# 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-scheme* $\Rightarrow ('v,'es)$ *state-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*]:
  $counter\ (discover\ u\ v\ s) = Suc\ (counter\ s)$
  $discovered\ (discover\ u\ v\ s) = (discovered\ s)(v \mapsto counter\ s)$
  $finished\ (discover\ u\ v\ s) = finished\ s$
  $stack\ (discover\ u\ v\ s) = v\#stack\ s$
  $pending\ (discover\ u\ v\ s) = pending\ s \cup \{v\} \times E``\{v\}$
  $tree\text{-}edges\ (discover\ u\ v\ s) = insert\ (u,v)\ (tree\text{-}edges\ s)$
  $cross\text{-}edges\ (discover\ u\ v\ s) = cross\text{-}edges\ s$
  $back\text{-}edges\ (discover\ u\ v\ s) = back\text{-}edges\ s$
  $state.more\ (discover\ u\ v\ s) = state.more\ s$
  $\langle proof \rangle$

**definition** *finish*
  $:: 'v \Rightarrow ('v,'es)$ *state-scheme* $\Rightarrow ('v,'es)$ *state-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*]:
  $counter\ (finish\ u\ s) = Suc\ (counter\ s)$
  $discovered\ (finish\ u\ s) = discovered\ s$
  $finished\ (finish\ u\ s) = (finished\ s)(u \mapsto counter\ s)$
  $stack\ (finish\ u\ s) = tl\ (stack\ s)$
  $pending\ (finish\ u\ s) = pending\ s$
  $tree\text{-}edges\ (finish\ u\ s) = tree\text{-}edges\ s$
  $cross\text{-}edges\ (finish\ u\ s) = cross\text{-}edges\ s$
  $back\text{-}edges\ (finish\ u\ s) = back\text{-}edges\ s$
  $state.more\ (finish\ u\ s) = state.more\ s$
  $\langle proof \rangle$

**definition** *back-edge*
  $:: 'v \Rightarrow 'v \Rightarrow ('v,'es)$ *state-scheme* $\Rightarrow ('v,'es)$ *state-scheme*
**where**
*back-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*]:
$counter\ (back\text{-}edge\ u\ v\ s) = counter\ s$
$discovered\ (back\text{-}edge\ u\ v\ s) = discovered\ s$
$finished\ (back\text{-}edge\ u\ v\ s) = finished\ s$
$stack\ (back\text{-}edge\ u\ v\ s) = stack\ s$
$pending\ (back\text{-}edge\ u\ v\ s) = pending\ s$
$tree\text{-}edges\ (back\text{-}edge\ u\ v\ s) = tree\text{-}edges\ s$
$cross\text{-}edges\ (back\text{-}edge\ u\ v\ s) = cross\text{-}edges\ s$
$back\text{-}edges\ (back\text{-}edge\ u\ v\ s) = insert\ (u,v)\ (back\text{-}edges\ s)$
$state.more\ (back\text{-}edge\ u\ v\ s) = state.more\ s$
⟨*proof*⟩

**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*]:
$counter\ (cross\text{-}edge\ u\ v\ s) = counter\ s$
$discovered\ (cross\text{-}edge\ u\ v\ s) = discovered\ s$
$finished\ (cross\text{-}edge\ u\ v\ s) = finished\ s$
$stack\ (cross\text{-}edge\ u\ v\ s) = stack\ s$
$pending\ (cross\text{-}edge\ u\ v\ s) = pending\ s$
$tree\text{-}edges\ (cross\text{-}edge\ u\ v\ s) = tree\text{-}edges\ s$
$cross\text{-}edges\ (cross\text{-}edge\ u\ v\ s) = insert\ (u,v)\ (cross\text{-}edges\ s)$
$back\text{-}edges\ (cross\text{-}edge\ u\ v\ s) = back\text{-}edges\ s$
$state.more\ (cross\text{-}edge\ u\ v\ s) = state.more\ s$
⟨*proof*⟩

**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*
⟨*proof*⟩

**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*
⟨*proof*⟩

### 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* $\longleftrightarrow$ $(V0 \subseteq \{v.$ *is-discovered v s*$\}$ $\longrightarrow$ $\neg$*is-empty-stack s*$)$
$\wedge$ $\neg$*is-break param s*

**lemma** *cond-alt*:
*cond* $= (\lambda s.\ (V0 \subseteq$ *dom* $($*discovered s*$)$ $\longrightarrow$ *stack s* $\neq [])$ $\wedge$ $\neg$*is-break param s*$)$
⟨*proof*⟩

**definition** *get-pending* ::
$('v,\ 'es)$ *state-scheme* $\Rightarrow$ $('v \times 'v$ *option* $\times$ $('v,\ 'es)$ *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* ← *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* ← *get-new-root s*;
*do-new-root v0 s*
} *else do* {
(*u*, *Vs*,*s*) ← *get-pending s*;
*case Vs of*
*None* ⇒ *do-finish u s*
| *Some v* ⇒ *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
⟨*proof*⟩

**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.
  ⟨*proof*⟩

**lemma** *DFS-invar-step*: ⟦*nofail it-dfs*; *DFS-invar G param s*; *cond s*⟧
  ⟹ *step s* ≤ *SPEC* (*DFS-invar G param*)
  — A step preserves the (best) invariant.
  ⟨*proof*⟩

**lemma** *DFS-invar-step′*: ⟦*nofail* (*step s*); *DFS-invar G param s*; *cond s*⟧
  ⟹ *step s* ≤ *SPEC* (*DFS-invar G param*)
  ⟨*proof*⟩

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* ≡
  *DFS-invar G param s* ∧ *cond s* ∧
  *stack s* ≠ [] ∧ *u* = *hd* (*stack s*) ∧ (*u*, *v*) ∈ *pending s*

**definition** *pre-on-cross-edge u v s′* ≡ ∃ *s*. *pre-edge-selected u v s* ∧
    *v* ∈ *dom* (*discovered s*) ∧ *v*∈*dom* (*finished s*)
    ∧ *s′* = *cross-edge u v* (*s*⦇*pending* := *pending s* − {(*u*,*v*)}⦈)

**definition** *pre-on-back-edge u v s′* ≡ ∃ *s. pre-edge-selected u v s* ∧
*v* ∈ *dom* (*discovered s*) ∧ *v*∉*dom* (*finished s*)
∧ *s′* = *back-edge u v* (*s*⦇*pending* := *pending s* − {(*u*,*v*)}⦈)

**definition** *pre-on-discover u v s′* ≡ ∃ *s. pre-edge-selected u v s* ∧
*v* ∉ *dom* (*discovered s*)
∧ *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* $\le_n$ *SPEC* (λ*x. I* (*empty-state x*))
**assumes** *new-root*: ⋀*s s′ v0*.
⟦*DFS-invar G param s*; *I s*; *cond s*; ¬ *is-break param s*;
*stack s* = []; *v0* ∈ *V0*; *v0* ∉ *dom* (*discovered s*);
*s′* = *new-root v0 s*⟧
⟹ *on-new-root param v0 s′* $\le_n$
*SPEC* (λ*x. DFS-invar G param* (*s′*⦇*state.more* := *x*⦈)
⟶ *I* (*s′*⦇*state.more* := *x*⦈))
**assumes** *finish*: ⋀*s s′ u*.
⟦*DFS-invar G param s*; *I s*; *cond s*; ¬ *is-break param s*;
*stack s* ≠ []; *u* = *hd* (*stack s*);
*pending s* “ {*u*} = {};
*s′* = *finish u s*⟧
⟹ *on-finish param u s′* $\le_n$
*SPEC* (λ*x. DFS-invar G param* (*s′*⦇*state.more* := *x*⦈)
⟶ *I* (*s′*⦇*state.more* := *x*⦈))
**assumes** *cross-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′* = *cross-edge u v* (*s*⦇*pending* := *pending s* − {(*u*,*v*)}⦈)⟧
⟹ *on-cross-edge param u v s′* $\le_n$
*SPEC* (λ*x. DFS-invar G param* (*s′*⦇*state.more* := *x*⦈)
⟶ *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′* $\le_n$
*SPEC* (λ*x. DFS-invar G param* (*s′*⦇*state.more* := *x*⦈)
⟶ *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*⟩

**lemma** *establish-invarI'*[*case-names init new-root finish cross-edge back-edge discover*]:
— Establish a DFS invariant (symbolic preconditions).
**assumes** *init*: *on-init param* $\leq_n$ *SPEC* ($\lambda x.$ *I* (*empty-state x*))
**assumes** *new-root*: $\bigwedge s'\ v0.$ *pre-on-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*: $\bigwedge s'\ u.$ *pre-on-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*: $\bigwedge s'\ u\ v.$ *pre-on-cross-edge u v 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** *back-edge*: $\bigwedge s'\ u\ v.$ *pre-on-back-edge u v s'*
$\Longrightarrow$ *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* := $x|\!)$))
**assumes** *discover*: $\bigwedge s'\ u\ v.$ *pre-on-discover u v s'*
$\Longrightarrow$ *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*⟩

**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-discover param u v s* = *on-new-root param v s*
**assumes** *init*: *on-init param* $\leq_n$ *SPEC* ($\lambda x.$ *I* (*empty-state x*))
**assumes** *new-discover*: $\bigwedge s\ s'\ v.$
$[\![$*DFS-invar G param s*; *I s*; *cond s*; $\neg$ *is-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-edges* $s'$ = *back-edges s*; *cross-edges* $s'$ = *cross-edges s*;

$\quad$ *tree-edges* $s' \supseteq$ *tree-edges* $s$;
$\quad$ *state.more* $s'$ = *state.more* $s$⟧
$\quad\quad \Longrightarrow$ *on-new-root param* $v\ s' \leq_n$
$\quad\quad\quad$ *SPEC* ($\lambda x$. *DFS-invar G param* ($s'$⦇*state.more* := $x$⦈)
$\quad\quad\quad\quad \longrightarrow I$ ($s'$⦇*state.more* := $x$⦈))

**assumes** *finish*: $\bigwedge s\ s'\ u$.
⟦*DFS-invar G param s*; *I s*; *cond s*; ¬ *is-break param s*;
$\quad$ *stack s* $\neq$ []; $u$ = *hd* (*stack s*);
$\quad$ *pending s* ‘‘ $\{u\}$ = $\{\}$;
$\quad$ $s'$ = *finish u s*⟧
$\quad\quad \Longrightarrow$ *on-finish param* $u\ s' \leq_n$
$\quad\quad\quad$ *SPEC* ($\lambda x$. *DFS-invar G param* ($s'$⦇*state.more* := $x$⦈)
$\quad\quad\quad\quad \longrightarrow I$ ($s'$⦇*state.more* := $x$⦈))

**assumes** *cross-edge*: $\bigwedge s\ s'\ u\ v$.
⟦*DFS-invar G param s*; *I s*; *cond s*; ¬ *is-break param s*;
$\quad$ *stack s* $\neq$ []; $(u, v) \in$ *pending s*; $u$ = *hd* (*stack s*);
$\quad$ $v \in$ *dom* (*discovered s*); $v \in$ *dom* (*finished s*);
$\quad$ $s'$ = *cross-edge u v* ($s$⦇*pending* := *pending s* − $\{(u,v)\}$⦈)⟧
$\quad \Longrightarrow$ *on-cross-edge param* $u\ v\ s' \leq_n$
$\quad\quad$ *SPEC* ($\lambda x$. *DFS-invar G param* ($s'$⦇*state.more* := $x$⦈)
$\quad\quad\quad \longrightarrow I$ ($s'$⦇*state.more* := $x$⦈))

**assumes** *back-edge*: $\bigwedge s\ s'\ u\ v$.
⟦*DFS-invar G param s*; *I s*; *cond s*; ¬ *is-break param s*;
$\quad$ *stack s* $\neq$ []; $(u, v) \in$ *pending s*; $u$ = *hd* (*stack s*);
$\quad$ $v \in$ *dom* (*discovered s*); $v \notin$ *dom* (*finished s*);
$\quad$ $s'$ = *back-edge u v* ($s$⦇*pending* := *pending s* − $\{(u,v)\}$⦈)⟧
$\quad \Longrightarrow$ *on-back-edge param* $u\ v\ s' \leq_n$
$\quad\quad$ *SPEC* ($\lambda x$. *DFS-invar G param* ($s'$⦇*state.more* := $x$⦈)
$\quad\quad\quad \longrightarrow I$ ($s'$⦇*state.more* := $x$⦈))

**shows** *is-invar I*
⟨*proof*⟩

**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*; ¬ *is-break param s*;
$\quad$ *stack s* = []; $v0 \in V0$; $v0 \notin$ *dom* (*discovered s*);
$\quad$ $s'$ = *new-root v0 s*⟧
$\quad\quad \Longrightarrow$ *on-new-root param* $v0\ s' \leq_n$
$\quad\quad\quad$ *SPEC* ($\lambda x$. *DFS-invar G param* ($s'$⦇*state.more* := $x$⦈)
$\quad\quad\quad\quad \longrightarrow I$ ($s'$⦇*state.more* := $x$⦈))

**assumes** *finish*: $\bigwedge s\ s'\ u$.
⟦*DFS-invar G param s*; *I s*; *cond s*; ¬ *is-break param s*;
$\quad$ *stack s* $\neq$ []; $u$ = *hd* (*stack s*);
$\quad$ *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*⟩

**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-invar G param s*; *I s*; *cond s*; $\neg$ *is-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-edges s'* = *back-edges s*; *cross-edges s'* = *cross-edges s*;
*tree-edges s'* $\supseteq$ *tree-edges s*;
*state.more s'* = *state.more s*⟧
$\Longrightarrow$ *on-new-root param v s'* $\leq_n$
*SPEC* ($\lambda x$. *DFS-invar G param* ($s'$⦇*state.more* := $x$⦈)
$\longrightarrow$ *I* ($s'$⦇*state.more* := $x$⦈))

**assumes** *finish*: $\bigwedge s\ s'\ u$.
⟦*DFS-invar G param s*; *I s*; *cond s*; $\neg$ *is-break param s*;

$stack\ s \neq []$; $u = hd\ (stack\ s)$;
$pending\ s\ ``\ \{u\} = \{\}$;
$s' = finish\ u\ s⟧$
$\Longrightarrow on\text{-}finish\ param\ u\ s' \leq_n$
$SPEC\ (\lambda x.\ DFS\text{-}invar\ G\ param\ (s'⦇state.more := x⦈)$
$\longrightarrow I\ (s'⦇state.more := x⦈))$

**assumes** *cross-back-edge*: $\bigwedge s\ s'\ u\ v.$
$⟦DFS\text{-}invar\ G\ param\ s;\ I\ s;\ cond\ s;\ \neg\ is\text{-}break\ param\ s;$
$stack\ s \neq [];\ (u,\ v) \in pending\ s;\ u = hd\ (stack\ s);$
$v \in dom\ (discovered\ s);$
$discovered\ s' = discovered\ s;\ finished\ s' = finished\ s;$
$stack\ s' = stack\ s;\ tree\text{-}edges\ s' = tree\text{-}edges\ s;\ counter\ s' = counter\ s;$
$pending\ s' = pending\ s - \{(u,v)\};$
$cross\text{-}edges\ s' \cup back\text{-}edges\ s' = cross\text{-}edges\ s \cup back\text{-}edges\ s \cup \{(u,v)\};$
$state.more\ s' = state.more\ s\ ⟧$
$\Longrightarrow on\text{-}cross\text{-}edge\ param\ u\ v\ s' \leq_n$
$SPEC\ (\lambda x.\ DFS\text{-}invar\ G\ param\ (s'⦇state.more := x⦈)$
$\longrightarrow I\ (s'⦇state.more := x⦈))$

