stack s*));
    if ζ *s v* = δ *s v then do* {
        ASSERT (*scc-root′ E s v* (*scc-of E v*));
        (*tjs*,*scc*) ← *tj-stack-pop* (*tj-stack s*) *v*;
        RETURN (*state.more* (*s′*⦇ *tj-stack* := *tjs*, *sccs* := *insert scc* (*sccs s*)⦈))
    } *else do* {
        ASSERT (¬ *scc-root′ E s v* (*scc-of E v*));
        RETURN (*state.more s′*)
    }}

definition *tarjan-back* :: ′*v* ⇒ ′*v* ⇒ ′*v tarjan-state* ⇒ (′*v*,*unit*) *tarjan-state-ext nres* **where**
tarjan-back u v s = (
    if δ *s v* < δ *s u* ∧ *v* ∈ *set* (*tj-stack s*) *then*
        let ul′ = *min* (ζ *s u*) (δ *s v*)
        in RETURN (*state.more* (*s*⦇ *lowlink* := (*lowlink s*)(*u*↦*ul′*) ⦈))
    else NOOP s)
end

definition *tarjan-params* :: ′*v tarjan-param* **where**
tarjan-params = ⦇
    on-init = *RETURN* ⦇ *sccs* = {}, *lowlink* = *Map.empty*, *tj-stack* = [] ⦈,
    on-new-root = *tarjan-disc*,
    on-discover = λ*u*. *tarjan-disc*,
    on-finish = *tarjan-fin*,
    on-back-edge = *tarjan-back*,
    on-cross-edge = *tarjan-back*,

is-break = *λs. False* ⦈

schematic-goal *tarjan-params-simps*[*simp*]:
on-init tarjan-params = *?OI*
on-new-root tarjan-params = *?ONR*
on-discover tarjan-params = *?OD*
on-finish tarjan-params = *?OF*
on-back-edge tarjan-params = *?OBE*
on-cross-edge tarjan-params = *?OCE*
is-break tarjan-params = *?IB*
unfolding *tarjan-params-def gen-parameterization.simps*
by (*rule refl*)+

sublocale *param-DFS-defs G tarjan-params* **.**
end

locale *Tarjan* = *Tarjan-def G* +
param-DFS G tarjan-params
for *G* :: (*'v*, *'more*) *graph-rec-scheme*
begin

lemma [*simp*]:
sccs (*empty-state* ⦇*sccs* = *s*, *lowlink* = *l*, *tj-stack* = *t*⦈) = *s*
lowlink (*empty-state* ⦇*sccs* = *s*, *lowlink* = *l*, *tj-stack* = *t*⦈) = *l*
tj-stack (*empty-state* ⦇*sccs* = *s*, *lowlink* = *l*, *tj-stack* = *t*⦈) = *t*
by (*simp-all add*: *empty-state-def*)

lemma *sccs-more-cong*[*cong*]:*state.more s* = *state.more s'* ⟹ *sccs s* = *sccs s'*
by (*cases s*, *cases s'*) *simp*
lemma *lowlink-more-cong*[*cong*]:*state.more s* = *state.more s'* ⟹ *lowlink s* = *lowlink s'*
by (*cases s*, *cases s'*) *simp*
lemma *tj-stack-more-cong*[*cong*]:*state.more s* = *state.more s'* ⟹ *tj-stack s* = *tj-stack s'*
by (*cases s*, *cases s'*) *simp*

lemma [*simp*]:
s⦇ *state.more* := ⦇*sccs* = *sc*, *lowlink* = *l*, *tj-stack* = *t*⦈⦈
= *s*⦇ *sccs* := *sc*, *lowlink* := *l*, *tj-stack* := *t*⦈
by (*cases s*) *simp*
end

locale *Tarjan-invar* = *Tarjan* +
DFS-invar **where** *param* = *tarjan-params*

context *Tarjan-def* **begin**
lemma *Tarjan-invar-eq*[*simp*]:
DFS-invar G tarjan-params s ⟷ *Tarjan-invar G s* (**is** *?D* ⟷ *?T*)
proof

```
    assume ?D then interpret DFS-invar where param=tarjan-params .
    show ?T ..
  next
    assume ?T then interpret Tarjan-invar .
    show ?D ..
  qed
end
```

2.7.3 Correctness Proof

```
context Tarjan begin
  lemma i-tj-stack-discovered:
    is-invar (λs. set (tj-stack s) ⊆ dom (discovered s))
  proof (induct rule: establish-invarI)
    case (finish s)
    from finish show ?case
      apply simp
      unfolding tarjan-fin-def
      apply (refine-vcg tj-stack-pop-set-leof-rule)
      apply auto
      done
  qed (auto simp add: tarjan-disc-def tarjan-back-def)

  lemmas (in Tarjan-invar) tj-stack-discovered =
    i-tj-stack-discovered[THEN make-invar-thm]

  lemma i-tj-stack-distinct:
    is-invar (λs. distinct (tj-stack s))
  proof (induct rule: establish-invarI-ND)
    case (new-discover s s' v) then interpret Tarjan-invar where s=s by simp
    from new-discover tj-stack-discovered have v ∉ set (tj-stack s) by auto
    with new-discover show ?case by (simp add: tarjan-disc-def)
  next
    case (finish s) thus ?case
      apply simp
      unfolding tarjan-fin-def tj-stack-pop-def
      apply (refine-vcg)
      apply (auto intro: distinct-tl)
      done
  qed (simp-all add: tarjan-back-def)

  lemmas (in Tarjan-invar) tj-stack-distinct =
    i-tj-stack-distinct[THEN make-invar-thm]

  context begin interpretation timing-syntax .
  lemma i-tj-stack-incr-disc:
    is-invar (λs. ∀k<length (tj-stack s). ∀j<k. δ s (tj-stack s ! j) > δ s (tj-stack s
! k))
  proof (induct rule: establish-invarI-ND)
```

case (*new-discover s s′ v*) **then interpret** *Tarjan-invar* **where** *s*=*s* **by** *simp*

from *new-discover tj-stack-discovered* **have** $v \notin$ *set* (*tj-stack s*) **by** *auto*
moreover {
  fix *k j*
  assume $k <$ *Suc* (*length* (*tj-stack s*)) $j < k$
  hence $k -$ *Suc 0* $<$ *length* (*tj-stack s*) **by** *simp*
  hence *tj-stack s* ! ($k -$ *Suc 0*) $\in$ *set* (*tj-stack s*) **using** *nth-mem* **by** *metis*
  with *tj-stack-discovered timing-less-counter* **have** δ *s* (*tj-stack s* ! ($k -$ *Suc 0*)) $<$ *counter s* **by** *blast*
}
moreover {
  fix *k j*
  define k' **where** $k' = k -$ *Suc 0*
  define j' **where** $j' = j -$ *Suc 0*

  assume *A*: $k <$ *Suc* (*length* (*tj-stack s*)) $j < k$ (*v#tj-stack s*) ! $j \neq v$
  hence *gt-0*: $j > 0 \land k>0$ **by** (*cases j=0*) *simp-all*
  moreover with ‹$j < k$› **have** $j' < k'$ **by** (*simp add*: *j′-def k′-def*)
  moreover from *A* **have** $k' <$ *length* (*tj-stack s*) **by** (*simp add*: *k′-def*)
  ultimately have δ *s* (*tj-stack s* ! j') $> \delta$ *s* (*tj-stack s* ! k')
    using *new-discover* **by** *blast*
  with *gt-0* **have** δ *s* ((*v#tj-stack s*) ! *j*) $> \delta$ *s* (*tj-stack s* ! k')
    unfolding *j′-def*
    by (*simp add*: *nth-Cons′*)
}

ultimately show *?case*
  using *new-discover*
  by (*auto simp add*: *tarjan-disc-def*)
next
  case (*finish s s′ u*)

  {
    let *?dw* = *dropWhile* (($\neq$) *u*) (*tj-stack s*)
    let *?tw* = *takeWhile* (($\neq$) *u*) (*tj-stack s*)

    fix *a k j*
    assume *A*: *a* = *tl ?dw* $k <$ *length a* $j < k$
    and $u \in$ *set* (*tj-stack s*)
    hence *?dw* $\neq$ [] **by** *auto*

    define j' k' **where** $j' =$ *Suc j* + *length ?tw* **and** $k' =$ *Suc k* + *length ?tw*
    with ‹$j < k$› **have** $j' < k'$ **by** *simp*

    have *length* (*tj-stack s*) = *length ?tw* + *length ?dw*
      by (*simp add*: *length-append*[*symmetric*])
    moreover from *A* **have** ∗: *Suc k* $<$ *length ?dw* **and** ∗∗: *Suc j* $<$ *length ?dw* **by** *auto*

ultimately have $k' <$ *length* (*tj-stack* s) **by** (*simp add*: k'-*def*)