**shows** *is-invar I*
⟨*proof*⟩

**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*: $\bigwedge s\ s'\ v0\ e.$
$⟦I\ s;\ cond\ s;\ DFS\text{-}invar\ G\ param\ s;\ DFS\text{-}invar\ G\ param\ s';$
$stack\ s = [];\ v0 \notin dom\ (discovered\ s);\ v0 \in V0;$
$s' = new\text{-}root\ v0\ s⦇state.more := e⦈⟧$
$\Longrightarrow I\ s'$
**and** *finish*: $\bigwedge s\ s'\ u\ e.$
$⟦I\ s;\ cond\ s;\ DFS\text{-}invar\ G\ param\ s;\ DFS\text{-}invar\ G\ param\ s';$
$stack\ s \neq [];\ pending\ s\ ``\ \{u\} = \{\};$
$u = hd\ (stack\ s);\ s' = finish\ u\ s⦇state.more := e⦈⟧$
$\Longrightarrow I\ s'$
**and** *cross-edge*: $\bigwedge s\ s'\ u\ v\ e.$
$⟦I\ s;\ cond\ s;\ DFS\text{-}invar\ G\ param\ s;\ DFS\text{-}invar\ G\ param\ s';$
$stack\ s \neq [];\ v \in pending\ s\ ``\ \{u\};\ v \in dom\ (discovered\ s);$
$v \in dom\ (finished\ s);$
$u = hd\ (stack\ s);$
$s' = (cross\text{-}edge\ u\ v\ (s⦇pending := pending\ s - \{(u,v)\}⦈))⦇state.more := e⦈⟧$
$\Longrightarrow I\ s'$
**and** *back-edge*: $\bigwedge s\ s'\ u\ v\ e.$
$⟦I\ s;\ cond\ s;\ DFS\text{-}invar\ G\ param\ s;\ DFS\text{-}invar\ G\ param\ s';$
$stack\ s \neq [];\ v \in pending\ s\ ``\ \{u\};\ v \in dom\ (discovered\ s);\ v \notin dom\ (finished\ s);$
$u = hd\ (stack\ s);$
$s' = (back\text{-}edge\ u\ v\ (s⦇pending := pending\ s - \{(u,v)\}⦈))⦇state.more := e⦈⟧$

$\Longrightarrow I\ s'$

**and** *discover*: $\bigwedge s\ s'\ u\ v\ e.$

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

**shows** *is-invar I*
⟨*proof*⟩

**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'*: $\bigwedge s\ s'\ v0\ e.$

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

**and** *finish'*: $\bigwedge s\ s'\ u\ e.$

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

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

$[\![I\ s;\ cond\ s;\ DFS\text{-}invar\ G\ param\ s;\ DFS\text{-}invar\ G\ param\ s';$
$stack\ s \neq [];\ v \in pending\ s\ ``\ \{u\};\ v \in dom\ (discovered\ s);$
$u = hd\ (stack\ s);$
$cross\text{-}edges\ s \subseteq c;\ back\text{-}edges\ s \subseteq b;$
$s' = s(\!|$
$pending := pending\ s - \{(u,v)\},$
$state.more := e,$
$cross\text{-}edges := c,$
$back\text{-}edges := b|\!)]\!]$
$\Longrightarrow I\ s'$

**and** *discover'*: $\bigwedge s\ s'\ u\ v\ e.$

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

**shows** *is-invar I*
⟨*proof*⟩

**end**

### 1.1.5 Basic Invariants

We establish some basic invariants

**context** *param-DFS* **begin**

**definition** *basic-invar s* $\equiv$
$set\ (stack\ s) = dom\ (discovered\ s) - dom\ (finished\ s) \wedge$
$distinct\ (stack\ s) \wedge$
$(stack\ s \neq [] \longrightarrow last\ (stack\ s) \in V0) \wedge$
$dom\ (finished\ s) \subseteq dom\ (discovered\ s) \wedge$
$Domain\ (pending\ s) \subseteq dom\ (discovered\ s) - dom\ (finished\ s) \wedge$
$pending\ s \subseteq E$

**lemma** *i-basic-invar*: *is-invar basic-invar*
⟨*proof*⟩
**end**

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

**lemma** *pending-ssE*: $pending\ s \subseteq E$
⟨*proof*⟩

**lemma** *pendingD*:
$(u,v) \in pending\ s \Longrightarrow (u,v) \in E \wedge u \in dom\ (discovered\ s)$
⟨*proof*⟩

**lemma** *stack-set-def*:
$set\ (stack\ s) = dom\ (discovered\ s) - dom\ (finished\ s)$
⟨*proof*⟩

**lemma** *stack-discovered*:
$set\ (stack\ s) \subseteq dom\ (discovered\ s)$
⟨*proof*⟩

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

**lemma** *last-stack-in-V0*:
$stack\ s \neq [] \Longrightarrow last\ (stack\ s) \in V0$
⟨*proof*⟩

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

**lemma** *discovered-not-stack-imp-finished*:
$x \in dom\ (discovered\ s) \Longrightarrow x \notin set\ (stack\ s) \Longrightarrow x \in dom\ (finished\ s)$
⟨*proof*⟩

**lemma** *finished-discovered*:

$dom\ (finished\ s) \subseteq dom\ (discovered\ s)$
⟨*proof*⟩

**lemma** *finished-no-pending*:
$v \in dom\ (finished\ s) \Longrightarrow pending\ s\ ``\ \{v\} = \{\}$
⟨*proof*⟩

**lemma** *discovered-eq-finished-un-stack*:
$dom\ (discovered\ s) = dom\ (finished\ s) \cup set\ (stack\ s)$
⟨*proof*⟩

**lemma** *pending-on-stack*:
$(v,w) \in pending\ s \Longrightarrow v \in set\ (stack\ s)$
⟨*proof*⟩

**lemma** *empty-stack-imp-empty-pending*:
$stack\ s = [] \Longrightarrow pending\ s = \{\}$
⟨*proof*⟩
**end**

**context** *param-DFS* **begin**

**lemma** *i-discovered-reachable*:
$is\text{-}invar\ (\lambda s.\ dom\ (discovered\ s) \subseteq reachable)$
⟨*proof*⟩

**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*⟩

**lemma** *i-discovered-finite*: $is\text{-}invar\ (\lambda s.\ finite\ (dom\ (discovered\ s)))$
⟨*proof*⟩

**end**

**context** *DFS-invar*
**begin**

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

**lemma** *stack-reachable*: $set\ (stack\ s) \subseteq reachable$
⟨*proof*⟩

**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*))
⟨*proof*⟩

**lemma** *finished-closed*:
*E* `` *dom* (*finished s*) ⊆ *dom* (*discovered s*)
⟨*proof*⟩

**lemma** *finished-imp-succ-discovered*:
$v \in$ *dom* (*finished s*) $\Longrightarrow$ $w \in$ *succ v* $\Longrightarrow$ $w \in$ *dom* (*discovered s*)
⟨*proof*⟩

**lemma** *pending-reachable*: *pending s* ⊆ *reachable* × *reachable*
⟨*proof*⟩

**lemma** *pending-finite*[*simp*, *intro*!]: *finite* (*pending s*)
⟨*proof*⟩

**lemma** *no-pending-imp-succ-discovered*:
**assumes** $u \in$ *dom* (*discovered s*)
**and** *pending s* `` {*u*} = {}
**and** $v \in$ *succ u*
**shows** $v \in$ *dom* (*discovered s*)
⟨*proof*⟩

**lemma** *nc-finished-eq-reachable*:
**assumes** *NC*: ¬*cond s* ¬*is-break param s*
**shows** *dom* (*finished s*) = *reachable*
⟨*proof*⟩

**lemma** *nc-V0-finished*:
**assumes** *NC*: ¬ *cond s* ¬ *is-break param s*
**shows** *V0* ⊆ *dom* (*finished s*)
⟨*proof*⟩

**lemma** *nc-discovered-eq-finished*:
**assumes** *NC*: ¬ *cond s* ¬ *is-break param s*
**shows** *dom* (*discovered s*) = *dom* (*finished s*)
⟨*proof*⟩

**lemma** *nc-discovered-eq-reachable*:
**assumes** *NC*: ¬ *cond s* ¬ *is-break param s*
**shows** *dom* (*discovered s*) = *reachable*
⟨*proof*⟩

**lemma** *nc-fin-closed*:

**assumes** *NC*: ¬*cond s*
**assumes** *NB*: ¬*is-break param s*
**shows** *E*‘‘*dom* (*finished s*) ⊆ *dom* (*finished s*)
⟨*proof*⟩

**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*
⟨*proof*⟩

**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*⟩

**end**

**context** *param-DFS* **begin**
**lemma** *it-dfsT-eq-it-dfs*:
**assumes** [*simp*, *intro*!]: *finite reachable*
**shows** *it-dfsT* = *it-dfs*
⟨*proof*⟩
**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* $\Longrightarrow$ *nofail* (*on-new-root param v0 s*)

**assumes** *nofail-on-finish*:
*pre-on-finish u s* $\Longrightarrow$ *nofail* (*on-finish param u s*)

**assumes** *nofail-on-cross-edge*:
*pre-on-cross-edge u v s* $\Longrightarrow$ *nofail* (*on-cross-edge param u v s*)

**assumes** *nofail-on-back-edge*:
*pre-on-back-edge u v s* $\Longrightarrow$ *nofail* (*on-back-edge param u v s*)

**assumes** *nofail-on-discover*:
*pre-on-discover u v s* $\Longrightarrow$ *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*)
⟨*proof*⟩

**lemma** *it-dfs-eq-spec*: *it-dfs* = *SPEC* ($\lambda$*s. DFS-invar G param s* $\wedge$ $\neg$*cond s*)
⟨*proof*⟩

**lemma** *it-dfs-correct*: *it-dfs* $\leq$ *SPEC* ($\lambda$*s. DFS-invar G param s* $\wedge$ $\neg$*cond s*)
⟨*proof*⟩

**lemma** *it-dfs-SPEC*:
**assumes** $\bigwedge$*s*. ⟦*DFS-invar G param s*; $\neg$*cond s*⟧ $\Longrightarrow$ *P s*
**shows** *it-dfs* $\leq$ *SPEC P*
⟨*proof*⟩

**lemma** *it-dfsT-correct*:
**assumes** *finite reachable*
**shows** *it-dfsT* $\leq$ *SPEC* ($\lambda$*s. DFS-invar G param s* $\wedge$ $\neg$*cond s*)
⟨*proof*⟩

**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*
⟨*proof*⟩

**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* ($\delta$)
  **notation** *the-finished* ($\varphi$)
**end**

**context** *param-DFS* **begin context begin interpretation** *timing-syntax* ⟨*proof*⟩

  **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*⟩
**end end**

**context** *DFS-invar* **begin context begin interpretation** *timing-syntax* ⟨*proof*⟩

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

**lemma** *timing-less-counter*:
$v \in dom\ (discovered\ s) \Longrightarrow \delta\ s\ v < counter\ s$
$v \in dom\ (finished\ s) \Longrightarrow \varphi\ s\ v < counter\ s$
⟨*proof*⟩

**lemma** *disc-lt-fin*:
$v \in dom\ (finished\ s) \Longrightarrow \delta\ s\ v < \varphi\ s\ v$
⟨*proof*⟩

**lemma** *disc-unequal*:
**assumes** $v \in dom\ (discovered\ s)\ w \in dom\ (discovered\ s)$
**and** $v \neq w$
**shows** $\delta\ s\ v \neq \delta\ s\ w$
⟨*proof*⟩

**lemma** *fin-unequal*:
**assumes** $v \in dom\ (finished\ s)\ w \in dom\ (finished\ s)$
**and** $v \neq w$
**shows** $\varphi\ s\ v \neq \varphi\ s\ w$
⟨*proof*⟩

**lemma** *finished-succ-fin*:
**assumes** $v \in dom\ (finished\ s)$
**and** $w \in succ\ v$
**shows** $\delta\ s\ w < \varphi\ s\ v$
⟨*proof*⟩
**end end**

**context** *param-DFS* **begin context begin interpretation** *timing-syntax* ⟨*proof*⟩

**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*⟩
**end end**

**context** *DFS-invar* **begin context begin interpretation** *timing-syntax* ⟨*proof*⟩

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

**lemma** *prev-stack-discover*:
$[\![ n < length\ (stack\ s);\ v \in set\ (drop\ (Suc\ n)\ (stack\ s))\ ]\!]$
$\Longrightarrow \delta\ s\ (stack\ s\ !\ n) > \delta\ s\ v$
⟨*proof*⟩

**lemma** *Suc-stack-discover*:
**assumes** $n$: $n < (length\ (stack\ s)) - 1$
**shows** $\delta\ s\ (stack\ s\ !\ n) > \delta\ s\ (stack\ s\ !\ Suc\ n)$

⟨*proof*⟩

**lemma** *tl-lt-stack-hd-discover*:
**assumes** *notempty*: $stack\ s \neq []$
**and** $x \in set\ (tl\ (stack\ s))$
**shows** $\delta\ s\ x < \delta\ s\ (hd\ (stack\ s))$
⟨*proof*⟩

**lemma** *stack-nth-order*:
**assumes** *l*: $i < length\ (stack\ s)\ j < length\ (stack\ s)$
**shows** $\delta\ s\ (stack\ s\ !\ i) < \delta\ s\ (stack\ s\ !\ j) \longleftrightarrow i > j$ (**is** $\delta\ s\ ?i < \delta\ s\ ?j \longleftrightarrow$ -)
⟨*proof*⟩
**end end**

### 1.2.2 Paranthesis Theorem

**context** *param-DFS* **begin context begin interpretation** *timing-syntax* ⟨*proof*⟩

**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*⟩
**end end**

**context** *DFS-invar* **begin context begin interpretation** *timing-syntax* ⟨*proof*⟩

**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
$\vee\ (\varphi\ s\ v > \delta\ s\ w \wedge w \in dom\ (finished\ s) \wedge \varphi\ s\ w < \varphi\ s\ v)$
⟨*proof*⟩

**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** $w \in dom\ (finished\ s) \wedge \varphi\ s\ w < \varphi\ s\ v$
⟨*proof*⟩

**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** $\varphi\ s\ v < \delta\ s\ w$
⟨*proof*⟩

**lemma** *finished-succ-contained*:
**assumes** $v \in dom\ (finished\ s)$
**and** $w \in succ\ v$
**and** $\delta\ s\ v < \delta\ s\ w$
**shows** $w \in dom\ (finished\ s) \wedge \varphi\ s\ w < \varphi\ s\ v$
$\langle proof \rangle$

**end end**

### 1.2.3 Edge Types

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

**lemma** $is\text{-}invar\ (\lambda s.\ finite\ (edges\ s))$
$\langle proof \rangle$

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

**lemma** *edgesE-CB*:
**assumes** $x \in edges\ s$
**and** $x \in tree\text{-}edges\ s \Longrightarrow P$
**and** $x \in cross\text{-}edges\ s \cup back\text{-}edges\ s \Longrightarrow P$
**shows** $P$
$\langle proof \rangle$

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

**lemma** *i-edges-basic*:
$is\text{-}invar\ edges\text{-}basic$
$\langle proof \rangle$

**lemmas** (**in** *DFS-invar*) $edges\text{-}basic = i\text{-}edges\text{-}basic[THEN\ make\text{-}invar\text{-}thm]$

**lemma** *i-edges-covered*:
$is\text{-}invar\ (\lambda s.\ (E \cap dom\ (discovered\ s) \times UNIV) - pending\ s = edges\ s)$
$\langle proof \rangle$
**end**

**context** *DFS-invar* **begin**

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

**lemma** *edges-ss-reachable-edges*:
$edges\ s \subseteq E \cap reachable \times UNIV$
⟨*proof*⟩

**lemma** *nc-edges-covered*:
**assumes** $\neg cond\ s$ $\neg is\text{-}break\ param\ s$
**shows** $E \cap reachable \times UNIV = edges\ s$
⟨*proof*⟩

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

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

*back-edges-ssE*: $back\text{-}edges\ s \subseteq E$ **and**
*back-edges-not-pending*: $back\text{-}edges\ s \subseteq -\ pending\ s$ **and**
*back-edge-is-succ*: $(v,w) \in back\text{-}edges\ s \Longrightarrow w \in succ\ v$ **and**
*back-edges-discovered*: $Field\ (back\text{-}edges\ s) \subseteq dom\ (discovered\ s)$
⟨*proof*⟩

**lemma** *edges-disjoint*:
$back\text{-}edges\ s \cap cross\text{-}edges\ s = \{\}$
$back\text{-}edges\ s \cap tree\text{-}edges\ s = \{\}$
$cross\text{-}edges\ s \cap tree\text{-}edges\ s = \{\}$
⟨*proof*⟩

**lemma** *tree-edge-imp-discovered*:
$(v,w) \in tree\text{-}edges\ s \Longrightarrow v \in dom\ (discovered\ s)$
$(v,w) \in tree\text{-}edges\ s \Longrightarrow w \in dom\ (discovered\ s)$
⟨*proof*⟩

**lemma** *back-edge-imp-discovered*:
$(v,w) \in back\text{-}edges\ s \Longrightarrow v \in dom\ (discovered\ s)$
$(v,w) \in back\text{-}edges\ s \Longrightarrow w \in dom\ (discovered\ s)$
⟨*proof*⟩