with *finish* ‹$j'<k'$› **have** $\delta\ s$ (*tj-stack* s ! k') $< \delta\ s$ (*tj-stack* s ! j') **by** *simp*
also from *dropWhile-nth*[*OF* ∗] **have** *tj-stack* s ! k' = *?dw* ! *Suc* k
by (*simp add*: k'-*def*)
also from *dropWhile-nth*[*OF* ∗∗] **have** *tj-stack* s ! j' = *?dw* ! *Suc* j
by (*simp add*: j'-*def*)
also from *nth-tl*[*OF* ‹*?dw* ≠ []›] **have** *?dw* ! *Suc* $k = a$! k **by** (*simp add*: *A*)
also from *nth-tl*[*OF* ‹*?dw* ≠ []›] **have** *?dw* ! *Suc* $j = a$! j **by** (*simp add*: *A*)
finally have $\delta\ s\ (a\ !\ k) < \delta\ s\ (a\ !\ j)$ **.**
} **note** *aux* = *this*

from *finish* **show** *?case*
apply *simp*
unfolding *tarjan-fin-def tj-stack-pop-def*
apply *refine-vcg*
apply (*auto intro*!: *aux*)
done
qed (*simp-all add*: *tarjan-back-def*)
end end

context *Tarjan-invar* **begin context begin interpretation** *timing-syntax* **.**
lemma *tj-stack-incr-disc*:
assumes $k <$ *length* (*tj-stack* s)
and $j < k$
shows $\delta\ s$ (*tj-stack* s ! j) $> \delta\ s$ (*tj-stack* s ! k)
using *assms i-tj-stack-incr-disc*[*THEN make-invar-thm*]
by *blast*

lemma *tjs-disc-dw-tw*:
fixes u
defines *dw* ≡ *dropWhile* ((≠) u) (*tj-stack* s)
defines *tw* ≡ *takeWhile* ((≠) u) (*tj-stack* s)
assumes $x \in$ *set dw* $y \in$ *set tw*
shows $\delta\ s\ x < \delta\ s\ y$
proof −
from *assms* **obtain** k **where** k: *dw* ! $k = x$ $k <$ *length dw* **by** (*metis in-set-conv-nth*)
from *assms* **obtain** j **where** j: *tw* ! $j = y$ $j <$ *length tw* **by** (*metis in-set-conv-nth*)

have *length* (*tj-stack* s) = *length tw* + *length dw*
by (*simp add*: *length-append*[*symmetric*] *tw-def dw-def*)
with k j **have** $\delta\ s$ (*tj-stack* s ! (k + *length tw*)) $< \delta\ s$ (*tj-stack* s ! j)
by (*simp add*: *tj-stack-incr-disc*)
also from j *takeWhile-nth* **have** *tj-stack* s ! $j = y$ **by** (*metis tw-def*)
also from *dropWhile-nth* k **have** *tj-stack* s ! (k + *length tw*) $= x$ **by** (*metis tw-def dw-def*)
finally show *?thesis* **.**
qed
end end

context *Tarjan* **begin context begin interpretation** *timing-syntax* **.**
lemma *i-sccs-finished-stack-ss-tj-stack*:
is-invar $(\lambda s. \bigcup(sccs\ s) \subseteq dom\ (finished\ s) \wedge set\ (stack\ s) \subseteq set\ (tj\text{-}stack\ s))$
proof (*induct rule*: *establish-invarI*)
case (*finish s s′ u*) **then interpret** *Tarjan-invar* **where** *s=s* **by** *simp*

let $?tw = takeWhile\ ((\neq)\ u)\ (tj\text{-}stack\ s)$
let $?dw = dropWhile\ ((\neq)\ u)\ (tj\text{-}stack\ s)$

{
fix *x*
assume *A*: $x \neq u$ $x \in set\ ?tw$ $u \in set\ (tj\text{-}stack\ s)$
hence *x-tj*: $x \in set\ (tj\text{-}stack\ s)$ **by** (*auto dest*: *set-takeWhileD*)

have $x \in dom\ (finished\ s)$
proof (*rule ccontr*)
assume $x \notin dom\ (finished\ s)$
with *x-tj tj-stack-discovered discovered-eq-finished-un-stack* **have** $x \in set\ (stack\ s)$ **by** *blast*
with ‹$x\neq u$› *finish* **have** $x \in set\ (tl\ (stack\ s))$ **by** (*cases stack s*) *auto*
with *tl-lt-stack-hd-discover finish* **have** $*$: $\delta\ s\ x < \delta\ s\ u$ **by** *simp*

from *A* **have** $?dw \neq []$ **by** *simp*
with *hd-dropWhile*[*OF this*] *hd-in-set* **have** $u \in set\ ?dw$ **by** *metis*
with *tjs-disc-dw-tw* ‹$x \in set\ ?tw$› **have** $\delta\ s\ u < \delta\ s\ x$ **by** *simp*

with $*$ **show** *False* **by** *force*
qed
hence $\exists\, y.\ finished\ s\ x = Some\ y$ **by** *blast*
} **note** *aux-scc* = *this*

{
fix *x*
assume *A*: $x \in set\ (tl\ (stack\ s))$ $u \in set\ (tj\text{-}stack\ s)$
with *finish stack-distinct* **have** $x \neq u$ **by** (*cases stack s*) *auto*

moreover
from *A* **have** $x \in set\ (stack\ s)$ **by** (*metis in-set-tlD*)
with *stack-not-finished* **have** $x \notin dom\ (finished\ s)$ **by** *simp*
with *A aux-scc*[*OF* ‹$x \neq u$›] **have** $x \notin set\ ?tw$ **by** *blast*

moreover
from *finish* ‹$x \in set\ (stack\ s)$› **have** $x \in set\ (tj\text{-}stack\ s)$ **by** *auto*

moreover note *tjs-union*[*OF* ‹$u \in set\ (tj\text{-}stack\ s)$›]

ultimately have $x \in set\ (tl\ ?dw)$ **by** *blast*
} **note** *aux-tj* = *this*

from *finish* **show** *?case*
apply *simp*
unfolding *tarjan-fin-def tj-stack-pop-def*
apply (*refine-vcg*)
using *aux-scc aux-tj* **apply** (*auto dest*: *in-set-tlD*)
done
qed (*auto simp add*: *tarjan-disc-def tarjan-back-def*)

lemma *i-tj-stack-ss-stack-finished*:
is-invar (λs. *set* (*tj-stack s*) $\subseteq$ *set* (*stack s*) $\cup$ *dom* (*finished s*))
proof (*induct rule*: *establish-invarI*)
case (*finish s*) **thus** *?case*
apply *simp*
unfolding *tarjan-fin-def*
apply (*refine-vcg tj-stack-pop-set-leof-rule*)
apply ((*simp*, *cases stack s*, *simp-all*)[])+
done
qed (*auto simp add*: *tarjan-disc-def tarjan-back-def*)

lemma *i-finished-ss-sccs-tj-stack*:
is-invar (λs. *dom* (*finished s*) $\subseteq \bigcup$(*sccs s*) $\cup$ *set* (*tj-stack s*))
proof (*induction rule*: *establish-invarI-ND*)
case (*new-discover s s′ v*) **then interpret** *Tarjan-invar* **where** *s*=*s* **by** *simp*
from *new-discover finished-discovered* **have** $v \notin$ *dom* (*finished s*) **by** *auto*
with *new-discover* **show** *?case*
by (*auto simp add*: *tarjan-disc-def*)
next
case (*finish s s′ u*) **then interpret** *Tarjan-invar* **where** *s*=*s* **by** *simp*
from *finish* **show** *?case*
apply *simp*
unfolding *tarjan-fin-def*
apply (*refine-vcg tj-stack-pop-set-leof-rule*)
apply *auto*
done
qed (*simp-all add*: *tarjan-back-def*)
end end

context *Tarjan-invar* **begin**
lemmas *finished-ss-sccs-tj-stack* =
i-finished-ss-sccs-tj-stack[*THEN make-invar-thm*]

lemmas *tj-stack-ss-stack-finished* =
i-tj-stack-ss-stack-finished[*THEN make-invar-thm*]

lemma *sccs-finished*:
$\bigcup$(*sccs s*) $\subseteq$ *dom* (*finished s*)
using *i-sccs-finished-stack-ss-tj-stack*[*THEN make-invar-thm*]
by *blast*

lemma *stack-ss-tj-stack*:
set (*stack s*) ⊆ *set* (*tj-stack s*)
using *i-sccs-finished-stack-ss-tj-stack*[*THEN make-invar-thm*]
by *blast*

lemma *hd-stack-in-tj-stack*:
stack s ≠ [] ⟹ *hd* (*stack s*) ∈ *set* (*tj-stack s*)
using *stack-ss-tj-stack hd-in-set*
by *auto*
end

context *Tarjan* **begin context begin interpretation** *timing-syntax* **.**
lemma *i-no-finished-root*:
is-invar (λ*s. scc-root s r scc* ∧ *r* ∈ *dom* (*finished s*) ⟶ (∀ *x* ∈ *scc. x* ∉ *set* (*tj-stack s*)))
proof (*induct rule*: *establish-invarI-ND-CB*)
case (*new-discover s s′ v*) **then interpret** *Tarjan-invar* **where** *s*=*s* **by** *simp*
{
fix *x*
let *?s* = *s′*⦇*state.more* := *x*⦈

assume *TRANS*: ⋀Ψ. *tarjan-disc v s′* $\leq_n$ *SPEC* Ψ ⟹ Ψ *x*
and *inv′*: *DFS-invar G tarjan-params* (*s′*⦇*state.more* := *x*⦈)
and *r*: *scc-root ?s r scc r* ∈ *dom* (*finished s′*)