**lemma** *cross-edge-imp-discovered*:
$(v,w) \in cross\text{-}edges\ s \Longrightarrow v \in dom\ (discovered\ s)$
$(v,w) \in cross\text{-}edges\ s \Longrightarrow w \in dom\ (discovered\ s)$
⟨*proof*⟩

**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*)
  ⟨*proof*⟩

**lemma** *tree-edges-finite*[*simp*, *intro*!]: *finite* (*tree-edges s*)
  ⟨*proof*⟩

**lemma** *cross-edges-finite*[*simp*, *intro*!]: *finite* (*cross-edges s*)
  ⟨*proof*⟩

**lemma** *back-edges-finite*[*simp*, *intro*!]: *finite* (*back-edges s*)
  ⟨*proof*⟩

**lemma** *edges-finite*: *finite* (*edges s*)
  ⟨*proof*⟩

**end**

### Properties of the DFS Tree

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

**lemma** *back-edge-disc-lt-fin*:
  $(v,w) \in$ *back-edges s* $\Longrightarrow v \in$ *dom* (*finished s*) $\Longrightarrow \delta\ s\ w < \varphi\ s\ v$
  ⟨*proof*⟩

**lemma** *cross-edge-disc-lt-fin*:
  $(v,w) \in$ *cross-edges s* $\Longrightarrow v \in$ *dom* (*finished s*) $\Longrightarrow \delta\ s\ w < \varphi\ s\ v$
  ⟨*proof*⟩
**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*⟩
**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*:
$stack\ s \neq [] \Longrightarrow \exists\, v0 \in V0.\ path\ E\ v0\ (rev\ (tl\ (stack\ s)))\ (hd\ (stack\ s))$
⟨*proof*⟩

**lemma** *hd-succ-stack-is-path*:
**assumes** *ne*: $stack\ s \neq []$
**and** *succ*: $v \in succ\ (hd\ (stack\ s))$
**shows** $\exists\, v0 \in V0.\ path\ E\ v0\ (rev\ (stack\ s))\ v$
⟨*proof*⟩

**lemma** *tl-stack-hd-tree-path*:
**assumes** $stack\ s \neq []$
**and** $v \in set\ (tl\ (stack\ s))$
**shows** $(v,\ hd\ (stack\ s)) \in (tree\text{-}edges\ s)^+$
⟨*proof*⟩
**end**

**context** *param-DFS* **begin**
**definition** *tree-discovered-inv s* ≡
$(tree\text{-}edges\ s = \{\} \longrightarrow dom\ (discovered\ s) \subseteq V0 \wedge (stack\ s = []$
$\vee (\exists\, v0 \in V0.\ stack\ s = [v0])))$
$\wedge (tree\text{-}edges\ s \neq \{\} \longrightarrow (tree\text{-}edges\ s)^+ \text{ `` } V0 \cup V0 = dom$
$(discovered\ s) \cup V0)$

**lemma** *i-tree-discovered-inv*:
*is-invar tree-discovered-inv*
⟨*proof*⟩

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

**lemma** (**in** *DFS-invar*) *discovered-iff-tree-path*:
$v \notin V0 \Longrightarrow v \in dom\ (discovered\ s) \longleftrightarrow (\exists\, v0 \in V0.\ (v0,v) \in (tree\text{-}edges\ s)^+)$
⟨*proof*⟩

**lemma** *i-tree-one-predecessor*:
$is\text{-}invar\ (\lambda s.\ \forall\, (v,v') \in tree\text{-}edges\ s.\ \forall\, y.\ y \neq v \longrightarrow (y,v') \notin tree\text{-}edges\ s)$
⟨*proof*⟩

**lemma** (**in** *DFS-invar*) *tree-one-predecessor*:
**assumes** $(v,w) \in tree\text{-}edges\ s$
**and** $a \neq v$
**shows** $(a,w) \notin tree\text{-}edges\ s$
⟨*proof*⟩

**lemma** (**in** *DFS-invar*) *tree-eq-rule*:

$[\![(v,w) \in \textit{tree-edges } s;\ (u,w) \in \textit{tree-edges } s]\!] \Longrightarrow v=u$
⟨*proof*⟩

**context begin interpretation** *timing-syntax* ⟨*proof*⟩

**lemma** *i-tree-edge-disc*:
*is-invar* $(\lambda s.\ \forall (v,v') \in \textit{tree-edges } s.\ \delta\ s\ v < \delta\ s\ v')$
⟨*proof*⟩
**end end**

**context** *DFS-invar* **begin context begin interpretation** *timing-syntax* ⟨*proof*⟩

**lemma** *tree-edge-disc*:
$(v,w) \in \textit{tree-edges } s \Longrightarrow \delta\ s\ v < \delta\ s\ w$
⟨*proof*⟩

**lemma** *tree-path-disc*:
$(v,w) \in (\textit{tree-edges } s)^+ \Longrightarrow \delta\ s\ v < \delta\ s\ w$
⟨*proof*⟩

**lemma** *no-loop-in-tree*:
$(v,v) \notin (\textit{tree-edges } s)^+$
⟨*proof*⟩

**lemma** *tree-acyclic*:
*acyclic* (*tree-edges s*)
⟨*proof*⟩

**lemma** *no-self-loop-in-tree*:
$(v,v) \notin \textit{tree-edges } s$
⟨*proof*⟩

**lemma** *tree-edge-unequal*:
$(v,w) \in \textit{tree-edges } s \Longrightarrow v \neq w$
⟨*proof*⟩

**lemma** *tree-path-unequal*:
$(v,w) \in (\textit{tree-edges } s)^+ \Longrightarrow v \neq w$
⟨*proof*⟩

**lemma** *tree-subpath'*:
**assumes** *x*: $(x,v) \in (\textit{tree-edges } s)^+$
**and** *y*: $(y,v) \in (\textit{tree-edges } s)^+$
**and** $x \neq y$
**shows** $(x,y) \in (\textit{tree-edges } s)^+ \lor (y,x) \in (\textit{tree-edges } s)^+$
⟨*proof*⟩

**lemma** *tree-subpath*:
**assumes** $(x,v) \in (\textit{tree-edges } s)^+$

**and** $(y,v) \in (\textit{tree-edges } s)^+$
**and** $\delta$: $\delta\ s\ x < \delta\ s\ y$
**shows** $(x,y) \in (\textit{tree-edges } s)^+$
⟨*proof*⟩

**lemma** *on-stack-is-tree-path*:
**assumes** $x$: $x \in \textit{set } (\textit{stack } s)$
**and** $y$: $y \in \textit{set } (\textit{stack } s)$
**and** $\delta$: $\delta\ s\ x < \delta\ s\ y$
**shows** $(x,y) \in (\textit{tree-edges } s)^+$
⟨*proof*⟩

**lemma** *hd-stack-tree-path-finished*:
**assumes** $\textit{stack } s \neq []$
**assumes** $(\textit{hd } (\textit{stack } s),\ v) \in (\textit{tree-edges } s)^+$
**shows** $v \in \textit{dom } (\textit{finished } s)$
⟨*proof*⟩

**lemma** *tree-edge-impl-parenthesis*:
**assumes** $t$: $(v,w) \in \textit{tree-edges } s$
**and** $f$: $v \in \textit{dom } (\textit{finished } s)$
**shows** $w \in \textit{dom } (\textit{finished } s)$
$\wedge\ \delta\ s\ v < \delta\ s\ w$
$\wedge\ \varphi\ s\ w < \varphi\ s\ v$
⟨*proof*⟩

**lemma** *tree-path-impl-parenthesis*:
**assumes** $(v,w) \in (\textit{tree-edges } s)^+$
**and** $v \in \textit{dom } (\textit{finished } s)$
**shows** $w \in \textit{dom } (\textit{finished } s)$
$\wedge\ \delta\ s\ v < \delta\ s\ w$
$\wedge\ \varphi\ s\ w < \varphi\ s\ v$
⟨*proof*⟩

**lemma** *nc-reachable-v0-parenthesis*:
**assumes** $C$: $\neg\ \textit{cond } s\ \neg\ \textit{is-break param } s$
**and** $v$: $v \in \textit{reachable } v \notin \textit{V0}$
**obtains** $\textit{v0}$ **where** $\textit{v0} \in \textit{V0}$
**and** $\delta\ s\ \textit{v0} < \delta\ s\ v \wedge \varphi\ s\ v < \varphi\ s\ \textit{v0}$
⟨*proof*⟩

**end end**

**context** *param-DFS* **begin context begin interpretation** *timing-syntax* ⟨*proof*⟩

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

**lemma** *paren-imp-tree-reach*:
*is-invar paren-imp-tree-reach*
⟨*proof*⟩
**end end**

**context** *DFS-invar* **begin context begin interpretation** *timing-syntax* ⟨*proof*⟩

**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** $(v,w) \in (tree\text{-}edges\ s)^+$
⟨*proof*⟩

**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*⟩

**lemma** *tree-path-iff-parenthesis*:
**assumes** $v \in dom\ (finished\ s)\ w \in dom\ (finished\ s)$
**shows** $(v,w) \in (tree\text{-}edges\ s)^+ \longleftrightarrow \delta\ s\ v < \delta\ s\ w \wedge \varphi\ s\ v > \varphi\ s\ w$
⟨*proof*⟩

**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** $\delta$: $\delta\ s\ v < \delta\ s\ w$
**shows** $(v,w) \in (tree\text{-}edges\ s)^+$
⟨*proof*⟩

**lemma** *finished-succ-impl-path-in-tree*:
**assumes** $f$: $v \in dom\ (finished\ s)$
**and** $s$: $w \in succ\ v$
**and** $\delta$: $\delta\ s\ v < \delta\ s\ w$
**shows** $(v,w) \in (tree\text{-}edges\ s)^+$
⟨*proof*⟩
**end end**

## Properties of Cross Edges

**context** *param-DFS* **begin context begin interpretation** *timing-syntax* ⟨*proof*⟩

**lemma** *i-cross-edges-finished*: *is-invar* ($\lambda s.$ $\forall$ $(u,v)\in$*cross-edges s*.
$v \in$ *dom* (*finished s*) $\wedge$ ($u \in$ *dom* (*finished s*) $\longrightarrow$ $\varphi$ $s$ $v$ $<$ $\varphi$ $s$ $u$))
⟨*proof*⟩

**end end**

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

**lemma** *cross-edges-target-finished*:
$(u,v)\in$*cross-edges s* $\Longrightarrow$ $v \in$ *dom* (*finished s*)
⟨*proof*⟩

**lemma** *cross-edges-finished-decr*:
$[\![(u,v)\in$*cross-edges s*; $u\in$*dom* (*finished s*)$]\!]$ $\Longrightarrow$ $\varphi$ $s$ $v$ $<$ $\varphi$ $s$ $u$
⟨*proof*⟩

**lemma** *cross-edge-unequal*:
**assumes** *cross*: $(v,w) \in$ *cross-edges s*
**shows** $v \neq w$
⟨*proof*⟩
**end end**

### Properties of Back Edges

**context** *param-DFS* **begin context begin interpretation** *timing-syntax* ⟨*proof*⟩

**lemma** *i-back-edge-impl-tree-path*:
*is-invar* ($\lambda s.$ $\forall$ $(v,w) \in$ *back-edges s*. $(w,v) \in$ (*tree-edges s*)$^+$ $\vee$ $w = v$)
⟨*proof*⟩

**end end**

**context** *DFS-invar* **begin context begin interpretation** *timing-syntax* ⟨*proof*⟩

**lemma** *back-edge-impl-tree-path*:
$[\![(v,w) \in$ *back-edges s*; $v \neq w]\!]$ $\Longrightarrow$ $(w,v) \in$ (*tree-edges s*)$^+$
⟨*proof*⟩

**lemma** *back-edge-disc*:
**assumes** $(v,w) \in$ *back-edges s*
**shows** $\delta$ $s$ $w \leq \delta$ $s$ $v$
⟨*proof*⟩

**lemma** *back-edges-tree-disjoint*:
*back-edges s* $\cap$ *tree-edges s* = {}
⟨*proof*⟩

**lemma** *back-edges-tree-pathes-disjoint*:
  *back-edges s* $\cap$ (*tree-edges s*)$^+$ = {}
  ⟨*proof*⟩

**lemma** *back-edge-finished*:
  **assumes** ($v$,$w$) $\in$ *back-edges s*
  **and** $w \in$ *dom* (*finished s*)
  **shows** $v \in$ *dom* (*finished s*) $\wedge$ $\varphi$ $s$ $v \leq \varphi$ $s$ $w$
⟨*proof*⟩

**end end**

**context** *param-DFS* **begin context begin interpretation** *timing-syntax* ⟨*proof*⟩

**lemma** *i-disc-imp-back-edge-or-pending*:
  *is-invar* ($\lambda s$. $\forall$ ($v$,$w$) $\in E$.
      $v \in$ *dom* (*discovered s*) $\wedge$ $w \in$ *dom* (*discovered s*)
    $\wedge$ $\delta$ $s$ $v \geq \delta$ $s$ $w$
    $\wedge$ ($w \in$ *dom* (*finished s*) $\longrightarrow$ $v \in$ *dom* (*finished s*) $\wedge$ $\varphi$ $s$ $w \geq \varphi$ $s$ $v$)
    $\longrightarrow$ ($v$,$w$) $\in$ *back-edges s* $\vee$ ($v$,$w$) $\in$ *pending s*)
⟨*proof*⟩
**end end**

**context** *DFS-invar* **begin context begin interpretation** *timing-syntax* ⟨*proof*⟩

**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-edges s* $\vee$ ($v$, $w$) $\in$ *pending s*
  ⟨*proof*⟩

**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-edges s*
  ⟨*proof*⟩

**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-edges s*
  ⟨*proof*⟩

**lemma** *finished-self-loop-in-back-edges*:
  **assumes** $v \in$ *dom* (*finished s*)
  **and** ($v$,$v$) $\in E$
  **shows** ($v$,$v$) $\in$ *back-edges s*
  ⟨*proof*⟩

**end** **end**

**context** *DFS-invar* **begin**

  **context begin interpretation** *timing-syntax* ⟨*proof*⟩

    **lemma** *tree-cross-acyclic*:
      *acyclic* (*tree-edges s* ∪ *cross-edges s*) (**is** *acyclic ?E*)
    ⟨*proof*⟩
  **end**

  **lemma** *cycle-contains-back-edge*:
    **assumes** *cycle*: $(u,u) \in (edges\ s)^+$
    **shows** $\exists\, v\ w.\ (u,v) \in (edges\ s)^* \wedge (v,w) \in back\text{-}edges\ s \wedge (w,u) \in (edges\ s)^*$
  ⟨*proof*⟩

  **lemma** *cycle-needs-back-edge*:
    **assumes** *back-edges s* = {}
    **shows** *acyclic* (*edges s*)
  ⟨*proof*⟩

  **lemma** *back-edge-closes-cycle*:
    **assumes** *back-edges s* ≠ {}
    **shows** ¬ *acyclic* (*edges s*)
  ⟨*proof*⟩

  **lemma** *back-edge-closes-reachable-cycle*:
    *back-edges s* ≠ {} ⟹ ¬ *acyclic* (*E* ∩ *reachable* × *UNIV*)
    ⟨*proof*⟩

  **lemma** *cycle-iff-back-edges*:
    *acyclic* (*edges s*) ⟷ *back-edges s* = {}
  ⟨*proof*⟩
**end**

### 1.2.4 White Path Theorem

**context** *DFS* **begin**
**context begin interpretation** *timing-syntax* ⟨*proof*⟩

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

  **lemma** *white-path*:
    $it\text{-}dfs \leq SPEC(\lambda s.\ \forall\, x \in reachable.\ \forall\, y \in reachable.\ \neg\ is\text{-}break\ param\ s \longrightarrow$

*white-path s x y* $\longleftrightarrow$ $(x,y) \in (\textit{tree-edges } s)^*$)
⟨*proof*⟩
**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-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-edges s*)$^*$ `` $\{v\}$
**shows** *scc-root s v scc*
⟨*proof*⟩

**definition** *scc-roots s* = $\{v.\ \exists$ *scc. scc-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*)
⟨*proof*⟩

**lemma** *scc-root-scc-tree-rtrancl*:
**assumes** *scc-root s v scc*
**and** $x \in$ *scc* $x \in$ *dom* (*discovered s*)
**shows** $(v,x) \in$ (*tree-edges s*)$^*$
⟨*proof*⟩

**lemma** *scc-root-scc-reach*:
  **assumes** *scc-root s r scc*
  **and** $v \in scc$
  **shows** $(r,v) \in E^*$
⟨*proof*⟩

**lemma** *scc-reach-scc-root*:
  **assumes** *scc-root s r scc*
  **and** $v \in scc$
  **shows** $(v,r) \in E^*$
⟨*proof*⟩

**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** $(v,x) \in$ (*tree-edges s*)$^+$
  ⟨*proof*⟩

**lemma** *scc-root-unique-scc*:
  *scc-root s v scc* $\Longrightarrow$ *scc-root s v scc'* $\Longrightarrow$ $scc = scc'$
  ⟨*proof*⟩

**lemma** *scc-root-unique-root*:
  **assumes** *scc1*: *scc-root s v scc*
  **and** *scc2*: *scc-root s v' scc*
  **shows** $v = v'$
⟨*proof*⟩

**lemma** *scc-root-unique-is-scc*:
  **assumes** *scc-root s v scc*
  **shows** *scc-root s v* (*scc-of E v*)
⟨*proof*⟩

**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*⟩

**context begin interpretation** *timing-syntax* ⟨*proof*⟩
**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*⟩

**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*⟩

**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*⟩

**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*⟩

**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* $\longleftrightarrow Min\ (\delta\ s\ `\ (scc \cap dom\ (discovered\ s))) = \delta\ s\ r$ (**is** $?L \longleftrightarrow ?R$)
⟨*proof*⟩

**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*⟩

**lemma** *scc-root-of-node-exists*:
**assumes** $v \in dom\ (discovered\ s)$
**shows** $\exists\, r.$ *scc-root s r (scc-of E v)*
⟨*proof*⟩

**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*
⟨*proof*⟩

**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-edges s* $\subseteq$ *tree-edges s'*
**shows** *scc-root s r scc* $\longleftrightarrow$ *scc-root s' r scc*
⟨*proof*⟩

**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 (λv0. 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*
∧ *inres* (*gds-get-pending gds s0*) (*u*, *Vs*,*s*)

**definition** *pre-finish u s0 s* ≡ *post-get-pending u None s0 s*
**definition** *pre-cross-edge u v s0 s* ≡
*post-get-pending u* (*Some v*) *s0 s* ∧ *gds-is-discovered gds v s*
∧ *gds-is-finished gds v s*
**definition** *pre-back-edge u v s0 s* ≡
*post-get-pending u* (*Some v*) *s0 s* ∧ *gds-is-discovered gds v s*
∧ ¬*gds-is-finished gds v s*
**definition** *pre-discover u v s0 s* ≡
*post-get-pending u* (*Some v*) *s0 s* ∧ ¬*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* ≡
  *if gds-is-empty-stack gds s then do* {
    *v0* ← *SPEC* ($\lambda$*v0*. *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*
  ≡ *gds-init gds* ≫= *WHILE gen-cond gen-step-assert*

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

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

**lemma** *gen-step-eq-assert*: ⟦*gen-cond s*; *gen-rwof s*⟧
    ⟹ *gen-step s* = *gen-step-assert s*
  ⟨*proof*⟩

**lemma** *gen-dfs-eq-assert*: *gen-dfs* = *gen-dfs-assert*
  ⟨*proof*⟩

**lemma** *gen-dfsT-eq-assert*: *gen-dfsT* = *gen-dfsT-assert*
  ⟨*proof*⟩

**lemma** *gen-rwof-eq-assert*:
  **assumes** *NF*: *nofail gen-dfs*

**shows** *gen-rwof* = *gen-rwof-assert*
⟨*proof*⟩

**lemma** *gen-dfs-le-gen-dfsT*: *gen-dfs* ≤ *gen-dfsT*
⟨*proof*⟩

**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* ⇒ *'s nres*
*gbs-is-empty-stack* :: *'s* ⇒ *bool*
*gbs-new-root* :: *'v* ⇒ *'s* ⇒ *'s nres*
*gbs-get-pending* :: *'s* ⇒ (*'v* × *'v option* × *'s*) *nres*
*gbs-finish* :: *'v* ⇒ *'s* ⇒ *'s nres*
*gbs-is-discovered* :: *'v* ⇒ *'s* ⇒ *bool*
*gbs-is-finished* :: *'v* ⇒ *'s* ⇒ *bool*
*gbs-back-edge* :: *'v* ⇒ *'v* ⇒ *'s* ⇒ *'s nres*
*gbs-cross-edge* :: *'v* ⇒ *'v* ⇒ *'s* ⇒ *'s nres*
*gbs-discover* :: *'v* ⇒ *'v* ⇒ *'s* ⇒ *'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*⇒*'es*) ⇒ *'s* ⇒ *'s*
**fixes** *V0* :: *'v set*
**begin**

**definition** *do-action bf ef s* ≡ *do* {
*s* ← *bf s*;
*e* ← *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* ⟨*proof*⟩
**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* ⟨*proof*⟩

**lemma** *gen-cond-simp*[*simp*]: *gen-dfs.gen-cond* = *cond*
⟨*proof*⟩

**lemma** *gen-step-simp*[*simp*]: *gen-dfs.gen-step* = *step*
⟨*proof*⟩

**lemma** *gen-init-simp*[*simp*]: *gen-dfs.do-init* = *init*
⟨*proof*⟩

**lemma** *gen-dfs-simp*[*simp*]: *gen-dfs.gen-dfs* = *it-dfs*
⟨*proof*⟩

**lemma** *gen-dfsT-simp*[*simp*]: *gen-dfs.gen-dfsT* = *it-dfsT*
⟨*proof*⟩

**end**

**context** *param-DFS* **begin**
**sublocale** *gen-dfs*: *gen-param-dfs gbs param state.more-update V0* ⟨*proof*⟩
**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*⟧
⟹ *gds-discover gdsi ui vi si* ≤ ⇓ *S* (*gds-discover gds u v s*)

**begin**
**term** *gds-is-discovered gdsi*

**lemma** *select-v0-refine*[*refine*]:
**assumes** *s-param*: (*si*,*s*)∈*S*
**shows** *SPEC* (λ*v0*. *v0* ∈ *V0i* ∧ ¬ *gds-is-discovered gdsi v0 si*)
≤ ⇓ *V* (*SPEC* (λ*v0*. *v0* ∈ *V0* ∧ ¬ *gds-is-discovered gds v0 s*))
⟨*proof*⟩

**lemma** *gen-rwof-refine*:
**assumes** *NF*: *nofail* (*a.gen-dfs*)
**assumes** *RW*: *c.gen-rwof s*
**obtains** *s′* **where** (*s*,*s′*)∈*S* **and** *a.gen-rwof s′*
⟨*proof*⟩

**lemma** *gen-step-refine*[*refine*]: (*si*,*s*)∈*S* ⟹ *c.gen-step si* ≤ ⇓*S* (*a.gen-step-assert s*)
⟨*proof*⟩

**lemma** *gen-dfs-refine*[*refine*]: *c.gen-dfs* $\leq \Downarrow S$ *a.gen-dfs*
⟨*proof*⟩

**lemma** *gen-dfsT-refine*[*refine*]: *c.gen-dfsT* $\leq \Downarrow S$ *a.gen-dfsT*
⟨*proof*⟩

**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*)$\in S \rightarrow$*bool-rel*

**assumes** *gbs-init-refine*[*refine*]: $(ei, e) \in ES \Longrightarrow$ *gbs-init gbsi ei* $\leq \Downarrow S$ (*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* $\leq \Downarrow S$ (*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*
$\leq \Downarrow (V \times_r \langle V \rangle option\text{-}rel \times_r S)$ (*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* $\leq \Downarrow S$ (*gbs-finish gbs v s*)

**assumes** *gbs-cross-edge-refine*[*refine*]:

⟦*a.pre-cross-edge u v s0 s*; (*ui*, *u*) ∈ *V*; (*vi*, *v*) ∈ *V*; (*si*, *s*) ∈ *S*⟧
⟹ *gbs-cross-edge gbsi ui vi si* ≤ ⇓ *S* (*gbs-cross-edge gbs u v s*)

**assumes** *gbs-back-edge-refine*[*refine*]:
⟦*a.pre-back-edge u v s0 s*; (*ui*, *u*) ∈ *V*; (*vi*, *v*) ∈ *V*; (*si*, *s*) ∈ *S*⟧
⟹ *gbs-back-edge gbsi ui vi si* ≤ ⇓ *S* (*gbs-back-edge gbs u v s*)

**assumes** *gbs-discover-refine*[*refine*]:
⟦*a.pre-discover u v s0 s*; (*ui*, *u*) ∈ *V*; (*vi*, *v*) ∈ *V*; (*si*, *s*) ∈ *S*⟧
⟹ *gbs-discover gbsi ui vi si* ≤ ⇓ *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* → *ES*) → *S* → *S*

**assumes** *on-init-refine*[*refine*]: *on-init parami* ≤ ⇓ *ES* (*on-init param*)

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

**assumes** *on-new-root-refine*[*refine*]:
⟦*a.pre-new-root v0 s*; (*v0i*, *v0*) ∈ *V*; (*si*, *s*) ∈ *S*;
(*si*′, *s*′) ∈ *S*; *nf-inres* (*gbs-new-root gbs v0 s*) *s*′⟧
⟹ *on-new-root parami v0i si*′ ≤ ⇓ *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* ⟨*proof*⟩
**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*
    ⟨*proof*⟩

**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* ≡ λ*s*. *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 ≤n SPEC (gds-is-empty-stack gds)
  assumes new-root-discovered:
    ⟦pre-new-root v0 s⟧
      ⟹ gds-new-root gds v0 s ≤n SPEC (λs'.
        insert v0 (gen-discovered s) ⊆ gen-discovered s')
  assumes get-pending-incr:
    ⟦pre-get-pending s⟧ ⟹ gds-get-pending gds s ≤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 ≤n SPEC (λs'.
      gen-discovered s ⊆ gen-discovered s')
  assumes cross-edge-incr: pre-cross-edge u v s0 s
    ⟹ gds-cross-edge gds u v s ≤n SPEC (λs'.
      gen-discovered s ⊆ gen-discovered s')
  assumes back-edge-incr: pre-back-edge u v s0 s
    ⟹ gds-back-edge gds u v s ≤n SPEC (λs'.
      gen-discovered s ⊆ gen-discovered s')
  assumes discover-incr: pre-discover u v s0 s
    ⟹ gds-discover gds u v s ≤n SPEC (λs'.
      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*
$\leq$ *SPEC* ($\lambda$*s*. *gen-rwof s* $\wedge$ *gds-is-empty-stack gds s*)
⟨*proof*⟩

**lemma** *gds-new-root-refine*:
**assumes** *PNR*: *pre-new-root v0 s*
**shows** *gds-new-root gds v0 s*
$\leq$ *SPEC* ($\lambda$*s'*. *gen-rwof s'*
$\wedge$ *insert v0* (*gen-discovered s*) $\subseteq$ *gen-discovered s'* )
⟨*proof*⟩

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

**lemma** *gds-get-pending-refine*:
**assumes** *PRE*: *pre-get-pending s*
**shows** *gds-get-pending gds s* $\leq$ *SPEC* ($\lambda$(*u*,*Vs*,*s'*).
*post-get-pending u Vs s s'*
$\wedge$ *gen-discovered s* $\subseteq$ *gen-discovered s'*)
⟨*proof*⟩

**lemma** *gds-finish-refine*:
**assumes** *PRE*: *pre-finish u s0 s*
**shows** *gds-finish gds u s* $\leq$ *SPEC* ($\lambda$*s'*. *gen-rwof s'*
$\wedge$ *gen-discovered s* $\subseteq$ *gen-discovered s'*)
⟨*proof*⟩

**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'*)
⟨*proof*⟩

**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'*)
⟨*proof*⟩

**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'*)
⟨*proof*⟩

**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* ($\lambda$*s. insert v0* (*gen-discovered s0*) $\subseteq$ *gen-discovered s*)
⟨*proof*⟩

**theorem** *tailrec-impl*: *tailrec-impl* $\leq$ *gen-dfs*
⟨*proof*⟩

**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*
⟨*proof*⟩

**lemma** *tailrecT-impl*: *tailrec-implT* $\leq$ *gen-dfsT*
⟨*proof*⟩

**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* $\Rightarrow$ *'v rel*
**fixes** *stack* :: *'s* $\Rightarrow$ *'v list*
**fixes** *choose-pending* :: *'v* $\Rightarrow$ *'v option* $\Rightarrow$ *'s* $\Rightarrow$ *'s nres*
**begin**

**definition** *gen-step' s* $\equiv$ *do* { *ASSERT* (*gen-rwof s*);
*if gds-is-empty-stack gds s then do* {
*v0* $\leftarrow$ *SPEC* ($\lambda$*v0. v0* $\in$ *V0* $\wedge$ $\neg$ *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* :: $'s \Rightarrow 'v$ *rel*
  **and** *stack* :: $'s \Rightarrow 'v$ *list*
  **and** *choose-pending* :: $'v \Rightarrow 'v$ *option* $\Rightarrow 's \Rightarrow 's$ *nres*
  +
  **assumes** [*simp*]: *gds-is-empty-stack gds s* $\longleftrightarrow$ *stack s* = []
  **assumes** *init-spec*:
    *gds-init gds* $\leq_n$ *SPEC* ($\lambda s$. *stack s* = [] $\wedge$ *pending s* = {})
  **assumes** *new-root-spec*:
    ⟦*pre-new-root v0 s*⟧
      $\Longrightarrow$ *gds-new-root gds v0 s* $\leq_n$ *SPEC* ($\lambda s'$.
        *stack* $s'$ = [*v0*] $\wedge$ *pending* $s'$ = {*v0*}$\times$*E*‘‘{*v0*} $\wedge$
        *gen-discovered* $s'$ = *insert v0* (*gen-discovered s*))

  **assumes** *get-pending-fmt*: ⟦ *pre-get-pending s* ⟧ $\Longrightarrow$
    *do* {
      *let u* = *hd* (*stack s*);
      *vo* ← *SELECT* ($\lambda v$. (*u*,*v*)∈*pending s*);
      *s* ← *choose-pending u vo s*;
      *RETURN* (*u*,*vo*,*s*)
    }
  $\leq$ *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*}
  ⟧ $\Longrightarrow$
    *choose-pending u vo s* $\leq_n$ *SPEC* ($\lambda s'$.