from *inv′* **interpret** *s′*: *Tarjan-invar* **where** *s*=*?s* **by** *simp*

have *tj-stack ?s* = *v*#*tj-stack s*
by (*rule TRANS*) (*simp add*: *new-discover tarjan-disc-def*)

moreover
from *r s′.scc-root-finished-impl-scc-finished* **have** *scc* ⊆ *dom* (*finished ?s*) **by** *auto*
with *new-discover finished-discovered* **have** *v* ∉ *scc* **by** *force*

moreover
from *r finished-discovered new-discover* **have** *r* ∈ *dom* (*discovered s*) **by** *auto*
with *r inv′ new-discover* **have** *scc-root s r scc*
apply (*intro scc-root-transfer*[**where** *s′*=*?s*, *THEN iffD2*])
apply *clarsimp-all*
done
with *new-discover r* **have** ∀ *x* ∈ *scc. x* ∉ *set* (*tj-stack s′*) **by** *simp*

ultimately have ∀ *x*∈*scc. x* ∉ *set* (*tj-stack ?s*) **by** (*auto simp*: *new-discover*)
}
with *new-discover* **show** *?case* **by** (*simp add*: *pw-leof-iff*)
next
case (*cross-back-edge s s′ u v*) **then interpret** *Tarjan-invar* **where** *s*=*s* **by**

simp

```
  {
    fix x
    let ?s = s'(|state.more := x|)
    assume TRANS: ⋀Ψ. tarjan-back u v s' ≤n SPEC Ψ ⟹ Ψ x
    and r: scc-root ?s r scc r ∈ dom (finished s')
    with cross-back-edge have scc-root s r scc
      by (simp add: scc-root-transfer'[where s'=?s])

    moreover
     have tj-stack ?s = tj-stack s by (rule TRANS) (simp add: cross-back-edge
tarjan-back-def)

    ultimately  have ∀x∈scc. x ∉ set (tj-stack ?s)
      using cross-back-edge r by simp
  }
  with cross-back-edge show ?case by (simp add: pw-leof-iff)
 next
  case (finish s s' u) then interpret Tarjan-invar where s=s by simp

  {
    fix x
    let ?s = s'(|state.more := x|)
    assume TRANS:  ⋀Ψ. tarjan-fin u s' ≤n SPEC Ψ ⟹ Ψ x
    and inv': DFS-invar G tarjan-params (s'(|state.more := x|))
    and r: scc-root ?s r scc r ∈ dom (finished s')

    from inv' interpret s': Tarjan-invar where s=?s by simp

    have ∀x∈scc. x ∉ set (tj-stack ?s)
    proof (cases r = u)
      case False with finish r have ∀x∈scc. x ∉ set (tj-stack s)
        using scc-root-transfer'[where s'=?s]
        by simp
      moreover have set (tj-stack ?s) ⊆ set (tj-stack s)
        apply (rule TRANS)
        unfolding tarjan-fin-def
        apply (refine-vcg tj-stack-pop-set-leof-rule)
        apply (simp-all add: finish)
        done
      ultimately show ?thesis by blast
    next
       case True with r s'.scc-root-unique-is-scc have scc-root ?s u (scc-of E u)
by simp
       with s'.scc-root-transfer'[where s'=s'] finish have scc-root s' u (scc-of E
u) by simp

      moreover
      hence [simp]: tj-stack ?s = tl (dropWhile ((≠) u) (tj-stack s))
```

apply (*rule-tac TRANS*)
unfolding *tarjan-fin-def tj-stack-pop-def*
apply (*refine-vcg*)
apply (*simp-all add*: *finish*)
done

{
let *?dw* = *dropWhile* ((≠) *u*) (*tj-stack s*)
let *?tw* = *takeWhile* ((≠) *u*) (*tj-stack s*)
fix *x*
define *j*::*nat* **where** *j* = *0*

assume *x*: *x* ∈ *set* (*tj-stack ?s*)
then obtain *i* **where** *i*: *i* < *length* (*tj-stack ?s*) *tj-stack ?s* ! *i* = *x*
by (*metis in-set-conv-nth*)

have *length* (*tj-stack s*) = *length ?tw* + *length ?dw*
by (*simp add*: *length-append*[*symmetric*])
with *i* **have** δ *s* (*tj-stack s* ! (*Suc i* + *length ?tw*)) < δ *s* (*tj-stack s* ! *length ?tw*)
by (*simp add*: *tj-stack-incr-disc*)

also from *hd-stack-in-tj-stack finish* **have** *ne*: *?dw* ≠ [] **and** *length ?dw* > *0* **by** *simp-all*
from *hd-dropWhile*[*OF ne*] *hd-conv-nth*[*OF ne*] **have** *?dw* ! *0* = *u* **by** *simp*
with *dropWhile-nth*[*OF* ‹*length ?dw* > *0*›] **have** *tj-stack s* ! *length ?tw* = *u* **by** *simp*

also from *i* **have** *?dw* ! *Suc i* = *x Suc i* < *length ?dw* **by** (*simp-all add*: *nth-tl*[*OF ne*])
with *dropWhile-nth*[*OF this*(*2*)] **have** *tj-stack s* ! (*Suc i* + *length ?tw*) = *x* **by** *simp*

finally have δ *?s x* < δ *?s u* **by** (*simp add*: *finish*)

moreover from *x s′.tj-stack-discovered* **have** *x* ∈ *dom* (*discovered ?s*) **by** *auto*
ultimately have *x* ∉ *scc* **using** *s′.scc-root-disc-le r True* **by** *force*
} **thus** *?thesis* **by** *metis*
qed
}
with *finish* **show** *?case* **by** (*simp add*: *pw-leof-iff*)
qed *simp-all*
end end

context *Tarjan-invar* **begin**
lemma *no-finished-root*:
assumes *scc-root s r scc*
and *r* ∈ *dom* (*finished s*)

and $x \in scc$
shows $x \notin set$ (*tj-stack s*)
using *assms*
using *i-no-finished-root*[*THEN make-invar-thm*]
by *blast*

context begin interpretation *timing-syntax* **.**

lemma *tj-stack-reach-stack*:
assumes $u \in set$ (*tj-stack s*)
shows $\exists\, v \in set$ (*stack s*). $(u,v) \in E^* \wedge \delta\ s\ v \leq \delta\ s\ u$
proof −
have *u-scc*: $u \in$ *scc-of E u* **by** *simp*

from *assms tj-stack-discovered* **have** *u-disc*: $u \in dom$ (*discovered s*) **by** *auto*
with *scc-root-of-node-exists* **obtain** *r* **where** *r*: *scc-root s r* (*scc-of E u*) **by** *blast*
have $r \in set$ (*stack s*)
proof (*rule ccontr*)
assume $r \notin set$ (*stack s*)
with *r*[*unfolded scc-root-def*] *stack-set-def* **have** $r \in dom$ (*finished s*) **by** *simp*
with *u-scc* **have** $u \notin set$ (*tj-stack s*) **using** *no-finished-root r* **by** *blast*
with *assms* **show** *False* **by** *contradiction*
qed
moreover from *r scc-reach-scc-root u-scc u-disc* **have** $(u,r) \in E^*$ **by** *blast*
moreover from *r scc-root-disc-le u-scc u-disc* **have** $\delta\ s\ r \leq \delta\ s\ u$ **by** *blast*
ultimately show *?thesis* **by** *metis*
qed

lemma *tj-stack-reach-hd-stack*:
assumes $v \in set$ (*tj-stack s*)
shows $(v,\ hd\ (stack\ s)) \in E^*$
proof −
from *tj-stack-reach-stack assms* **obtain** *r* **where** *r*: $r \in set$ (*stack s*) $(v,r) \in E^*$ **by** *blast*
hence $r = hd$ (*stack s*) $\vee\ r \in set$ (*tl* (*stack s*)) **by** (*cases stack s*) *auto*
thus *?thesis*
proof
assume $r = hd$ (*stack s*) **with** *r* **show** *?thesis* **by** *simp*
next
from *r* **have** *ne* :*stack s* $\neq$ [] **by** *auto*

assume $r \in set$ (*tl* (*stack s*))
with *tl-stack-hd-tree-path ne* **have** $(r,hd\ (stack\ s)) \in (tree\text{-}edges\ s)^+$ **by** *simp*
with *trancl-mono-mp tree-edges-ssE* **have** $(r,hd\ (stack\ s)) \in E^*$ **by** (*metis rtrancl-eq-or-trancl*)
with ‹$(v,r) \in E^*$› **show** *?thesis* **by** (*metis rtrancl-trans*)
qed
qed

lemma *empty-stack-imp-empty-tj-stack*:
  assumes *stack s* = []
  shows *tj-stack s* = []
proof (*rule ccontr*)
  assume *ne*: *tj-stack s* ≠ []
  then obtain *x* **where** *x*: *x* ∈ *set* (*tj-stack s*) **by** *auto*
  with *tj-stack-reach-stack* **obtain** *r* **where** *r* ∈ *set* (*stack s*) **by** *auto*
  with *assms* **show** *False* **by** *simp*
qed

lemma *stacks-eq-iff*: *stack s* = [] ⟷ *tj-stack s* = []
  using *empty-stack-imp-empty-tj-stack stack-ss-tj-stack*
  by *auto*
end end

context *Tarjan* **begin context begin interpretation** *timing-syntax* **.**
lemma *i-sccs-are-sccs*:
  is-invar (λ*s*. ∀ *scc* ∈ *sccs s*. *is-scc E scc*)
proof (*induction rule*: *establish-invarI*)
  case (*finish s s′ u*) **then interpret** *Tarjan-invar* **where** *s*=*s* **by** *simp*
  from *finish* **have** *EQ*[*simp*]:
    finished s′ = (*finished s*)(*u* ↦ *counter s*)
    discovered s′ = *discovered s*
    tree-edges s′ = *tree-edges s*
    sccs s′ = *sccs s*
    tj-stack s′ = *tj-stack s*
    by *simp-all*