(*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* $\leq_n$ *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* $\leq_n$ *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*
  ⟹ *gds-back-edge gds u v s* $\leq_n$ *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** *discover-spec*: *pre-discover u v s0 s*
  ⟹ *gds-discover gds u v s* $\leq_n$ *SPEC* (λ*s′*.
  *pending s′* = *pending s* ∪ ({*v*} × *E*‘‘{*v*}) ∧ *stack s′* = *v*#*stack s* ∧
  *gen-discovered s′* = *insert v* (*gen-discovered s*))

**begin**

**lemma** *gen-step′-refine*:
  ⟦*gen-rwof s*; *gen-cond s*⟧ ⟹ *gen-step′ s* ≤ *gen-step s*
  ⟨*proof*⟩

**lemma** *gen-dfs′-refine*: *gen-dfs′* ≤ *gen-dfs*
  ⟨*proof*⟩

**lemma** *gen-rwof′-imp-rwof*:
  **assumes** *NF*: *nofail gen-dfs*
  **assumes** *A*: *gen-rwof′ s*
  **shows** *gen-rwof s*
  ⟨*proof*⟩

**lemma** *reachable-invar*:
  *gen-rwof′ s* ⟹ *set* (*stack s*) ⊆ *reachable* ∧ *pending s* ⊆ *E*
  ∧ *set* (*stack s*) ⊆ *gen-discovered s* ∧ *distinct* (*stack s*)
  ∧ *pending s* ⊆ *set* (*stack s*) × *UNIV*

⟨*proof*⟩

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

**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*
⟨*proof*⟩

**schematic-goal** *gds-init-refine*: *?prop*
⟨*proof*⟩

**schematic-goal** *gds-new-root-refine*:
⟦*pre-new-root v0 s*; *gen-rwof′ s*⟧ ⟹ *gds-new-root gds v0 s* ≤ *SPEC ?*Φ
⟨*proof*⟩

**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*⟩

**schematic-goal** *gds-finish-refine*:

$\llbracket$*pre-finish u s0 s*; *post-choose-pending u None s0 s*$\rrbracket \Longrightarrow$ *gds-finish gds u s* $\leq$ *SPEC ?*$\Phi$
⟨*proof*⟩

**schematic-goal** *gds-cross-edge-refine*:
$\llbracket$*pre-cross-edge u v s0 s*; *post-choose-pending u* (*Some v*) *s0 s*$\rrbracket \Longrightarrow$ *gds-cross-edge gds u v s* $\leq$ *SPEC ?*$\Phi$
⟨*proof*⟩

**schematic-goal** *gds-back-edge-refine*:
$\llbracket$*pre-back-edge u v s0 s*; *post-choose-pending u* (*Some v*) *s0 s*$\rrbracket \Longrightarrow$ *gds-back-edge gds u v s* $\leq$ *SPEC ?*$\Phi$
⟨*proof*⟩

**schematic-goal** *gds-discover-refine*:
$\llbracket$*pre-discover u v s0 s*; *post-choose-pending u* (*Some v*) *s0 s*$\rrbracket \Longrightarrow$ *gds-discover gds u v s* $\leq$ *SPEC ?*$\Phi$
⟨*proof*⟩
**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*))
⟨*proof*⟩

**lemma** *rec-impl*: *rec-impl* $\leq$ *gen-dfs*
⟨*proof*⟩

**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* ≡ { (*s*,*s'*) .
*ss-stack s* = *map* (λ*u*. (*u*,*pending s'* `` {*u*})) (*stack s'*) ∧
*on-stack s* = *set* (*stack s'*) ∧
*visited s* = *dom* (*discovered s'*) ∧
*dom* (*finished s'*) = *dom* (*discovered s'*) − *set* (*stack s'*) ∧ — TODO: Hmm, this is an invariant of the abstract
*set* (*stack s'*) ⊆ *dom* (*discovered s'*) ∧
(*simple-state.more s*, *state.more s'*) ∈ *erel*
}

**lemma** *simple-state-relI*:
**assumes**
*dom* (*finished s'*) = *dom* (*discovered s'*) − *set* (*stack s'*)
*set* (*stack s'*) ⊆ *dom* (*discovered s'*)
(*m'*, *state.more s'*) ∈ *erel*
**shows** ((|
*ss-stack* = *map* (λ*u*. (*u*,*pending s'* `` {*u*})) (*stack s'*),
*on-stack* = *set* (*stack s'*),
*visited* = *dom* (*discovered s'*),
. . . = *m'*
|), *s'*)∈⟨*erel*⟩*simple-state-rel*
⟨*proof*⟩

**lemma** *simple-state-more-refine*[*param*]:
(*simple-state.more-update*, *state.more-update*)
∈ (*R* → *R*) → ⟨*R*⟩*simple-state-rel* → ⟨*R*⟩*simple-state-rel*
⟨*proof*⟩

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

**definition** *is-empty-stack-impl s* ≡ (*ss-stack s* = [])
**definition** *is-discovered-impl u s* ≡ (*u*∈*visited s*)
**definition** *is-finished-impl u s* ≡ (*u*∈*visited s* − (*on-stack s*))

**definition** *finish-impl u s* ≡ *do* {
*ASSERT* (*ss-stack s* ≠ [] ∧ *u*∈*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 ≡ do {
    ASSERT (ss-stack s ≠ []);
    let (u,Vs) = hd (ss-stack s);
    if Vs = {} then
      RETURN (u,None,s)
    else do {
      v ← SPEC (λv. v∈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 ≡ do {
  ASSERT (v∉on-stack s ∧ v∉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 ≡ do {
  ASSERT (v0∉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 = (λu v s. RETURN s) ,
  gbs-cross-edge = (λu 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* ⟨*proof*⟩

  **definition** *get-pending s* $\equiv \bigcup$(*set* (*map* ($\lambda$(*u*, *Vs*). {*u*}× *Vs*) (*ss-stack s*)))
  **definition** *get-stack s* ≡ *map fst* (*ss-stack s*)
  **definition** *choose-pending*
    :: ′*v* ⇒ ′*v option* ⇒ (′*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* ⟨*proof*⟩
**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** $S = \langle ES\rangle simple\text{-}state\text{-}rel$
**begin**

**lemma** *init-impl*: $(ei,\ e) \in ES \Longrightarrow$
$c.init\text{-}impl\ ei \leq\Downarrow(\langle ES\rangle simple\text{-}state\text{-}rel)\ (RETURN\ (a.empty\text{-}state\ e))$
$\langle proof\rangle$

**lemma** *new-root-impl*:
$\llbracket a.gen\text{-}dfs.pre\text{-}new\text{-}root\ v0\ s;$
$(v0i,\ v0){\in}Id;\ (si,\ s) \in \langle ES\rangle simple\text{-}state\text{-}rel\rrbracket$
$\Longrightarrow c.new\text{-}root\text{-}impl\ v0\ si \leq\Downarrow(\langle ES\rangle simple\text{-}state\text{-}rel)\ (RETURN\ (a.new\text{-}root\ v0$
$s))$
$\langle proof\rangle$

**lemma** *get-pending-impl*:
$\llbracket a.gen\text{-}dfs.pre\text{-}get\text{-}pending\ s;\ (si,\ s) \in \langle ES\rangle simple\text{-}state\text{-}rel\rrbracket$
$\Longrightarrow c.get\text{-}pending\text{-}impl\ si$
$\leq \Downarrow (Id \times_r Id \times_r \langle ES\rangle simple\text{-}state\text{-}rel)\ (a.get\text{-}pending\ s)$
$\langle proof\rangle$

**lemma** *inres-get-pending-None-conv*: $inres\ (a.get\text{-}pending\ s0)\ (v,\ None,\ s)$
$\longleftrightarrow s{=}s0 \wedge v{=}hd\ (stack\ s0) \wedge pending\ s0``\{hd\ (stack\ s0)\} = \{\}$
$\langle proof\rangle$

**lemma** *inres-get-pending-Some-conv*: $inres\ (a.get\text{-}pending\ s0)\ (v{,}Some\ Vs{,}s)$
$\longleftrightarrow v = hd\ (stack\ s) \wedge s = s0(\!|pending := pending\ s0 - \{(hd\ (stack\ s0),$
$Vs)\}|\!)$
$\wedge\ (hd\ (stack\ s0),\ Vs) \in pending\ s0$
$\langle proof\rangle$

**lemma** *finish-impl*:
$\llbracket a.gen\text{-}dfs.pre\text{-}finish\ v\ s0\ s;\ (vi,\ v){\in}Id;\ (si,\ s) \in \langle ES\rangle simple\text{-}state\text{-}rel\rrbracket$
$\Longrightarrow c.finish\text{-}impl\ v\ si \leq\Downarrow(\langle ES\rangle simple\text{-}state\text{-}rel)\ (RETURN\ (a.finish\ v\ s))$
$\langle proof\rangle$

**lemma** *cross-edge-impl*:
$\llbracket a.gen\text{-}dfs.pre\text{-}cross\text{-}edge\ u\ v\ s0\ s;$
$(ui,\ u){\in}Id;\ (vi,\ v){\in}Id;\ (si,\ s) \in \langle ES\rangle simple\text{-}state\text{-}rel\rrbracket$
$\Longrightarrow (si,\ a.cross\text{-}edge\ u\ v\ s) \in \langle ES\rangle simple\text{-}state\text{-}rel$
$\langle proof\rangle$

**lemma** *back-edge-impl*:
$\llbracket a.gen\text{-}dfs.pre\text{-}back\text{-}edge\ u\ v\ s0\ s;$
$(ui,\ u){\in}Id;\ (vi,\ v){\in}Id;\ (si,\ s) \in \langle ES\rangle simple\text{-}state\text{-}rel\rrbracket$
$\Longrightarrow (si,\ a.back\text{-}edge\ u\ v\ s) \in \langle ES\rangle simple\text{-}state\text{-}rel$
$\langle proof\rangle$

**lemma** *discover-impl*:

$[\![a.gen\text{-}dfs.pre\text{-}discover\ u\ v\ s0\ s;\ (ui,\ u) \in Id;\ (vi,\ v) \in Id;\ (si,\ s) \in \langle ES \rangle simple\text{-}state\text{-}rel]\!]$
$\Longrightarrow c.discover\text{-}impl\ ui\ vi\ si \leq \Downarrow(\langle ES \rangle simple\text{-}state\text{-}rel)\ (RETURN\ (a.discover\ u\ v\ s))$
⟨*proof*⟩

**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*
⟨*proof*⟩

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*]: $c.gen\text{-}dfs \leq \Downarrow(\langle ES \rangle simple\text{-}state\text{-}rel)\ a.it\text{-}dfs$
⟨*proof*⟩

**lemma** *simple-refineT*[*refine*]: $c.gen\text{-}dfsT \leq \Downarrow(\langle ES \rangle simple\text{-}state\text{-}rel)\ a.it\text{-}dfsT$
⟨*proof*⟩

Link with tail-recursive implementation

**sublocale** *tailrec-impl G c.gds*
⟨*proof*⟩

**lemma** *simple-tailrec-refine*[*refine*]: $tailrec\text{-}impl \leq \Downarrow(\langle ES \rangle simple\text{-}state\text{-}rel)\ a.it\text{-}dfs$
⟨*proof*⟩

**lemma** *simple-tailrecT-refine*[*refine*]: $tailrec\text{-}implT \leq \Downarrow(\langle ES \rangle simple\text{-}state\text{-}rel)\ a.it\text{-}dfsT$
⟨*proof*⟩

Link to recursive implementation

**lemma** *reachable-invar*:
**assumes** *c.gen-rwof s*
**shows** $set\ (map\ fst\ (ss\text{-}stack\ s)) \subseteq visited\ s$
$\wedge\ distinct\ (map\ fst\ (ss\text{-}stack\ s))$
⟨*proof*⟩

**sublocale** *rec-impl G c.gds get-pending get-stack choose-pending*
⟨*proof*⟩

**lemma** *simple-rec-refine*[*refine*]: $rec\text{-}impl \leq \Downarrow(\langle ES \rangle simple\text{-}state\text{-}rel)\ a.it\text{-}dfs$
⟨*proof*⟩

**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* ≡ ⟨*s-rel*,*vis-rel*,*erel*⟩*ss-impl-rel*
**shows**
(*ss-stack-impl*, *ss-stack*) ∈ *R* → *s-rel*
(*ss-on-stack-impl*, *on-stack*) ∈ *R* → *vis-rel*
(*ss-visited-impl*, *visited*) ∈ *R* → *vis-rel*
(*simple-state-impl.more*, *simple-state.more*) ∈ *R* → *erel*
(*ss-stack-impl-update*, *ss-stack-update*) ∈ (*s-rel* → *s-rel*) → *R* → *R*
(*ss-on-stack-impl-update*, *on-stack-update*) ∈ (*vis-rel* → *vis-rel*) → *R* → *R*
(*ss-visited-impl-update*, *visited-update*) ∈ (*vis-rel* → *vis-rel*) → *R* → *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*
⟨*proof*⟩

### 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* ⟨*proof*⟩
**lemma** [*autoref-op-pat-def*]: *op-nos-on-stack-update f s*
≡ *OP* (*op-nos-on-stack-update f*)$*s* ⟨*proof*⟩

**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* ≡ ⟨*s-rel*,*vis-rel*,*erel*⟩*ssnos-impl-rel*
**shows**
(*ssnos-stack-impl*, *ss-stack*) ∈ *R* → *s-rel*
(*ssnos-visited-impl*, *visited*) ∈ *R* → *vis-rel*
(*simple-state-nos-impl.more*, *simple-state.more*) ∈ *R* → *erel*
(*ssnos-stack-impl-update*, *ss-stack-update*) ∈ (*s-rel* → *s-rel*) → *R* → *R*
(λ*x*. *x*, *op-nos-on-stack-update f*) ∈ *R* → *R*
(*ssnos-visited-impl-update*, *visited-update*) ∈ (*vis-rel* → *vis-rel*) → *R* → *R*
(*simple-state-nos-impl.more-update*, *simple-state.more-update*) ∈ (*erel* → *erel*) → *R* → *R*
(λ*ns* - *ps vs*. *simple-state-nos-impl-ext ns ps vs*, *simple-state-ext*)
∈ *s-rel* → *ANY-rel* → *vis-rel* → *erel* → *R*
⟨*proof*⟩

### 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*×′*b*) *set*) *vis-rel erel* ≡

{((|*ssns-visited-impl* = *visi*, ... = *mi*|),
(|*ss-stack* = *s*, *on-stack* = *os*, *visited* = *vis*, ... = *m*|)) |
*visi mi s os vis m*.
(*visi*, *vis*) ∈ *vis-rel* ∧
(*mi*, *m*) ∈ *erel*
}

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

**definition** *op-ns-on-stack-update*
:: (- *set* ⇒ - *set*) ⇒ (-,-)*simple-state-scheme* ⇒ -
**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* ⟨*proof*⟩
**lemma** [*autoref-op-pat-def*]: *op-ns-on-stack-update f s*
≡ *OP* (*op-ns-on-stack-update f*)$*s* ⟨*proof*⟩

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

**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*
)
})
⟨*proof*⟩

**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** $R \equiv \langle ANY\text{-}rel,vis\text{-}rel,erel\rangle ssns\text{-}impl\text{-}rel$
**shows**
$(ssns\text{-}visited\text{-}impl,\ visited) \in R \to vis\text{-}rel$
$(simple\text{-}state\text{-}ns\text{-}impl.more,\ simple\text{-}state.more) \in R \to erel$
$\bigwedge f.\ (\lambda x.\ x,\ op\text{-}ns\text{-}stack\text{-}update\ f) \in R \to R$
$\bigwedge f.\ (\lambda x.\ x,\ op\text{-}ns\text{-}on\text{-}stack\text{-}update\ f) \in R \to R$
$(ssns\text{-}visited\text{-}impl\text{-}update,\ visited\text{-}update) \in (vis\text{-}rel \to vis\text{-}rel) \to R \to R$
$(simple\text{-}state\text{-}ns\text{-}impl.more\text{-}update,\ simple\text{-}state.more\text{-}update) \in (erel \to erel) \to R \to R$
$(\lambda\text{-}\ \text{-}\ ps\ vs.\ simple\text{-}state\text{-}ns\text{-}impl\text{-}ext\ ps\ vs,\ simple\text{-}state\text{-}ext)$
$\in ANY1\text{-}rel \to ANY2\text{-}rel \to vis\text{-}rel \to erel \to R$
$\langle proof \rangle$

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

**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-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** $visited\ s = visited\ s' \cup R$
**and** $simple\text{-}state.more\ s = simple\text{-}state.more\ s'$
$\langle proof\rangle$

**lemma**
**assumes** $(s,s')\in restr\text{-}rel\ R$
**shows** *restr-rel-stackD*: $(ss\text{-}stack\ s,\ ss\text{-}stack\ s') \in \langle Id \times_r \{(U,U').\ U-R = U'\}\rangle list\text{-}rel$
**and** *restr-rel-vis-djD*: $visited\ s' \cap R = \{\}$
$\langle proof\rangle$

**context fixes** $R :: 'v\ set$ **begin**
**definition** [*to-relAPP*]: $restr\text{-}simple\text{-}state\text{-}rel\ ES \equiv \{\ (s,s')\ .$
$(ss\text{-}stack\ s,\ map\ (\lambda u.\ (u,pending\ s'\ ``\ \{u\}))\ (stack\ s'))$
$\in \langle Id \times_r \{(U,U').\ U-R = U'\}\rangle list\text{-}rel\ \wedge$
$on\text{-}stack\ s = set\ (stack\ s') \wedge$
$visited\ s = dom\ (discovered\ s') \cup R \wedge dom\ (discovered\ s') \cap R = \{\} \wedge$
$dom\ (finished\ s') = dom\ (discovered\ s') - set\ (stack\ s') \wedge$
$set\ (stack\ s') \subseteq dom\ (discovered\ s') \wedge$
$(simple\text{-}state.more\ s,\ state.more\ s') \in ES$
$\}$
**end**

**lemma** *restr-simple-state-rel-combine*:
$\langle ES\rangle restr\text{-}simple\text{-}state\text{-}rel\ R = restr\text{-}rel\ R\ O\ \langle ES\rangle simple\text{-}state\text{-}rel$
$\langle proof\rangle$

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** $G :: ('v,\ 'more)\ graph\text{-}rec\text{-}scheme$
**and** $R$
**begin**
**sublocale** *pre-simple-impl* $G\ \langle proof\rangle$

**abbreviation** $rel \equiv restr\text{-}rel\ R$

**definition** $gbs' \equiv gbs\ (\!|$

*gbs-init* := $\lambda$*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* ⟨*proof*⟩

**sublocale** *tailrec-impl-defs G gds* ⟨*proof*⟩
**end**

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

**assumes** [*simp*]: *on-cross-edge parami* = ($\lambda$*u v s*. *RETURN* (*simple-state.more s*))
**assumes** [*simp*]: *on-back-edge parami* = ($\lambda$*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* ⟨*proof*⟩

**lemma** *is-break-param'*[*param*]: (*is-break parami*, *is-break parami*)∈*rel* → *bool-rel*
⟨*proof*⟩

**lemma** *do-init-refine*[*refine*]: *do-init* $\leq \Downarrow$ *rel* (*a.c.do-init*)
⟨*proof*⟩

**lemma** *gen-cond-param*: (*gen-cond*,*a.c.gen-cond*)$\in$*rel* $\to$ *bool-rel*
⟨*proof*⟩

**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*
⟨*proof*⟩

**lemma** *pred-rel-simps*:
**assumes** ($s$,$s'$)$\in$*rel*
**shows** *a.c.is-discovered-impl u s* $\longleftrightarrow$ *a.c.is-discovered-impl u* $s' \vee u \in R$
**and** *a.c.is-empty-stack-impl s* $\longleftrightarrow$ *a.c.is-empty-stack-impl* $s'$
⟨*proof*⟩

**lemma** *no-pending-refine*:
**assumes** ($s$,$s'$)$\in$*rel* $\neg$*a.c.is-empty-stack-impl* $s'$
**shows** (*hd* (*ss-stack s*) = ($u$,{})) $\Longrightarrow$ *hd* (*ss-stack* $s'$) = ($u$,{})
⟨*proof*⟩

**lemma** *do-new-root-refine*[*refine*]:
⟦ (*v0i*,*v0*)$\in$*Id*; (*si*,*s*)$\in$*rel*; *v0*$\notin R$ ⟧
$\Longrightarrow$ *do-new-root v0i si* $\leq \Downarrow$ *rel* (*a.c.do-new-root v0 s*)
⟨*proof*⟩

**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* $u'$ $s'$)
⟨*proof*⟩

**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*$-R$))

⟨*proof*⟩

**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* $\times_r$ ⟨*Id*⟩*option-rel* $\times_r$ *rel*) (*inf*
(*get-pending-impl* $s'$)
(*SPEC* ($\lambda$(-,*Vs*,-). *case Vs of None* $\Rightarrow$ *True* | *Some* $v \Rightarrow v \notin R$))))
($\Downarrow$(*Id* $\times_r$ ⟨*Id*⟩*option-rel* $\times_r$ *rel*) (
*SPEC* ($\lambda(u,Vs,s'')$. $\exists\, v$. *Vs*=*Some* $v \wedge v{\in}R \wedge s''{=}s'$)
)))
⟨*proof*⟩

**lemma** *do-discover-refine*[*refine*]:
⟦ $(s,\ s') \in$ *rel*; $(u,u'){\in}$*Id*; $(v,v'){\in}$*Id*; $v' \notin R$ ⟧
$\Longrightarrow$ *do-discover* $u\ v\ s \le \Downarrow$ *rel* (*a.c.do-discover* $u'\ v'\ s'$)
⟨*proof*⟩

**lemma** *aux-R-node-discovered*: ⟦$(s,s'){\in}$*rel*; $v{\in}R$⟧ $\Longrightarrow$ *is-discovered-impl* $v\ s$
⟨*proof*⟩

**lemma** *re-refine-aux*: *gen-dfs* $\le \Downarrow$*rel a.c.gen-dfs*
⟨*proof*⟩

**theorem** *re-refine-aux2*: *gen-dfs* $\le\Downarrow$(*rel O* ⟨*ES*⟩*simple-state-rel*) *a.a.it-dfs*
⟨*proof*⟩

**theorem** *re-refine*: *gen-dfs* $\le\Downarrow$(⟨*ES*⟩*restr-simple-state-rel R*) *a.a.it-dfs*
⟨*proof*⟩

**sublocale** *tailrec-impl G gds*
⟨*proof*⟩

**lemma** *tailrec-refine*: *tailrec-impl* $\le \Downarrow$(⟨*ES*⟩*restr-simple-state-rel R*) *a.a.it-dfs*
⟨*proof*⟩

**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*: $state.more\ s = state.more\ s' \Longrightarrow break\ s = break\ s'$
⟨*proof*⟩

**lemma** [*simp*]: $s⦇\ state.more := ⦇\ break = foo\ ⦈\ ⦈ = s\ ⦇\ break := foo\ ⦈$
⟨*proof*⟩

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* ⟨*proof*⟩

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*
⟨*proof*⟩

**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*⟩

**lemma** *cyccI′*:
  **assumes** *graph G*
  **and** *FR*: *finite* (*graph-defs.reachable G*)
  **shows** *cycc G*
⟨*proof*⟩

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*⟩

### 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* ⟷ *back-edges s* $\neq$ {})
  ⟨*proof*⟩

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* ($\lambda s.$ *break s* ⟷ *back-edges s* $\neq$ {})
    ⟨*proof*⟩

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** $break\ s \longleftrightarrow \neg acyclic\ (E \cap reachable \times UNIV)$
$\langle proof \rangle$

Again, we have a short two-line proof:

**lemma** *cycc-correct-aux-short-proof*:
  **assumes** *NC*: $\neg cond\ s$
  **shows** $break\ s \longleftrightarrow \neg acyclic\ (E \cap reachable \times UNIV)$
  $\langle proof \rangle$

**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* {
  $ASSERT\ (fb\text{-}graph\ G)$;
  $SPEC\ (\lambda r.\ r \longleftrightarrow \neg acyclic\ (g\text{-}E\ G \cap ((g\text{-}E\ G)^*\ ``\ g\text{-}V0\ G) \times UNIV))$}

**theorem** *cyc-checker-correct*: $cyc\text{-}checker\ G \leq cyc\text{-}checker\text{-}spec\ G$
  $\langle proof \rangle$

The same for the total correct variant:

**definition** *cyc-checkerT-spec G* $\equiv$ *do* {
  $ASSERT\ (graph\ G \wedge finite\ (graph\text{-}defs.reachable\ G))$;
  $SPEC\ (\lambda r.\ r \longleftrightarrow \neg acyclic\ (g\text{-}E\ G \cap ((g\text{-}E\ G)^*\ ``\ g\text{-}V0\ G) \times UNIV))$}

**theorem** *cyc-checkerT-correct*: $cyc\text{-}checkerT\ G \leq cyc\text{-}checkerT\text{-}spec\ G$
  $\langle proof \rangle$

### 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*
⟨*proof*⟩

**interpretation** *cycc-impl*: *simple-impl-defs G cycc-params-impl cycc-params*
**for** *G* ⟨*proof*⟩

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*));

$s \leftarrow$ *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*
⟨*proof*⟩

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* $\leq \Downarrow$ *cycc-rel it-dfs*
*cycc-impl.rec-impl TYPE('more) G* $\leq \Downarrow$ *cycc-rel it-dfs*
*cycc-impl.tailrec-implT TYPE('more) G* $\leq \Downarrow$ *cycc-rel it-dfsT*
⟨*proof*⟩

**end**

Finally, we get correctness of our cyclicity checker implementations

**lemma** *cyc-checker-impl-refine*: *cyc-checker-impl G* $\leq \Downarrow$*Id (cyc-checker G)*
⟨*proof*⟩

**lemma** *cyc-checker-rec-impl-refine*:
*cyc-checker-rec-impl G* $\leq \Downarrow$*Id (cyc-checker G)*
⟨*proof*⟩

**lemma** *cyc-checker-implT-refine*: *cyc-checker-implT G* $\leq \Downarrow$*Id (cyc-checkerT G)*
⟨*proof*⟩

### 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* $\equiv$ {
  $(\llparenthesis$*break-impl* = *bi*, … = *mi*$\rrparenthesis$,$\llparenthesis$*break* = *b*, … = *m*$\rrparenthesis$) | *bi mi b m*.
    (*bi*,*b*)∈*bool-rel* ∧ (*mi*,*m*)∈*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* ≡ ⟨*ns-rel*,*vis-rel*,⟨*erel*⟩*cycc-state-erel*⟩*ss-impl-rel*
  **shows**
   (*cycc-state-impl′-ext*, *cycc-state-impl-ext*) ∈ *bool-rel* → *erel* → ⟨*erel*⟩*cycc-state-erel*
    (*break-impl*, *cycc-state-impl.break*) ∈ *R* → *bool-rel*
  ⟨*proof*⟩

Finally, we can synthesize an implementation for our cyclicity checker, using the standard Autoref-approach:

**schematic-goal** *cyc-checker-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*]
    *TYRELI*[**where** *R*=⟨*V* $\times_r$ ⟨*V*⟩*list-set-rel*⟩*ras-rel*]
  **shows** *nres-of* (*?c*::*?′c dres*) ≤⇓*?R* (*cyc-checker-impl G*)
  ⟨*proof*⟩
**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*) ∈ ⟨*Rm*, *Id*⟩*g-impl-rel-ext*
  **assumes** *4*: *cyc-checker-code Gi* = *dRETURN x*
  **shows** *x* ⟷ (¬*acyclic* (*g-E G* ∩ ((*g-E G*)$^*$ `` *g-V0 G*) × *UNIV*))
⟨*proof*⟩

We can repeat the same boilerplate for the recursive version of the algorithm:

**schematic-goal** *cyc-checker-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** $R=\langle V\rangle$*dflt-ahs-rel*]
*TYRELI*[**where** $R=\langle V \times_r \langle V\rangle$*list-set-rel*$\rangle$*ras-rel*]
**shows** *nres-of* (*?c*::*?'c dres*) $\leq\Downarrow$*?R* (*cyc-checker-rec-impl G*)
$\langle$*proof*$\rangle$
**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-impl-rel-ext*
**assumes** *4*: *cyc-checker-rec-code Gi* = *dRETURN x*
**shows** *x* $\longleftrightarrow$ ($\neg$*acyclic* (*g-E G* $\cap$ ((*g-E G*)$^*$ ‘‘ *g-V0 G*) $\times$ *UNIV*))
$\langle$*proof*$\rangle$

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-impl-rel-ext*
**notes** [*unfolded V-def*,*autoref-tyrel*] =
*TYRELI*[**where** $R=\langle V\rangle$*dflt-ahs-rel*]
*TYRELI*[**where** $R=\langle V \times_r \langle V\rangle$*list-set-rel*$\rangle$*ras-rel*]
**shows** *RETURN* (*?c*::*?'c*) $\leq\Downarrow$*?R* (*cyc-checker-implT G*)
$\langle$*proof*$\rangle$
**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-impl-rel-ext*
**shows** *cyc-checker-codeT Gi* $\longleftrightarrow$ ($\neg$*acyclic* (*g-E G* $\cap$ ((*g-E G*)$^*$ ‘‘ *g-V0 G*) $\times$ *UNIV*))
$\langle$*proof*$\rangle$

**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 × 'v) option*

**type-synonym** *'v fp0-param = ('v, ('v,unit) fp0-state-ext) parameterization*

**lemma** [*simp*]: *s⦇ state.more := ⦇ ppath = foo ⦈ ⦈ = s ⦇ ppath := foo ⦈*
 ⟨*proof*⟩

**abbreviation** *no-path ≡ ⦇ ppath = None ⦈*
**abbreviation** *a-path p v ≡ ⦇ ppath = Some (p,v) ⦈*

**definition** *fp0-params :: ('v ⇒ bool) ⇒ 'v fp0-param*
 **where** *fp0-params P ≡ ⦇*
 *on-init = RETURN no-path,*
 *on-new-root = λv0 s. if P v0 then RETURN (a-path [] v0) else RETURN no-path,*
 *on-discover = λu 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 = λu s. RETURN (state.more s),*
 *on-back-edge = λu v s. RETURN (state.more s),*
 *on-cross-edge = λu v s. RETURN (state.more s),*
 *is-break = λs. ppath s ≠ 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* ⟨*proof*⟩

**locale** *fp0 = param-DFS G fp0-params P*
 **for** *G* **and** *P :: 'v ⇒ bool*
**begin**

 **lemma** [*simp*]:
 *ppath (empty-state (⦇ppath = e⦈)) = e*
 ⟨*proof*⟩

**lemma** [*simp*]:
  *ppath* $(s(\!|state.more := state.more\ s'|\!)) = ppath\ s'$
  ⟨*proof*⟩

**sublocale** *DFS* **where** *param* = *fp0-params P*
  ⟨*proof*⟩

**end**

**lemma** *fp0I*: **assumes** *fb-graph G* **shows** *fp0 G*
⟨*proof*⟩

**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*⟩

**context** *fp0* **begin**

**lemma** *i-no-path-no-P-discovered*:
  *is-invar* ($\lambda s.$ *ppath s* = *None* $\longrightarrow$ *dom* (*discovered s*) $\cap$ *Collect P* = {})
⟨*proof*⟩

**lemma** *i-path-to-P*:
  *is-invar* ($\lambda s.$ *ppath s* = *Some* (*vs*,*v*) $\longrightarrow$ *P v*)
⟨*proof*⟩

**lemma** *i-path-invar*:
  *is-invar* ($\lambda s.$ *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*⟩
**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*
⟨*proof*⟩

**lemma** *path-invar-empty*:
**assumes** *ppath s = Some (vs,v)*
**and** *vs* = []
**shows** $v \in$ *V0 path E v vs v*
⟨*proof*⟩

**lemma** *fp0-correct*:
**assumes** $\neg$*cond s*
**shows** *case ppath s of*
*None* $\Rightarrow \neg(\exists\, v0 \in V0.\ \exists\, v.\ (v0,v) \in E^* \wedge P\ v)$
| *Some (p,v)* $\Rightarrow (\exists\, v0 \in V0.\ path\ E\ v0\ p\ v \wedge P\ v \wedge distinct\ (p@[v]))$
⟨*proof*⟩

**end**

**context** *fp0* **begin**
**lemma** *fp0-correct*: *it-dfs* $\leq$ *SPEC* ($\lambda s.$ *case ppath s of*
*None* $\Rightarrow \neg(\exists\, v0 \in V0.\ \exists\, v.\ (v0,v) \in E^* \wedge P\ v)$
| *Some (p,v)* $\Rightarrow (\exists\, v0 \in V0.\ path\ E\ v0\ p\ v \wedge P\ v \wedge distinct\ (p@[v])))$
⟨*proof*⟩
**end**

### Basic Interface

Use this interface, rather than the internal stuff above!