  {
    fix *x*

    let *?s* = *s′*⦇*state.more* := *x*⦈
    assume *TRANS*: ⋀Ψ. *tarjan-fin u s′* $\leq_n$ *SPEC* Ψ ⟹ Ψ *x*
    and *inv′*: *DFS-invar G tarjan-params* (*s′*⦇*state.more* := *x*⦈)
    then interpret *s′*: *Tarjan-invar* **where** *s*=*?s* **by** *simp*

    from *finish hd-in-set stack-set-def* **have**
      u-disc: *u* ∈ *dom* (*discovered s*)
      and *u-n-fin*: *u* ∉ *dom* (*finished s*) **by** *blast+*

    have ∀ *scc* ∈ *sccs ?s*. *is-scc E scc*
    proof (*cases scc-root s′ u* (*scc-of E u*))
      case *False*
      have *sccs ?s* = *sccs s*
        apply (*rule TRANS*)
        unfolding *tarjan-fin-def tj-stack-pop-def*
        by (*refine-vcg*) (*simp-all add*: *False*)
      thus *?thesis* **by** (*simp add*: *finish*)

next
case *True*
let *?dw* = *dropWhile* ((≠) *u*) (*tj-stack s*)
let *?tw* = *takeWhile* ((≠) *u*) (*tj-stack s*)
let *?tw′* = *insert u* (*set ?tw*)

have [*simp*]: *sccs ?s* = *insert ?tw′* (*sccs s*)
apply (*rule TRANS*)
unfolding *tarjan-fin-def tj-stack-pop-def*
by (*refine-vcg*) (*simp-all add*: *True*)

have [*simp*]: *tj-stack ?s* = *tl ?dw*
apply (*rule TRANS*)
unfolding *tarjan-fin-def tj-stack-pop-def*
by (*refine-vcg*) (*simp-all add*: *True*)

from *True scc-root-transfer′*[**where** *s′=s′*] **have** *scc-root s u* (*scc-of E u*)
by *simp*
with *inv′ scc-root-transfer*[**where** *s′=?s*] *u-disc* **have** *u-root*: *scc-root ?s u*
(*scc-of E u*) **by** *simp*

have *?tw′* ⊆ *scc-of E u*
proof
fix *v* **assume** *v*: *v* ∈ *?tw′*
show *v* ∈ *scc-of E u*
proof *cases*
assume *v* ≠ *u* **with** *v* **have** *v*: *v* ∈ *set ?tw* **by** *auto*
hence *v-tj*: *v* ∈ *set* (*tj-stack s*) **by** (*auto dest*: *set-takeWhileD*)
with *tj-stack-discovered* **have** *v-disc*: *v* ∈ *dom* (*discovered s*) **by** *auto*

from *hd-stack-in-tj-stack finish* **have** *?dw* ≠ [] **by** *simp*
with *hd-dropWhile*[*OF this*] *hd-in-set* **have** *u* ∈ *set ?dw* **by** *metis*
with *v* **have** *δ s v* > *δ s u* **using** *tjs-disc-dw-tw* **by** *blast*

moreover have *v* ∈ *dom* (*finished s*)
proof (*rule ccontr*)
assume *v* ∉ *dom* (*finished s*)
with *v-disc stack-set-def* **have** *v* ∈ *set* (*stack s*) **by** *auto*
with ‹*v*≠*u*› *finish* **have** *v* ∈ *set* (*tl* (*stack s*)) **by** (*cases stack s*) *auto*
with *tl-lt-stack-hd-discover finish* **have** *δ s v* < *δ s u* **by** *simp*
with ‹*δ s v* > *δ s u*› **show** *False* **by** *force*
qed

ultimately have (*u*,*v*) ∈ (*tree-edges s*)$^+$
using *parenthesis-impl-tree-path-not-finished*[*OF u-disc*] *u-n-fin*
by *force*
with *trancl-mono-mp tree-edges-ssE* **have** (*u*,*v*)∈E^* **by** (*metis rtrancl-eq-or-trancl*)

moreover

from *tj-stack-reach-hd-stack v-tj finish* **have** $(v,u) \in E^*$ **by** *simp*

moreover have *is-scc E* (*scc-of E u*) $u \in$ *scc-of E u* **by** *simp-all*
ultimately show *?thesis* **using** *is-scc-closed* **by** *metis*
qed *simp*
qed
moreover have *scc-of E u* $\subseteq$ *?tw'*
proof
fix *v* **assume** *v*: $v \in$ *scc-of E u*
moreover note *u-root*
moreover have $u \in$ *dom* (*finished ?s*) **by** *simp*
ultimately have $v \in$ *dom* (*finished ?s*) $v \notin$ *set* (*tj-stack ?s*)
using *s'.scc-root-finished-impl-scc-finished s'.no-finished-root*
by *auto*
with *s'.finished-ss-sccs-tj-stack* **have** $v \in \bigcup$(*sccs ?s*) **by** *blast*
hence $v \in \bigcup$(*sccs s*) $\vee$ $v \in$ *?tw'* **by** *auto*
thus $v \in$ *?tw'*
proof
assume $v \in \bigcup$(*sccs s*)
then obtain *scc* **where** *scc*: $v \in$ *scc scc* $\in$ *sccs s* **by** *auto*
moreover with *finish* **have** *is-scc E scc* **by** *simp*
moreover have *is-scc E* (*scc-of E u*) **by** *simp*
moreover note *v*
ultimately have *scc* = *scc-of E u* **using** *is-scc-unique* **by** *metis*
hence $u \in$ *scc* **by** *simp*
with *scc sccs-finished* **have** $u \in$ *dom* (*finished s*) **by** *auto*
with *u-n-fin* **show** *?thesis* **by** *contradiction*
qed *simp*
qed
ultimately have *?tw'* = *scc-of E u* **by** *auto*
hence *is-scc E ?tw'* **by** *simp*
with *finish* **show** *?thesis* **by** *auto*
qed
}
thus *?case* **by** (*auto simp*: *pw-leof-iff finish*)
qed (*simp-all add*: *tarjan-back-def tarjan-disc-def*)
end

lemmas (**in** *Tarjan-invar*) *sccs-are-sccs* =
i-sccs-are-sccs[*THEN make-invar-thm*]

context begin interpretation *timing-syntax* **.**

lemma *i-lowlink-eq-LowLink*:
is-invar ($\lambda s.\ \forall x \in$ *dom* (*discovered s*). ζ *s x* = *LowLink s x*)
proof −
{
fix *s s'* :: *'v tarjan-state*
fix *v w*

fix x

let *?s* = s'(|*state.more* := x|)

assume *pre-ll-sub-rev*: $\bigwedge w$. ⟦*Tarjan-invar G ?s*; $w \in$ *dom* (*discovered ?s*); $w \neq v$⟧ $\Longrightarrow$ *lowlink-set ?s w* $\subseteq$ *lowlink-set s w* $\cup$ $\{v\}$
assume *tree-sub* : *tree-edges* s' = *tree-edges s* $\vee$ ($\exists\, u.\ u \neq v \wedge$ *tree-edges* s' = *tree-edges s* $\cup$ $\{(u,v)\}$)

assume *Tarjan-invar G s*
assume [*simp*]: *discovered* s' = (*discovered s*)($v \mapsto$ *counter s*)
finished s' = *finished s*
lowlink s' = *lowlink s*
cross-edges s' = *cross-edges s back-edges* s' = *back-edges s*
assume *v-n-disc*: $v \notin$ *dom* (*discovered s*)
assume *IH*: $\bigwedge w$. $w \in$*dom* (*discovered s*) $\Longrightarrow \zeta$ *s w* = *LowLink s w*

assume *TRANS*: $\bigwedge \Psi$. *tarjan-disc v* $s' \leq_n$ *SPEC* $\Psi \Longrightarrow \Psi\ x$
and *INV*: *DFS-invar G tarjan-params ?s*
and *w-disc*: $w \in$ *dom* (*discovered ?s*)

interpret *Tarjan-invar* **where** *s=s* **by** *fact*
from *INV* **interpret** s':*Tarjan-invar* **where** *s=?s* **by** *simp*

have [*simp*]: *lowlink ?s* = (*lowlink s*)($v \mapsto$ *counter s*)
by (*rule TRANS*) (*auto simp*: *tarjan-disc-def*)

from *v-n-disc edge-imp-discovered* **have** *edges s* `` $\{v\}$ = {} **by** *auto*
with *tree-sub tree-edge-imp-discovered* **have** *edges ?s* `` $\{v\}$ = {} **by** *auto*
with s'.*no-path-imp-no-lowlink-path* **have** $\bigwedge w$. $\neg$($\exists\, p$. *lowlink-path ?s v p w*) **by** *metis*
hence *ll-v*: *lowlink-set ?s v* = $\{v\}$
unfolding *lowlink-set-def* **by** *auto*

have ζ *?s w* = *LowLink ?s w*
proof (*cases w=v*)
case *True* **with** *ll-v* **show** *?thesis* **by** *simp*
next
case *False* **hence** ζ *?s w* = ζ *s w* **by** *simp*
also from *IH* **have** ζ *s w* = *LowLink s w* **using** *w-disc False* **by** *simp*
also have *LowLink s w* = *LowLink ?s w*
proof (*rule LowLink-eqI*[*OF INV*])
from *v-n-disc* **show** *discovered s* $\subseteq_m$ *discovered ?s* **by** (*simp add*: *map-le-def*)

from *tree-sub* **show** *lowlink-set s w* $\subseteq$ *lowlink-set ?s w*
unfolding *lowlink-set-def lowlink-path-def*
by *auto*

show *lowlink-set ?s w* $\subseteq$ *lowlink-set s w* $\cup$ $\{v\}$

```
proof (cases w = v)
  case True with ll-v show ?thesis by auto
next
  case False thus ?thesis
    using pre-ll-sub-rev w-disc INV
    by simp
qed

show w ∈ dom (discovered s) using w-disc False by simp

  fix ll assume ll ∈ {v} with timing-less-counter lowlink-set-discovered
have
  ⋀x. x∈δ s`lowlink-set s w ⟹ x < δ ?s ll by simp force
  moreover from Min-in lowlink-set-finite lowlink-set-not-empty w-disc
False have
  LowLink s w ∈ δ s`lowlink-set s w  by auto
ultimately show LowLink s w ≤ δ ?s ll by force
      qed
      finally show ?thesis .
    qed
  } note tarjan-disc-aux = this

  show ?thesis
  proof (induct rule: establish-invarI-CB)
    case (new-root s s' v0)
    {
      fix w x
      let ?s = new-root v0 s(|state.more := x|)
      have lowlink-set ?s w ⊆ lowlink-set s w ∪ {v0}
        unfolding lowlink-set-def lowlink-path-def
        by auto
    } note * = this

    from new-root show ?case
      using tarjan-disc-aux[OF *]
      by (auto simp add: pw-leof-iff)
  next
    case (discover s s' u v) then interpret Tarjan-invar where s=s by simp
     let ?s' = discover (hd (stack s)) v (s(|pending := pending s − {(hd (stack
s),v)}|))
    {
      fix w x
      let ?s = ?s'(|state.more := x|)
      assume INV: Tarjan-invar G ?s
        and d: w ∈ dom (discovered ?s')
        and w≠v

      interpret s': Tarjan-invar where s=?s by fact
```

have *lowlink-set ?s w* ⊆ *lowlink-set s w* ∪ {*v*}
proof
fix *ll*
assume *ll*: *ll* ∈ *lowlink-set ?s w*
hence *ll* = *w* ∨ (∃ *p*. *lowlink-path ?s w p ll*) **by** (*auto simp add*: *lowlink-set-def*)
thus *ll* ∈ *lowlink-set s w* ∪ {*v*} (**is** *ll* ∈ *?L*)
proof
assume *ll* = *w* **with** *d* **show** *?thesis* **by** (*auto simp add*: *lowlink-set-def*)
next
assume ∃ *p*. *lowlink-path ?s w p ll*
then obtain *p* **where** *p*: *lowlink-path ?s w p ll* **..**

hence [*simp*]: *p*≠[] **by** (*simp add*: *lowlink-path-def*)

from *p* **have** *hd p* = *w* **by** (*auto simp add*: *lowlink-path-def path-hd*)

show *?thesis*
proof (*rule tri-caseE*)
assume *v*≠*ll v* ∉ *set p* **hence** *lowlink-path s w p ll*
using *p* **by** (*auto simp add*: *lowlink-path-def*)
with *ll* **show** *?thesis* **by** (*auto simp add*: *lowlink-set-def*)
next
assume *v* = *ll* **thus** *?thesis* **by** *simp*
next
assume *v* ∈ *set p v* ≠ *ll*
then obtain *i* **where** *i*: *i* < *length p p*!*i* = *v*
by (*metis in-set-conv-nth*)
have *False*
proof (*cases i*)
case *0* **with** *i* **have** *hd p* = *v* **by** (*simp add*: *hd-conv-nth*)
with ‹*hd p* = *w*› ‹*w* ≠ *v*› **show** *False* **by** *simp*
next
case (*Suc n*) **with** *i s'*.*lowlink-path-finished*[*OF p*, **where** *j*=*i*] **have**
v ∈ *dom* (*finished ?s*) **by** *simp*
with *finished-discovered discover* **show** *False* **by** *auto*
qed
thus *?thesis* **..**
qed
qed
qed
} **note** ∗ = *this*

from *discover hd-in-set stack-set-def* **have** *v* ≠ *u* **by** *auto*
with *discover* **have** ∗∗: *tree-edges ?s'* = *tree-edges s* ∨ (∃ *u*. *u* ≠ *v* ∧ *tree-edges ?s'* = *tree-edges s* ∪ {(*u*,*v*)}) **by** *auto*

from *discover* **show** *?case*
using *tarjan-disc-aux*[*OF* ∗ ∗∗]

by (*auto simp*: *pw-leof-iff*)
next
case (*cross-back-edge s s' u v*) **then interpret** *Tarjan-invar* **where** *s=s* **by** *simp*
from *cross-back-edge* **have** [*simp*]:
discovered s' = discovered s
finished s' = finished s
tree-edges s' = tree-edges s
lowlink s' = lowlink s
by *simp-all*
{
fix w :: $'v$
fix x

let *?s = s'*(|*state.more* := x|)
let *?L* = δ *s* ‘ *lowlink-set s w*
let *?L'* = δ *?s* ‘ *lowlink-set ?s w*

assume *TRANS*: $\bigwedge\Psi$. *tarjan-back u v s'* $\leq_n$ *SPEC* $\Psi \Longrightarrow \Psi$ x
and *inv'*: *DFS-invar G tarjan-params ?s*
and *w-disc'*: $w \in$ *dom* (*discovered ?s*)

from *inv'* **interpret** *s'*:*Tarjan-invar* **where** *s=?s* **by** *simp*

have *ll-sub*: *lowlink-set s w* $\subseteq$ *lowlink-set ?s w*
unfolding *lowlink-set-def lowlink-path-def*
by (*auto simp*: *cross-back-edge*)

have *ll-sub-rev*: *lowlink-set ?s w* $\subseteq$ *lowlink-set s w* $\cup$ $\{v\}$
unfolding *lowlink-set-def lowlink-path-def*
by (*auto simp*: *cross-back-edge*)

from *w-disc'* **have** *w-disc*: $w \in$ *dom* (*discovered s*) **by** *simp*
with *LowLink-le-disc* **have** *LLw*: *LowLink s w* $\leq \delta$ *s w* **by** *simp*

from *cross-back-edge hd-in-set* **have** *u-n-fin*: $u \notin$ *dom* (*finished s*)
using *stack-not-finished* **by** *auto*

{
assume $*$: $v \in$ *lowlink-set ?s w* $\Longrightarrow$ *LowLink s w* $\leq \delta$ *?s v*
have *LowLink s w = LowLink ?s w*
proof (*rule LowLink-eqI*[*OF inv' - ll-sub ll-sub-rev w-disc*])
show *discovered s* $\subseteq_m$ *discovered ?s* **by** *simp*

fix *ll* **assume** *ll* $\in \{v\}$ *ll* $\in$ *lowlink-set ?s w*
with $*$ **show** *LowLink s w* $\leq \delta$ *?s ll* **by** *simp*
qed
} **note** *LL-eqI = this*

have ζ *?s w* = *LowLink ?s w*
proof (*cases w*=*u*)
case *True* **show** *?thesis*
proof (*cases* (δ *s v* < δ *s w* $\wedge$ *v* $\in$ *set* (*tj-stack s*) $\wedge$ δ *s v* < ζ *s w*))
case *False* **note** *all-False* = *this*
with ‹*w* = *u*› **have** ζ *?s w* = ζ *s w*
by (*rule-tac TRANS*) (*auto simp add*: *tarjan-back-def cross-back-edge*)
also from *cross-back-edge w-disc* **have** ζ*w*: ... = *LowLink s w* **by** *simp*
also have *LowLink s w* = *LowLink ?s w*
proof (*rule LL-eqI*)
assume *v*: *v* $\in$ *lowlink-set ?s w*
show *LowLink s w* $\leq$ δ *?s v*
proof (*cases* δ *s v* < δ *s w* $\wedge$ δ *s v* < ζ *s w*)
case *False* **with** ‹*LowLink s w* $\leq$ δ *s w*› ζ*w* **show** *?thesis* **by** *auto*
next
case *True* **with** *all-False* **have** *v-n-tj*: *v* $\notin$ *set* (*tj-stack s*) **by** *simp*
from *v* **have** *e*: (*v*,*u*) $\in$ E^* (*u*,*v*) $\in$ E^*
unfolding *lowlink-set-def* **by** (*auto simp add*: ‹*w*=*u*›)