**type-synonym** *'v fp-result* = (*'v list* $\times$ *'v*) *option*
**definition** *find-path0-pred G P* $\equiv \lambda r.$ *case r of*
*None* $\Rightarrow$ (*g-E G*)$^*$ `` *g-V0 G* $\cap$ *Collect P* = {}
| *Some (vs,v)* $\Rightarrow$ *P v* $\wedge$ *distinct (vs@[v])* $\wedge$ ($\exists$ *v0* $\in$ *g-V0 G. path (g-E G) v0 vs v*)

**definition** *find-path0-spec*
:: (*'v*, -) *graph-rec-scheme* $\Rightarrow$ (*'v* $\Rightarrow$ *bool*) $\Rightarrow$ *'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* $\equiv$ *do* {
*ASSERT* (*fb-graph G*);
*SPEC* (*find-path0-pred G P*)
}

**definition** *find-path0*

:: $('v, 'more)$ *graph-rec-scheme* $\Rightarrow$ $('v \Rightarrow bool) \Rightarrow$ *'v fp-result nres*
**where** *find-path0 G P* $\equiv$ *do* {
*ASSERT* (*fp0 G*);
*s* $\leftarrow$ *fp0.it-dfs TYPE*(*'more*) *G P*;
*RETURN* (*ppath s*)
}

**lemma** *find-path0-correct*:
**shows** *find-path0 G P* $\leq$ *find-path0-spec G P*
⟨*proof*⟩

**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-invar E R P* $\equiv$ *E* ‘‘ *R* $\subseteq$ *R* $\wedge$ *R* $\cap$ *Collect P* = {}

**lemma** *restr-invar-triv*[*simp*, *intro*!]: *restr-invar E* {} *P*
⟨*proof*⟩

**lemma** *restr-invar-imp-not-reachable*: *restr-invar E R P* $\Longrightarrow$ $E^*$ ‘‘*R* $\cap$ *Collect P* = {}
⟨*proof*⟩

**type-synonym** *'v fpr-result* = *'v set* + (*'v list* $\times$ *'v*)
**definition** *find-path0-restr-pred G P R* $\equiv$ $\lambda r$.
*case r of*
*Inl R'* $\Rightarrow$ *R'* = *R* $\cup$ (*g-E G*)$^*$ ‘‘ *g-V0 G* $\wedge$ *restr-invar* (*g-E G*) *R' P*
| *Inr* (*vs*,*v*) $\Rightarrow$ *P v* $\wedge$ ($\exists$ *v0* $\in$ *g-V0 G* $-$ *R*. *path* (*rel-restrict* (*g-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-path0-restr-spec G P R* $\equiv$ *do* {
*ASSERT* (*fb-graph G* $\wedge$ *restr-invar* (*g-E G*) *R P*);
*SPEC* (*find-path0-restr-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-rec-scheme* ⇒ ($'v$ ⇒ *bool*) ⇒ $'v$ *set* ⇒ $'v$ *fpr-result nres*
**where** *find-path0-restr G P R* ≡ *do* {
*ASSERT* (*fb-graph G*);
*ASSERT* (*fp0* (*graph-restrict G R*));
*s* ← *fp0.it-dfs TYPE*($'more$) (*graph-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*⟩

### 2.2.3 Path of Minimal Length One, with Restriction

**definition** *find-path1-restr-pred G P R* ≡ $\lambda r$.
*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* ($\lambda$*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* - $\Rightarrow$ *RETURN f0*
| *Inr* (*vs*,*v*) $\Rightarrow$ *RETURN* (*Inr* (*v0*#*vs*,*v*))
}) (*Inl R*)

**definition** *find-path1-tailrec-invar G P R0 it s* $\equiv$
*case s of*
*Inl R* $\Rightarrow$ *R* = *R0* $\cup$ (*g-E G*)$^+$ `` (*g-V0 G* $-$ *it*) $\wedge$ *restr-invar* (*g-E G*) *R P*
| *Inr* (*vs*, *v*) $\Rightarrow$ *P v* $\wedge$ *vs* $\neq$ [] $\wedge$ ($\exists$ *v0* $\in$ *g-V0 G* $-$ *it*. *path* (*g-E G* $\cap$ *UNIV* $\times$ $-$*R0*) *v0 vs v*)

**lemma** *find-path1-restr-correct*:
**shows** *find-path1-restr G P R* $\leq$ *find-path1-restr-spec G P R*
⟨*proof*⟩

**definition** *find-path1-pred G P* $\equiv$ $\lambda r$.
*case r of*
*None* $\Rightarrow$ (*g-E G*)$^+$ `` *g-V0 G* $\cap$ *Collect P* = {}
| *Some* (*vs*, *v*) $\Rightarrow$ *P v* $\wedge$ *vs* $\neq$ [] $\wedge$ ($\exists$ *v0* $\in$ *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* $\equiv$ *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* $\Rightarrow$ ('*v* $\Rightarrow$ *bool*) $\Rightarrow$ '*v fp-result nres*
**where** *find-path1 G P* $\equiv$ *do* {
*r* $\leftarrow$ *find-path1-restr-spec G P* {};
*case r of*
*Inl* - $\Rightarrow$ *RETURN None*
| *Inr vsv* $\Rightarrow$ *RETURN* (*Some vsv*)
}

**lemma** *find-path1-correct*:
**shows** *find-path1 G P* $\leq$ *find-path1-spec G P*
⟨*proof*⟩

### 2.2.5 Implementation

**record** '*v fp0-state-impl* = '*v simple-state* +
*ppath* :: ('*v list* $\times$ '*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'*
  ⟨*proof*⟩

**lemma** *fp0-ss-rel-ppath-cong*[*simp*]:
  (*s*,*s'*)∈⟨*fp0-erel*⟩*simple-state-rel* ⟹ *fp0-state-impl.ppath s* = *fp0-state.ppath s'*
  ⟨*proof*⟩

**lemma** *fp0i-cong*[*cong*]: *simple-state.more s* = *simple-state.more s'*
  ⟹ *fp0-state-impl.ppath s* = *fp0-state-impl.ppath s'*
  ⟨*proof*⟩

**lemma** *fp0-erelI*: *p*=*p'*
  ⟹ ((| *fp0-state-impl.ppath* = *p* |), (| *fp0-state.ppath* = *p'*|))∈*fp0-erel*
  ⟨*proof*⟩

**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* = λ*v0 s*.
    *if P v0 then RETURN* (*a-path-impl* [] *v0*) *else RETURN no-path-impl*,
  *on-discover* = λ*u 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* = λ*u s*. *RETURN* (*simple-state.more s*),
  *on-back-edge* = λ*u v s*. *RETURN* (*simple-state.more s*),
  *on-cross-edge* = λ*u v s*. *RETURN* (*simple-state.more s*),
  *is-break* = λ*s*. *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* ⟨*proof*⟩

**locale** *fp0-restr* = *fb-graph*
**begin**
  **sublocale** *fp0?*: *fp0 graph-restrict G R*
    ⟨*proof*⟩

**sublocale** *impl*: *restricted-impl G fp0-params P fp0-params-impl P fp0-erel R*
⟨*proof*⟩
**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*⟩

**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*⟩

### 2.2.6 Synthesis of Executable Code

**record** (′*v*,′*si*,′*nsi*)*fp0-state-impl*′ = (′*si*,′*nsi*)*simple-state-nos-impl* +
*ppath-impl* :: (′*v list* × ′*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** $R \equiv \langle$*ns-rel*,*vis-rel*,$\langle$*erel*$\rangle$*fp0-state-erel*$\rangle$*ssnos-impl-rel*
**shows**
(*fp0-state-impl′-ext*, *fp0-state-impl-ext*)
$\in \langle\langle Id\rangle$*list-rel* $\times_r$ *Id*$\rangle$*option-rel* $\rightarrow$ *erel* $\rightarrow \langle$*erel*$\rangle$*fp0-state-erel*
(*ppath-impl*, *fp0-state-impl.ppath*) $\in R \rightarrow \langle\langle Id\rangle$*list-rel* $\times_r$ *Id*$\rangle$*option-rel*
$\langle$*proof*$\rangle$

**schematic-goal** *find-path0-code*:
**fixes** $G$ :: (*′v* :: *hashable*, -) *graph-rec-scheme*
**assumes** [*autoref-rules*]:
(*Gi*, *G*) $\in \langle$*Rm*, *Id*$\rangle$*g-impl-rel-ext*
(*Pi*, *P*) $\in$ *Id* $\rightarrow$ *bool-rel*
**notes** [*autoref-tyrel*] = *TYRELI*[**where** *R*=$\langle$*Id*::(*′v*$\times$*′v*) *set*$\rangle$*dflt-ahs-rel*]
**shows** (*nres-of* (*?c*::*?′c dres*), *find-path0-impl G P*) $\in$ *?R*
$\langle$*proof*$\rangle$

**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** ($\lambda$*G P*. *nres-of* (*find-path0-code G P*), *find-path0-spec*)
$\in \langle$*Rm*, *Rv*$\rangle$*g-impl-rel-ext* $\rightarrow$ (*Rv* $\rightarrow$ *bool-rel*)
$\rightarrow \langle\langle\langle Rv\rangle$*list-rel* $\times_r$ *Rv*$\rangle$*option-rel*$\rangle$*nres-rel*
$\langle$*proof*$\rangle$
**lemmas** *find-path0-autoref*[*autoref-rules*] = *find-path0-autoref-aux*[*OF PREFER-id-D*]

**schematic-goal** *find-path0-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*::*?′c dres*),
*find-path0-restr-impl*
*G*
*P*
($R$:::$_r\langle$*Id*$\rangle$*vis-rel*)) $\in$ *?R*
$\langle$*proof*$\rangle$

**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*)
    ∈ ⟨*Rm*, *Rv*⟩*g-impl-rel-ext* → (*Rv* → *bool-rel*) → ⟨*Rv*⟩*vis-rel* →
    ⟨⟨⟨*Rv*⟩*vis-rel*, ⟨*Rv*⟩*list-rel* $\times_r$ *Rv*⟩*sum-rel*⟩*nres-rel*
  ⟨*proof*⟩
**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*×′*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*,*find-path1-restr G P R*)
  ∈ ⟨⟨⟨*Id*⟩*vis-rel*, ⟨*Id*⟩*list-rel* $\times_r$ *Id*⟩*sum-rel*⟩*nres-rel*
  ⟨*proof*⟩

**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*)∈*V* → ⟨*V*⟩*vis-rel* → ⟨*V*⟩*vis-rel*
        (*op-vis-memb*, (∈))∈*V* → ⟨*V*⟩*vis-rel* → *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*)
  ∈ ⟨*Rm*, *V*⟩*g-impl-rel-ext* → (*V* → *bool-rel*) → ⟨*V*⟩*vis-rel* →
    ⟨⟨⟨*V*⟩*vis-rel*, ⟨*V*⟩*list-rel* $\times_r$ *V*⟩*sum-rel*⟩*nres-rel*

⟨*proof*⟩

**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*) ∈ ⟨*Rm*, *V*⟩*g-impl-rel-ext*
    (*Pi*,*P*)∈*V* → *bool-rel*

**notes** [*autoref-tyrel*] = *TYRELI*[**where** $R = \langle(Id::('a\times'a::hashable)set)\rangle$*dflt-ahs-rel*]
**shows** (*nres-of ?c,find-path1 G P*)
$\in \langle\langle\langle V\rangle$*list-rel* $\times_r$ $V\rangle$*option-rel*$\rangle$*nres-rel*
$\langle proof \rangle$
**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** ($\lambda$ *G P. nres-of* (*find-path1-code G P*), *find-path1-spec*)
$\in \langle Rm, V\rangle$*g-impl-rel-ext* $\rightarrow$ ($V \rightarrow$ *bool-rel*) $\rightarrow \langle\langle\langle V\rangle$*list-rel* $\times_r$ $V\rangle$*option-rel*$\rangle$*nres-rel*
$\langle proof \rangle$

**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*: $(i,insert)\in R\rightarrow\langle R\rangle Rs\rightarrow\langle R\rangle Rs$
**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*⟩

**lemma** *obtain-finite-ahs*: *finite S* $\Longrightarrow \exists\, x.\ (x,S)\in\langle Id\rangle$ *dflt-ahs-rel*
⟨*proof*⟩

### 2.3.2 Framework Instantiation

**definition** *unit-parametrization* ≡ *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* ⟨*proof*⟩

**locale** *unit-DFS* = *param-DFS G unit-parametrization* **for** *G* :: (*'v*, *'more*) *graph-rec-scheme*
**begin**
**sublocale** *DFS G unit-parametrization*
⟨*proof*⟩
**end**

**lemma** *unit-DFSI*[*Pure.intro?*, *intro?*]:
**assumes** *fb-graph G*
**shows** *unit-DFS G*
⟨*proof*⟩

**definition** *find-reachable G* ≡ *do* {
*ASSERT* (*fb-graph G*);
*s* ← *unit-dfs.it-dfs TYPE*(*'a*) *G*;
*RETURN* (*dom* (*discovered s*))
}

**definition** *find-reachableT G* ≡ *do* {
  *ASSERT* (*fb-graph G*);
  *s* ← *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* ≤ *SPEC* (λ*r*. *r* = *reachable*)
    ⟨*proof*⟩

  **lemma** *find-reachableT-correct*:
    *finite reachable* ⟹ *find-reachableT G* ≤ *SPEC* (λ*r*. *r* = *reachable*)
    ⟨*proof*⟩
**end**

**context** *unit-DFS* **begin**

  **sublocale** *simple-impl G unit-parametrization unit-parametrization unit-rel*
    ⟨*proof*⟩

 **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* ⟨*proof*⟩

**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* $G$)
⟨*proof*⟩

**lemma** *find-reachable-implT-refine*:
*find-reachable-implT* $G \leq \Downarrow Id$ (*find-reachableT* $G$)
⟨*proof*⟩

**lemma** *find-reachable-rec-impl-refine*:
*find-reachable-rec-impl* $G \leq \Downarrow Id$ (*find-reachable* $G$)
⟨*proof*⟩

### 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*]:
($Gi$, $G$) ∈ ⟨$Rm$, $V$⟩*g-impl-rel-ext*
**notes** [*unfolded V-def*,*autoref-tyrel*] =
*TYRELI*[**where** $R$=⟨$V$⟩*dflt-ahs-rel*]
*TYRELI*[**where** $R$=⟨$V \times_r$ ⟨$V$⟩*list-set-rel*⟩*ras-rel*]
**shows** *nres-of* (*?c*::*?'c dres*) $\leq\Downarrow$*?R* (*find-reachable-impl* $G$)
⟨*proof*⟩
**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*: ($Gi$, $G$) ∈ ⟨$Rm$, $Id$⟩*g-impl-rel-ext*
**assumes** *4*: *find-reachable-code* $Gi$ = *dRETURN* $r$
**shows** ($r$, (*g-E* $G$)$^*$ ‘‘ *g-V0* $G$)∈⟨$Id$⟩*dflt-ahs-rel*
⟨*proof*⟩

**schematic-goal** *find-reachable-implT*:
**fixes** $V$ :: ($'vi\times'v$) *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*) $\leq\Downarrow$*?R* (*find-reachable-implT* $G$)
⟨*proof*⟩

**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** $V$ :: $('vi \times 'v)$ *set*
**assumes** $G$: *graph G*
**assumes** $FR$: *finite* $((g\text{-}E\ G)^* \text{ `` } g\text{-}V0\ G)$
**assumes** $BHC$: *is-bounded-hashcode V eq bhc*
**assumes** $EQ$: $(eq,(=)) \in V \to V \to bool\text{-}rel$
**assumes** $VDS$: *is-valid-def-hm-size TYPE* $('vi)$ *sz*
**assumes** *2*: $(Gi,\ G) \in \langle Rm,\ V\rangle g\text{-}impl\text{-}rel\text{-}ext$
**shows** (*find-reachable-codeT eq bhc sz Gi*, $(g\text{-}E\ G)^* \text{ `` } g\text{-}V0\ G) \in \langle V\rangle ahs\text{-}rel\ bhc$
⟨*proof*⟩

**definition** *all-unit-rel* :: $(unit \times 'a)$ *set* **where** *all-unit-rel* ≡ *UNIV*

**lemma** *all-unit-refine*[*simp*]:
$((),x) \in all\text{-}unit\text{-}rel$ ⟨*proof*⟩

**definition** *unit-list-rel* :: $('c \times 'a)\ set \Rightarrow (unit \times 'a\ list)\ set$
**where** [*to-relAPP*]: *unit-list-rel R* ≡ *UNIV*

**lemma** *unit-list-rel-refine*[*simp*]: $((),y) \in \langle R\rangle unit\text{-}list\text{-}rel$
⟨*proof*⟩

**lemmas** [*autoref-rel-intf*] = *REL-INTFI*[*of unit-list-rel i-list*]

**lemma** [*autoref-rules*]:
$((),[]) \in \langle R\rangle unit\text{-}list\text{-}rel$
$(\lambda\text{-}.\ (),tl) \in \langle R\rangle unit\text{-}list\text{-}rel \to \langle R\rangle unit\text{-}list\text{-}rel$
$(\lambda\text{-}\ \text{-}.\ (),(\#)) \in R \to \langle R\rangle unit\text{-}list\text{-}rel \to \langle R\rangle unit\text{-}list\text{-}rel$
⟨*proof*⟩

**schematic-goal** *find-reachable-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** $R = \langle V\rangle dflt\text{-}ahs\text{-}rel$]
**shows** *nres-of* $(?c::?'c\ dres) \le \Downarrow ?R$ (*find-reachable-rec-impl G*)
⟨*proof*⟩
**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*: $(Gi, G) \in \langle Rm, Id\rangle$*g-impl-rel-ext*
  **assumes** *4*: *find-reachable-rec-code Gi = dRETURN r*
  **shows** $(r, (g\text{-}E\ G)^* \text{``}\ g\text{-}V0\ G) \in \langle Id\rangle$*dflt-ahs-rel*
⟨*proof*⟩

**definition** [*simp*]: *op-reachable G* $\equiv (g\text{-}E\ G)^* \text{``}\ g\text{-}V0\ G$
**lemmas** [*autoref-op-pat*] = *op-reachable-def*[*symmetric*]

**context begin interpretation** *autoref-syn* ⟨*proof*⟩

**lemma** *autoref-op-reachable*[*autoref-rules*]:
  **fixes** *V* :: ('*vi*×'*v*) *set*
  **assumes** *G*: *SIDE-PRECOND* (*graph G*)
  **assumes** *FR*: *SIDE-PRECOND* (*finite* $((g\text{-}E\ G)^* \text{``}\ g\text{-}V0\ G)$)
  **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*: $(Gi, G) \in \langle Rm, V\rangle$*g-impl-rel-ext*
  **shows** (*find-reachable-codeT eq bhc sz Gi*,
    (*OP op-reachable* ::: $\langle Rm, V\rangle$*g-impl-rel-ext* $\to \langle V\rangle$*ahs-rel bhc*)$*G*)$\in\langle V\rangle$*ahs-rel*
*bhc*
  ⟨*proof*⟩