from *v-n-tj* **have** *v* $\notin$ *set* (*stack s*) **using** *stack-ss-tj-stack* **by** *auto*
with *cross-back-edge* **have** *v* $\in$ *dom* (*finished s*) **by** (*auto simp add*: *stack-set-def*)
with *finished-ss-sccs-tj-stack v-n-tj sccs-are-sccs* **obtain** *scc*
where *scc*: *v* $\in$ *scc scc* $\in$ *sccs s is-scc E scc* **by** *blast*
with *is-scc-closed e* **have** *u* $\in$ *scc* **by** *metis*
with *scc sccs-finished u-n-fin* **have** *False* **by** *blast*
thus *?thesis* **..**
qed
qed
finally show *?thesis* **.**
next
case *True* **note** *all-True* = *this*
with ‹*w*=*u*› **have** ζ *?s w* = δ *s v*
by (*rule-tac TRANS*) (*simp add*: *tarjan-back-def cross-back-edge*)

also from *True cross-back-edge w-disc* **have** δ *s v* < *LowLink s w* **by** *simp*
with *lowlink-set-finite lowlink-set-not-empty w-disc* **have** δ *s v* = *Min* (*?L* $\cup$ {δ *s v*}) **by** *simp*
also have *v* $\in$ *lowlink-set ?s w*
proof −
have *cb*: (*u*,*v*) $\in$ *cross-edges ?s* $\cup$ *back-edges ?s* **by** (*simp add*: *cross-back-edge*)
with *s′.lowlink-path-single* **have** *lowlink-path ?s u* [*u*] *v* **by** *auto*
moreover from *cb s′.cross-edges-ssE s′.back-edges-ssE* **have** (*u*,*v*) $\in$ *E* **by** *blast*
hence (*u*,*v*) $\in$ E^* **..**
moreover from *all-True tj-stack-reach-hd-stack* **have** (*v*,*u*) $\in$ E^* **by** (*simp add*: *cross-back-edge*)

moreover note ‹$v \in$ *dom* (*discovered s*)›
ultimately show *?thesis* **by** (*auto intro*: *s'.lowlink-setI simp*: ‹*w=u*›)
qed
with *ll-sub ll-sub-rev* **have** *lowlink-set ?s w* = *lowlink-set s w* ∪ {*v*} **by** *auto*
hence *Min* (*?L* ∪ {δ *s v*}) = *LowLink ?s w* **by** *simp*
finally show *?thesis* **.**
qed
next
case *False* — *w* ≠ *u*
hence ζ *?s w* = ζ *s w*
by (*rule-tac TRANS*) (*simp add*: *tarjan-back-def cross-back-edge*)
also have ζ *s w* = *LowLink s w* **using** *w-disc False* **by** (*simp add*: *cross-back-edge*)
also have *LowLink s w* = *LowLink ?s w*
proof (*rule LL-eqI*)
assume *v*: *v* ∈ *lowlink-set ?s w*
thus *LowLink s w* ≤ δ *?s v* **using** *LLw*
proof *cases*
assume *v* ≠ *w*
with *v* **obtain** *p* **where** *p*: *lowlink-path ?s w p v p*≠[]
by (*auto simp add*: *lowlink-set-def lowlink-path-def*)
hence *hd p* = *w* **by** (*auto simp add*: *lowlink-path-def path-hd*)

show *?thesis*
proof (*cases u* ∈ *set p*)
case *False* **with** *last-in-set p cross-back-edge* **have** *last p* ≠ *hd* (*stack s*) **by** *force*
with *p* **have** *lowlink-path s w p v*
by (*auto simp*: *cross-back-edge lowlink-path-def*)
with *v* **have** *v* ∈ *lowlink-set s w*
by (*auto intro*: *lowlink-setI simp*: *lowlink-set-def cross-back-edge*)
thus *?thesis* **by** *simp*
next
case *True* **then obtain** *i* **where** *i*: *i* < *length p p*!*i* = *u*
by (*metis in-set-conv-nth*)
have *False*
proof (*cases i*)
case *0* **with** *i* **have** *hd p* = *u* **by** (*simp add*: *hd-conv-nth*)
with ‹*hd p* = *w*› ‹*w* ≠ *u*› **show** *False* **by** *simp*
next
case (*Suc n*) **with** *i s'.lowlink-path-finished*[*OF p*(*1*), **where** *j=i*] **have**
u ∈ *dom* (*finished ?s*) **by** *simp*
with *u-n-fin* **show** *?thesis* **by** *simp*
qed
thus *?thesis* **..**
qed
qed *simp*

qed
finally show *?thesis* **.**
qed
} **note** *aux* = *this*

with *cross-back-edge* **show** *?case* **by** (*auto simp*: *pw-leof-iff*)
next
case (*finish s s′ u*) **then interpret** *Tarjan-invar* **where** *s*=*s* **by** *simp*
from *finish* **have** [*simp*]:
discovered s′ = *discovered s*
finished s′ = (*finished s*)(*u*↦*counter s*)
tree-edges s′ = *tree-edges s*
back-edges s′ = *back-edges s*
cross-edges s′ = *cross-edges s*
lowlink s′ = *lowlink s tj-stack s′* = *tj-stack s*
by *simp-all*

from *finish hd-in-set stack-discovered* **have** *u-disc*: $u \in$ *dom* (*discovered s*) **by** *blast*

{
fix *w* :: ′*v*
fix *x*

let *?s* = *s′*(|*state.more* := *x*|)
let *?L* = δ *s* ‘ *lowlink-set s w*
let *?Lu* = δ *s* ‘ *lowlink-set s u*
let *?L′* = δ *s* ‘ *lowlink-set ?s w*

assume *TRANS*: $\bigwedge\Psi$. *tarjan-fin u s′* $\leq_n$ *SPEC* $\Psi \Longrightarrow \Psi$ *x*
and *inv′*: *DFS-invar G tarjan-params ?s*
and *w-disc*: $w \in$ *dom* (*discovered ?s*)

from *inv′* **interpret** *s′*:*Tarjan-invar* **where** *s*=*?s* **by** *simp*

have *ll-sub*: *lowlink-set s w* ⊆ *lowlink-set ?s w*
unfolding *lowlink-set-def lowlink-path-def*
by *auto*

have *ll-sub-rev*: *lowlink-set ?s w* ⊆ *lowlink-set s w* ∪ *lowlink-set s u*
proof
fix *ll*
assume *ll*: *ll* ∈ *lowlink-set ?s w*
hence *ll* = *w* ∨ (∃ *p*. *lowlink-path ?s w p ll*) **by** (*auto simp add*: *lowlink-set-def*)
thus *ll* ∈ *lowlink-set s w* ∪ *lowlink-set s u*
proof (*rule disjE1*)
assume *ll* = *w* **with** *w-disc* **show** *?thesis* **by** (*auto simp add*: *lowlink-set-def*)

```
next
  assume ll ≠ w
  assume ∃ p. lowlink-path ?s w p ll
  then obtain p where p: lowlink-path ?s w p ll ..

  hence [simp]: p≠[] by (simp add: lowlink-path-def)

  from p have hd p = w by (auto simp add: lowlink-path-def path-hd)

  show ?thesis
  proof (cases u ∈ set p)
    case False hence lowlink-path s w p ll
      using p by (auto simp add: lowlink-path-def)
    with ll show ?thesis by (auto simp add: lowlink-set-def)
  next
    case True
    then obtain i where i: i < length p p!i = u
      by (metis in-set-conv-nth)
    moreover
    let ?dp = drop i p

    from i have ?dp ≠ [] by simp

    from i have hd ?dp = u by (simp add: hd-drop-conv-nth)
    moreover from i have last ?dp = last p by simp
    moreover {
      fix k
      assume 1 < length ?dp
      and  k < length ?dp − 1

      hence l: 1 < length p k+i < length p − 1 by (auto)
        with p have (p!(k+i), p!Suc (k+i)) ∈ tree-edges s by (auto simp
add: lowlink-path-def)
       moreover from l have i: i+k ≤ length p i+Suc k ≤ length p by
simp-all
        ultimately have  (?dp!k,?dp!Suc k) ∈ tree-edges s by (simp add:
add.commute)
    } note aux = this
    moreover {
      assume ∗: 1 < length ?dp
      hence l: 1 + i < length p by simp
       with s'.lowlink-path-finished[OF p] have p ! (1+i) ∈ dom (finished
?s) by auto
      moreover from l have i+1≤length p by simp
      ultimately have ?dp!1 ∈ dom (finished ?s) by simp
       moreover from aux[of 0] ∗ have (?dp!0,?dp!Suc 0) ∈ tree-edges s
by simp
          with ‹hd ?dp = u› hd-conv-nth[of ?dp] ∗ have (u,?dp!Suc 0) ∈
tree-edges s by simp
```

with *no-self-loop-in-tree* **have** $?dp!1 \neq u$ **by** *auto*
ultimately have $?dp!1 \in dom\ (finished\ s)$ **by** *simp*
}
moreover
from p **have** P: *path E w p ll* **by** (*simp add*: *lowlink-path-def*)

have $p = (take\ i\ p)@?dp$ **by** *simp*
with P *path-conc-conv* **obtain** x **where** p': *path E x ?dp ll path E w* (*take i p*) x **by** *metis*
with ‹$?dp \neq []$› *path-hd* **have** $hd\ ?dp = x$ **by** *metis*
with ‹$hd\ ?dp = u$› p' **have** *u-path*: *path E u ?dp ll* **and** *path-u*: *path E w* (*take i p*) u **by** *metis+*

ultimately have *lowlink-path s u ?dp ll* **using** p **by** (*simp add*: *lowlink-path-def*)
moreover from *u-path path-is-trancl* ‹$?dp \neq []$› **have** $(u,ll) \in E^+$ **by** *force*
moreover { from ll ‹$ll \neq w$› **have** $(ll,w) \in E^+$ **by** (*auto simp add*: *lowlink-set-def*)
also from *path-u path-is-rtrancl* **have** $(w,u) \in E^*$ **by** *metis*
finally have $(ll,u)\in E^+$ **.**
}
moreover note *ll u-disc*
ultimately have $ll \in$ *lowlink-set s u* **unfolding** *lowlink-set-def* **by** *auto*
thus *?thesis* **by** *auto*
qed
qed
qed
hence *ll-sub-rev'*: $?L' \subseteq ?L \cup ?Lu$ **by** *auto*

have *ref-ne*: $stack\ ?s \neq [] \Longrightarrow$
$lowlink\ ?s = (lowlink\ s)(hd\ (stack\ ?s) \mapsto min\ (\zeta\ s\ (hd\ (stack\ ?s)))\ (\zeta\ s\ u))$
apply (*rule TRANS*)
unfolding *tarjan-fin-def tj-stack-pop-def*
by *refine-vcg simp-all*

have *ref-e*: $stack\ ?s = [] \Longrightarrow lowlink\ ?s = lowlink\ s$
apply (*rule TRANS*)
unfolding *tarjan-fin-def tj-stack-pop-def*
by *refine-vcg simp-all*

have *ref-tj*: $\zeta\ s\ u \neq \delta\ s\ u \Longrightarrow tj\text{-}stack\ ?s = tj\text{-}stack\ s$
apply (*rule TRANS*)
unfolding *tarjan-fin-def tj-stack-pop-def*
by *refine-vcg simp-all*

have $\zeta\ ?s\ w = LowLink\ ?s\ w$

proof (*cases* $w = hd\ (stack\ ?s) \wedge stack\ ?s \neq []$)
case *True* **note** *all-True* = *this*
with *ref-ne* **have** $*$: $\zeta\ ?s\ w = min\ (\zeta\ s\ w)\ (\zeta\ s\ u)$ **by** *simp*
show *?thesis*
proof (*cases* $\zeta\ s\ u < \zeta\ s\ w$)
case *False* **with** $*$ *finish w-disc* **have** $\zeta\ ?s\ w = LowLink\ s\ w$ **by** *simp*
also have $LowLink\ s\ w = LowLink\ ?s\ w$
proof (*rule LowLink-eqI*[*OF inv′ - ll-sub ll-sub-rev*])
from *w-disc* **show** $w \in dom\ (discovered\ s)$ **by** *simp*
fix *ll* **assume** $ll \in lowlink\text{-}set\ s\ u$
hence $LowLink\ s\ u \leq \delta\ s\ ll$ **by** *simp*
moreover from *False finish w-disc u-disc* **have** $LowLink\ s\ w \leq LowLink$
$s\ u$ **by** *simp*
ultimately show $LowLink\ s\ w \leq \delta\ ?s\ ll$ **by** *simp*
qed *simp*
finally show *?thesis* **.**
next
case *True* **note** ζrel = *this*
have $LowLink\ s\ u \in\ ?L'$
proof −
from *all-True finish* **have** *w-tl*: $w \in set\ (tl\ (stack\ s))$ **by** *auto*

obtain *ll* **where** *ll*: $ll \in lowlink\text{-}set\ s\ u\ \ \delta\ s\ ll = LowLink\ s\ u$
using *Min-in*[*of ?Lu*] *lowlink-set-finite lowlink-set-not-empty u-disc*
by *fastforce*
have $ll \in lowlink\text{-}set\ ?s\ w$
proof (*cases* $\delta\ s\ u = \zeta\ s\ u$)
case *True*
moreover from *w-tl finish tl-lt-stack-hd-discover* **have** $\delta\ s\ w < \delta\ s\ u$
by *simp*
moreover from *w-disc* **have** $LowLink\ s\ w \leq \delta\ s\ w$ **by** (*simp add*:
LowLink-le-disc)
with *w-disc finish* **have** $\zeta\ s\ w \leq \delta\ s\ w$ **by** *simp*
moreover note ζrel
ultimately have *False* **by** *force*
thus *?thesis* **..**
next
case *False* **with** *u-disc finish ll* **have** $u \neq ll$ **by** *auto*
with *ll* **have**
e: $(ll,u) \in E^+\ \ (u,ll) \in E^+$ **and**
p: $\exists p.\ lowlink\text{-}path\ s\ u\ p\ ll$ **and**
ll-disc: $ll \in dom\ (discovered\ s)$
by (*auto simp*: *lowlink-set-def*)

from *p* **have** *p′*: $\exists p.\ lowlink\text{-}path\ ?s\ u\ p\ ll$
unfolding *lowlink-path-def*
by *auto*
from *w-tl tl-stack-hd-tree-path finish* **have** *T*: $(w,u) \in (tree\text{-}edges\ ?s)^+$
by *simp*

with s'*.lowlink-path-tree-prepend all-True* p' **have** $\exists\, p.$ *lowlink-path ?s w p ll* **by** *blast*

moreover from *T trancl-mono-mp*[*OF* s'*.tree-edges-ssE*] **have** $(w,u) \in E^+$ **by** *blast*

with *e* **have** $(w,ll) \in E^+$ **by** *simp*

moreover {

note *e*(*1*)

also from *finish False ref-tj* **have** *tj-stack ?s = tj-stack s* **by** *simp*

with *hd-in-set finish stack-ss-tj-stack* **have** $u \in$ *set* (*tj-stack ?s*) **by** *auto*

with s'*.tj-stack-reach-stack* **obtain** *x* **where** *x*: $x \in$ *set* (*stack ?s*) $(u,x) \in E^*$ **by** *blast*

note *this*(*2*)

also have $(x,w) \in E^*$

proof (*rule rtrancl-eq-or-trancl*[*THEN iffD2*], *safe*)

assume $x \neq w$ **with** *all-True x* **have** $x \in$ *set* (*tl* (*stack ?s*)) **by** (*cases stack ?s*) *auto*

with s'*.tl-stack-hd-tree-path all-True* **have** $(x,w) \in$ (*tree-edges s*)$^+$ **by** *auto*

with *trancl-mono-mp*[*OF tree-edges-ssE*] **show** $(x,w) \in E^+$ **by** *simp*

qed

finally have $(ll,w) \in E^+$ **.**

}

moreover note *ll-disc*

ultimately show *?thesis* **by** (*simp add*: *lowlink-set-def*)

qed

hence δ *s ll* $\in$ *?L'* **by** *auto*

with *ll* **show** *?thesis* **by** *simp*

qed

hence *LowLink ?s w* $\leq$ *LowLink s u*

using *Min-le-iff*[*of ?L'*] s'*.lowlink-set-not-empty w-disc* s'*.lowlink-set-finite*

by *fastforce*

also from *True u-disc w-disc finish* **have** *LowLink s u* $<$ *LowLink s w* **by** *simp*

hence *Min* (*?L* $\cup$ *?Lu*) = *LowLink s u*

using *Min-Un*[*of ?L ?Lu*] *lowlink-set-finite lowlink-set-not-empty u-disc w-disc*

by *simp*

hence *LowLink s u* $\leq$ *LowLink ?s w*

using *Min-antimono*[*OF ll-sub-rev'*] *lowlink-set-finite* s'*.lowlink-set-not-empty w-disc*

by *auto*

also from *True u-disc finish* $*$ **have** *LowLink s u* = ζ *?s w* **by** *simp*

finally show *?thesis* **..**

qed

next

case *False* **note** *all-False* = *this*

have ζ *?s w* = ζ *s w*

proof (*cases stack ?s* = [])
case *True* **with** *ref-e* **show** *?thesis* **by** *simp*
next
case *False* **with** *all-False* **have** $w \neq hd$ (*stack ?s*) **by** *simp*
with *False ref-ne* **show** *?thesis* **by** *simp*
qed
also from *finish* **have** ζ *s w* = *LowLink s w* **using** *w-disc* **by** *simp*
also {
fix *v*
assume $v \in$ *lowlink-set s u*
and ∗: $v \notin$ *lowlink-set s w*
hence $v \neq w$ $w{\neq}u$ **by** (*auto simp add*: *lowlink-set-def*)
have $v \notin$ *lowlink-set ?s w*
proof (*rule notI*)
assume *v*: $v \in$ *lowlink-set ?s w*
hence *e*: $(v,w) \in E^*$ $(w,v) \in E^*$
and *v-disc*: $v \in$ *dom* (*discovered s*) **by** (*auto simp add*: *lowlink-set-def*)

from *v* ‹$v{\neq}w$› **obtain** *p* **where** *p*: *lowlink-path ?s w p v* **by** (*auto simp add*: *lowlink-set-def*)
hence [*simp*]: $p{\neq}[]$ **by** (*simp add*: *lowlink-path-def*)

from *p* **have** *hd p* = *w* **by** (*auto simp add*: *lowlink-path-def path-hd*)

show *False*
proof (*cases* $u \in$ *set p*)
case *False* **hence** *lowlink-path s w p v*
using *p* **by** (*auto simp add*: *lowlink-path-def*)
with *e v-disc* **have** $v \in$ *lowlink-set s w* **by** (*auto intro*: *lowlink-setI*)
with ∗ **show** *False* **..**
next
case *True*
then obtain *i* **where** *i*: *i* < *length p p*!*i* = *u*
by (*metis in-set-conv-nth*)
show *False*
proof (*cases i*)
case *0* **with** *i* **have** *hd p* = *u* **by** (*simp add*: *hd-conv-nth*)
with ‹*hd p* = *w*› ‹$w \neq u$› **show** *False* **by** *simp*
next
case (*Suc n*) **with** *i p* **have** ∗: (*p*!*n*,*u*) $\in$ *tree-edges s n* < *length p*
unfolding *lowlink-path-def*
by *auto*
with *tree-edge-imp-discovered* **have** *p*!*n* $\in$ *dom* (*discovered s*) **by** *auto*
moreover from *finish hd-in-set stack-not-finished* **have** $u \notin$ *dom* (*finished s*) **by** *auto*
with ∗ **have** *pn-n-fin*: *p*!*n* $\notin$ *dom* (*finished s*) **by** (*metis tree-edge-impl-parenthesis*)
moreover from ∗ *no-self-loop-in-tree* **have** *p*!*n* $\neq$ *u* **by** *blast*

ultimately have *p*!*n* ∈ *set* (*stack ?s*) **using** *stack-set-def finish* **by** (*cases stack s*) *auto*
hence *s-ne*: *stack ?s* ≠ [] **by** *auto*
with *all-False* **have** *w* ≠ *hd* (*stack ?s*) **by** *simp*
from *stack-is-tree-path finish* **obtain** *v0* **where**
path (*tree-edges s*) *v0* (*rev* (*stack ?s*)) *u*
by *auto*
with *s-ne* **have** (*hd* (*stack ?s*), *u*) ∈ *tree-edges s* **by** (*auto simp*: *neq-Nil-conv path-simps*)
with ∗ *tree-eq-rule* **have** ∗∗: *hd* (*stack ?s*) = *p*!*n* **by** *simp*
show *?thesis*
proof (*cases n*)
case *0* **with** ∗ **have** *hd p* = *p*!*n* **by** (*simp add*: *hd-conv-nth*)
with ‹*hd p* = *w*› ∗∗ **have** *w* = *hd* (*stack ?s*) **by** *simp*
with ‹*w*≠*hd* (*stack ?s*)› **show** *False* **..**
next
case (*Suc m*) **with** ∗ ∗∗ *s*′.*lowlink-path-finished*[*OF p*, **where** *j*=*n*] **have**
hd (*stack ?s*) ∈ *dom* (*finished ?s*) **by** *simp*
with *hd-in-set*[*OF s-ne*] *s*′.*stack-not-finished* **show** *?thesis* **by** *blast*
qed
qed
qed
qed
} **with** *ll-sub ll-sub-rev* **have** *lowlink-set ?s w* = *lowlink-set s w* **by** *auto*
hence *LowLink s w* = *LowLink ?s w* **by** *simp*
finally show *?thesis* **.**
qed
}

with *finish* **show** *?case* **by** (*auto simp*: *pw-leof-iff*)
qed *simp-all*
qed
end end

context *Tarjan-invar* **begin context begin interpretation** *timing-syntax* **.**

lemmas *lowlink-eq-LowLink* =
i-lowlink-eq-LowLink[*THEN make-invar-thm*, *rule-format*]

lemma *lowlink-eq-disc-iff-scc-root*:
assumes *v* ∈ *dom* (*finished s*) ∨ (*stack s* ≠ [] ∧ *v* = *hd* (*stack s*) ∧ *pending s* `` {*v*} = {})
shows ζ *s v* = δ *s v* ⟷ *scc-root s v* (*scc-of E v*)
proof −
from *assms* **have** *v* ∈ *dom* (*discovered s*) **using** *finished-discovered hd-in-set stack-discovered* **by** *blast*
hence ζ *s v* = *LowLink s v* **using** *lowlink-eq-LowLink* **by** *simp*
with *LowLink-eq-disc-iff-scc-root*[*OF assms*] **show** *?thesis* **by** *simp*

qed

lemma *nc-sccs-eq-reachable*:
assumes *NC*: $\neg$ *cond s*
shows *reachable* = $\bigcup$(*sccs s*)
proof
from *nc-finished-eq-reachable NC* **have** [*simp*]: *reachable* = *dom* (*finished s*) **by** *simp*
with *sccs-finished* **show** $\bigcup$(*sccs s*) $\subseteq$ *reachable* **by** *simp*

from *NC* **have** *stack s* = [] **by** (*simp add*: *cond-alt*)
with *stacks-eq-iff* **have** *tj-stack s* = [] **by** *simp*
with *finished-ss-sccs-tj-stack* **show** *reachable* $\subseteq$ $\bigcup$(*sccs s*) **by** *simp*
qed
end end

context *Tarjan* **begin**
lemma *tarjan-fin-nofail*:
assumes *pre-on-finish u s'*
shows *nofail* (*tarjan-fin u s'*)
proof −
from *assms* **obtain** *s* **where** *s*: *DFS-invar G tarjan-params s stack s* $\neq$ [] *u* = *hd* (*stack s*) *s'* = *finish u s* *cond s pending s* ‘‘ {*u*} = {}
by (*auto simp*: *pre-on-finish-def*)
then interpret *Tarjan-invar* **where** *s*=*s* **by** *simp*

from *s hd-stack-in-tj-stack* **have** *u* $\in$ *set* (*tj-stack s'*) **by** *simp*

moreover from *s tj-stack-distinct* **have** *distinct* (*tj-stack s'*) **by** *simp*
moreover have *the* (*lowlink s' u*) = *the* (*discovered s' u*) $\longleftrightarrow$ *scc-root s' u* (*scc-of E u*)
proof −
from *s* **have** *the* (*lowlink s' u*) = *the* (*discovered s' u*) $\longleftrightarrow$ *the* (*lowlink s u*) = *the* (*discovered s u*) **by** *simp*
also from *s lowlink-eq-disc-iff-scc-root* **have** ... $\longleftrightarrow$ *scc-root s u* (*scc-of E u*) **by** *blast*
also from *s scc-root-transfer'*[**where** *s'*=*s'*] **have** ... $\longleftrightarrow$ *scc-root s' u* (*scc-of E u*) **by** *simp*
finally show *?thesis* **.**
qed
ultimately show *?thesis*
unfolding *tarjan-fin-def tj-stack-pop-def*
by *simp*
qed

sublocale *DFS G tarjan-params*
by *unfold-locales* (*simp-all add*: *tarjan-disc-def tarjan-back-def tarjan-fin-nofail*)
end

interpretation *tarjan*: *Tarjan-def* **for** *G* **.**

2.7.4 Interface

definition *tarjan* $G \equiv do$ {
  $ASSERT$ (*fb-graph* G);
  $s \leftarrow$ *tarjan.it-dfs* $TYPE('a)$ G;
  $RETURN$ (*sccs* s) }

definition *tarjan-spec* $G \equiv do$ {
  $ASSERT$ (*fb-graph* G);
  $SPEC$ (λ*sccs*. ($\forall$ *scc* $\in$ *sccs*. *is-scc* (*g-E* G) *scc*)
        $\wedge \bigcup$ *sccs* $=$ *tarjan.reachable* $TYPE('a)$ G)}

lemma *tarjan-correct*:
  tarjan $G \leq$ *tarjan-spec* G
  unfolding *tarjan-def tarjan-spec-def*
proof (*refine-vcg le-ASSERTI order-trans*[*OF DFS.it-dfs-correct*])
  assume *fb-graph* G
  then interpret *fb-graph* G **.**
  interpret *Tarjan* **..**
  show *DFS* G (*tarjan.tarjan-params* $TYPE('b)$ G) **..**
next
  fix s
  assume C: *DFS-invar* G (*tarjan.tarjan-params* $TYPE('b)$ G) $s \wedge \neg$ *tarjan.cond* $TYPE('b)$ G s
  then interpret *Tarjan-invar* G s **by** *simp*

  from *sccs-are-sccs* **show** $\forall$ *scc*$\in$*sccs* s. *is-scc* (*g-E* G) *scc* **.**

  from *nc-sccs-eq-reachable* C **show** $\bigcup$(*sccs* s) $=$ *tarjan.reachable* $TYPE('b)$ G **by** *simp*
qed

end