**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* ⇒ '*v rel* ⇒ *bool* **where**
  *is-fas G EC* $\equiv \neg(\exists\ u \in (g\text{-}E\ G)^* \text{``}\ g\text{-}V0\ G.\ (u, u) \in (g\text{-}E\ G - EC)^+)$

**lemma** *is-fas-alt*:

*is-fas G EC* = *acyclic* ((*g-E G* ∩ ((*g-E G*)* `` *g-V0 G* × *UNIV*) − *EC*))
⟨*proof*⟩

### 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*′
⟨*proof*⟩

**lemma** [*simp*]: *s*⦇ *state.more* := ⦇ *fas* = *foo* ⦈ ⦈ = *s* ⦇ *fas* := *foo* ⦈
⟨*proof*⟩

**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* ⟨*proof*⟩

Find feedback arc set

**definition** *find-fas G* ≡ *do* {
*ASSERT* (*graph G*);
*ASSERT* (*finite* ((*g-E G*)* `` *g-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-E G*)* `` *g-V0 G*)
**begin**

**sublocale** *DFS G fas-params*
⟨*proof*⟩

**end**

**lemma** *fasI*:
**assumes** *graph G*
**assumes** *finite* ((*g-E G*)* `` *g-V0 G*)
**shows** *fas G*

⟨*proof*⟩

### 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* ($\lambda s.$ *fas-state.fas s* = *back-edges s*)
⟨*proof*⟩
**lemmas** *fas-eq-back* = *i-fas-eq-back*[*THEN make-invar-thm*]

**lemma** *find-fas-correct-aux*:
**assumes** *NC*: $\neg$*cond s*
**shows** *is-fas G* (*fas-state.fas s*)
⟨*proof*⟩

**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*)
⟨*proof*⟩

### 2.4.3 Implementation

**record** *'v fas-state-impl* = *'v simple-state* +
*fas* :: (*'v*×*'v*) *set*

**definition** *fas-erel* ≡ {
((| *fas-state-impl.fas* = *f* |), (| *fas-state.fas* = *f*|)) | *f*. *True* }
**abbreviation** *fas-rel* ≡ ⟨*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*
≡ *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*)∈*fas-rel*
⟹ *fas-state-impl.fas si* = *fas-state.fas s*
⟨*proof*⟩

**interpretation** *fas-impl*: *simple-impl-defs G fas-params-impl fas-params*
**for** *G* ⟨*proof*⟩

**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-E G*)$^*$ `` *g-V0 G*));
  *s* $\leftarrow$ *fas-impl.tailrec-implT TYPE('a) G*;
  *RETURN* (*fas s*)
}

**context** *fas* **begin**

  **sublocale** *simple-impl G fas-params fas-params-impl fas-erel*
    ⟨*proof*⟩

  **lemmas** *impl-refine* = *simple-tailrec-refine simple-tailrecT-refine simple-rec-refine*
  **thm** *simple-refine*
**end**

**lemma** *find-fas-impl-refine*: *find-fas-impl G* $\leq$ ⇓*Id* (*find-fas G*)
  ⟨*proof*⟩

### 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* $\equiv$ {
  ((|*fas-impl* = *fi*, ... = *mi*|),(|*fas* = *f*, ... = *m*|)) | *fi mi f m*.
    (*fi*,*f*)∈*frel* ∧ (*mi*,*m*)∈*erel*}

**consts**
  *i-fas-state-ext* :: *interface* ⇒ *interface* ⇒ *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* $\equiv$ ⟨*ns-rel*,*vis-rel*,⟨*frel*,*erel*⟩*fas-state-erel*⟩*ss-impl-rel*
  **shows**
    (*fas-state-impl'-ext*, *fas-state-impl-ext*) ∈ *frel* → *erel* → ⟨*frel*,*erel*⟩*fas-state-erel*
    (*fas-impl*, *fas-state-impl.fas*) ∈ *R* → *frel*
    (*fas-state-impl'.fas-impl-update*, *fas-update*) ∈ (*frel* → *frel*) → *R* → *R*
  ⟨*proof*⟩

**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*)
  ⟨*proof*⟩
**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*⟩

**context begin interpretation** *autoref-syn* ⟨*proof*⟩

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* $\to RR$)\$$G)\in RR$
  ⟨*proof*⟩

**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*
  ⟨*proof*⟩

**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*⟩

**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* ($\lambda$*fas*. *is-fas G fas* $\wedge$ *FI* $\subseteq$ *fas*)
  ⟨*proof*⟩

**lemma** *gen-cast-set*[*autoref-rules-raw*]:
  **assumes** *PRIO-TAG-GEN-ALGO*
  **assumes** *INS*: *GEN-OP ins Set.insert* (*Rk*$\rightarrow$⟨*Rk*⟩*Rs2*$\rightarrow$⟨*Rk*⟩*Rs2*)
  **assumes** *EM*: *GEN-OP emp* {} (⟨*Rk*⟩*Rs2*)
  **assumes** *IT*: *SIDE-GEN-ALGO* (*is-set-to-list Rk Rs1 tsl*)
  **shows** ($\lambda$*s*. *gen-union* ($\lambda$*x*. *foldli* (*tsl x*)) *ins s emp*,*CAST*)
    $\in$ (⟨*Rk*⟩*Rs1*) $\rightarrow$ (⟨*Rk*⟩*Rs2*)
⟨*proof*⟩

**lemma** *gen-cast-fun-set-rel*[*autoref-rules-raw*]:
  **assumes** *INS*: *GEN-OP mem* ($\in$) (*Rk*$\rightarrow$⟨*Rk*⟩*Rs*$\rightarrow$*bool-rel*)
  **shows** ($\lambda$*s x*. *mem x s*,*CAST*) $\in$ (⟨*Rk*⟩*Rs*) $\rightarrow$ (⟨*Rk*⟩*fun-set-rel*)
⟨*proof*⟩

**lemma** *find-fas-init-impl-aux-unfolds*:
  *Let* ($E^*$``*V0*) = *Let* (*CAST* ($E^*$``*V0*))

$(\lambda S.\ RETURN\ (FI \cup S)) = (\lambda S.\ RETURN\ (FI \cup CAST\ S))$
⟨*proof*⟩

**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*]: $(eq,(=)) \in V \to V \to bool\text{-}rel$
**assumes** [*autoref-ga-rules*]: *is-valid-def-hm-size TYPE ('vi) sz*
**assumes** [*autoref-rules*]:
$(Gi,\ G) \in \langle Rm,\ V\rangle g\text{-}impl\text{-}rel\text{-}ext$
$(FIi,FI) \in \langle V \times_r V\rangle fun\text{-}set\text{-}rel$
**shows** $RETURN\ (?c::?'c) \leq \Downarrow ?R\ (find\text{-}fas\text{-}init\ G\ FI)$
⟨*proof*⟩

**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* ⟨*proof*⟩

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** $RR \equiv \langle V \times_r V\rangle fun\text{-}set\text{-}rel$
**assumes** *SIDE-GEN-ALGO (is-bounded-hashcode V eq bhc)*
**assumes** $GEN\text{-}OP\ eq\ (=)\ (V \to V \to bool\text{-}rel)$
**assumes** *SIDE-GEN-ALGO (is-valid-def-hm-size TYPE ('vi) sz)*
**shows** $(\lambda Gi\ FIi.\ RETURN\ (find\text{-}fas\text{-}init\text{-}code\ eq\ bhc\ sz\ Gi\ FIi), find\text{-}fas\text{-}init)$
$\in \langle Rm,\ V\rangle g\text{-}impl\text{-}rel\text{-}ext \to RR \to \langle RR\rangle nres\text{-}rel$
⟨*proof*⟩

**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*⟩

### 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'*
  ⟨*proof*⟩
**lemma** *red-more-cong*[*cong*]: *state.more s* = *state.more s'* ⟹ *red s* = *red s'*
  ⟨*proof*⟩

**lemma** [*simp*]: *s*⦇ *state.more* := ⦇ *lasso* = *foo*, *red* = *bar* ⦈ ⦈ = *s* ⦇ *lasso* := *foo*, *red* := *bar* ⦈
  ⟨*proof*⟩

**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'* ~~∪ 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) (λx. 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*
⟨*proof*⟩

**sublocale** *param-DFS-defs G blue-dfs-params*
⟨*proof*⟩

**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*⟩

**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* ⟷ *BlueDFS-invar G accpt s*
⟨*proof*⟩

**end**

### 2.5.3 Correctness Proof

**context** *BlueDFS* **begin**

**definition** *blue-basic-invar s* ≡
*case lasso s of*
*None* ⇒ *restr-invar E* (*red s*) (*λx. 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* $(G\ (\!|\ g\text{-}V0 := \{hd\ (stack\ s)\}\ |\!))$
**and** *fb-graph* $(G\ (\!|\ g\text{-}E := E \cap UNIV \times -\ red\ s,\ g\text{-}V0 := \{hd\ (stack\ s)\}\ |\!))$
**and** *restr-invar* $E\ (red\ s)\ (\lambda x.\ x \in set\ (stack\ s))$
⟨*proof*⟩

**lemma** (**in** *BlueDFS-invar*) *red-dfs-pres-bbi*:
**assumes** *BI*: *blue-basic-invar s*
**assumes** [*simp*]: $lasso\ s = None$ **and** *SNE*: $stack\ s \neq []$
**assumes** $pending\ s\ ``\ \{hd\ (stack\ s)\} = \{\}$
**shows** $run\text{-}red\text{-}dfs\ (hd\ (stack\ s))\ (finish\ (hd\ (stack\ s))\ s) \leq_n$
$SPEC\ (\lambda e.$
$DFS\text{-}invar\ G\ blue\text{-}dfs\text{-}params\ (finish\ (hd\ (stack\ s))\ s(\!|state.more := e|\!))$
$\longrightarrow blue\text{-}basic\text{-}invar\ (finish\ (hd\ (stack\ s))\ s(\!|state.more := e|\!)))$
⟨*proof*⟩

**lemma** *blue-basic-invar*: *is-invar blue-basic-invar*
⟨*proof*⟩

**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*
⟨*proof*⟩

**end**

**context** *BlueDFS-invar*
**begin**

**context assumes** [*simp*]: $lasso\ s = None$
**begin**
**lemma** *red-closed*:
$E\ ``\ red\ s \subseteq red\ s$
⟨*proof*⟩

**lemma** *red-stack-disjoint*:
$set\ (stack\ s) \cap red\ s = \{\}$
⟨*proof*⟩

**lemma** *red-finished*: $red\ s \subseteq dom\ (finished\ s)$
⟨*proof*⟩

**lemma** *all-nodes-colored*: $white\ s \cup blue\ s \cup cyan\ s \cup red\ s = UNIV$
⟨*proof*⟩

**lemma** *colors-disjoint*:
$white\ s \cap (blue\ s \cup cyan\ s \cup red\ s) = \{\}$
$blue\ s \cap (white\ s \cup cyan\ s \cup red\ s) = \{\}$
$cyan\ s \cap (white\ s \cup blue\ s \cup red\ s) = \{\}$
$red\ s \cap (white\ s \cup blue\ s \cup cyan\ s) = \{\}$
$\langle proof \rangle$

**end**

**lemma** (**in** *BlueDFS*) *i-no-accpt-cyle-in-finish*:
$is\text{-}invar\ (\lambda s.\ lasso\ s = None \longrightarrow (\forall x.\ accpt\ x \wedge x \in dom\ (finished\ s) \longrightarrow (x,x) \notin E^+))$
$\langle proof \rangle$

**lemma** *no-accpt-cycle-in-finish*:
$[\![lasso\ s = None;\ accpt\ v;\ v \in dom\ (finished\ s)]\!] \Longrightarrow (v,v) \notin E^+$
$\langle proof \rangle$

**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*: $lasso\text{-}inv\ s$
**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** $se\text{-}back\text{-}edge\ (hd\ (stack\ s))\ v\ s'$
$\leq SPEC\ (\lambda e.\ DFS\text{-}invar\ G\ blue\text{-}dfs\text{-}params\ (s'(\!|state.more := e|\!)) \longrightarrow$
$lasso\text{-}inv\ (s'(\!|state.more := e|\!)))$
$\langle proof \rangle$

**lemma** *lasso-inv*:
$is\text{-}invar\ lasso\text{-}inv$
$\langle proof \rangle$
**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*: $pl \neq []$
**and** *accpt-loop*: *accpt* (*hd pl*)
**and** *loop-is-path*: *path E* (*hd pl*) *pl* (*hd pl*)
**and** *loop-reachable*: $\exists\, v0 \in V0.$ *path E v0 pr* (*hd pl*)
⟨*proof*⟩

**lemma** *blue-dfs-correct*:
**assumes** *NC*: ¬ *cond s*
**shows** *case lasso s of*
$None \Rightarrow \neg(\exists\, v0 \in V0.\ \exists\, v.\ (v0,v) \in E^* \wedge accpt\ v \wedge (v,v) \in E^+)$
$|\ Some\ (pr,pl) \Rightarrow (\exists\, v0 \in V0.\ \exists\, v.$
$path\ E\ v0\ pr\ v \wedge accpt\ v \wedge pl \neq [] \wedge path\ E\ v\ pl\ v)$
⟨*proof*⟩

**end**

### 2.5.4 Interface

**interpretation** *BlueDFS-defs* **for** *G accpt* ⟨*proof*⟩

**definition** *nested-dfs-spec G accpt* ≡ λ*r*. *case r of*
$None \Rightarrow \neg(\exists\, v0 \in g\text{-}V0\ G.\ \exists\, v.\ (v0,v) \in (g\text{-}E\ G)^* \wedge accpt\ v \wedge (v,v) \in (g\text{-}E\ G)^+)$
$|\ Some\ (pr,pl) \Rightarrow (\exists\, v0 \in g\text{-}V0\ G.\ \exists\, v.$
$path\ (g\text{-}E\ G)\ v0\ pr\ v \wedge accpt\ v \wedge pl \neq [] \wedge path\ (g\text{-}E\ G)\ v\ pl\ v)$

**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*⟩

### 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*
  *⟨proof⟩*

**definition** *cyan-impl s ≡ on-stack s*
**lemma** *cyan-impl[refine]: ⟦(si,s)∈bdfs-rel⟧ ⟹ (cyan-impl si, cyan s)∈Id*
  *⟨proof⟩*

**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) (λx. 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)*
    *⟨proof⟩*

  **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* ⟧
⟹ *se-back-edge-impl accpt ui vi si* ≤⇓*bdfs-erel* (*se-back-edge accpt u v s*)
⟨*proof*⟩

**lemma** *NOOP-impl*: (*si*, *s*) ∈ *bdfs-rel*
⟹ *RETURN* (*simple-state.more si*) ≤ ⇓ *bdfs-erel* (*NOOP s*)
⟨*proof*⟩

**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* = *λv0 s. RETURN* (*simple-state.more s*),
*on-discover* = *λu v s. RETURN* (*simple-state.more s*),
*on-finish* = *λu 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* = *λu v s. RETURN* (*simple-state.more s*),
*is-break* = *λs. 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* ⟨*proof*⟩

**context** *BlueDFS* **begin**

**sublocale** *impl*: *simple-impl G blue-dfs-params bdfs-params-impl G accpt bdfs-erel*
⟨*proof*⟩

**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** $(Gi,G) \in Id$
  **assumes** $(accpti,accpt) \in Id$
  **shows** *nested-dfs-impl Gi accpti* $\leq \Downarrow(\langle\langle Id\rangle list\text{-}rel \times_r \langle Id\rangle list\text{-}rel\rangle option\text{-}rel)$
    (*nested-dfs G accpt*)
  ⟨*proof*⟩

### 2.5.6 Synthesis of Executable Code

**record** $('v,'si,'nsi)bdfs\text{-}state\text{-}impl' = ('si,'nsi)simple\text{-}state\text{-}impl$ +
  $lasso\text{-}impl' :: ('v\ list \times 'v\ list)\ option$
  $red\text{-}impl' :: 'nsi$

**definition** [*to-relAPP*]: $bdfs\text{-}state\text{-}erel'\ Vi \equiv \{$
  $((\!| lasso\text{-}impl' = li,\ red\text{-}impl'=ri |\!), (\!| lasso\text{-}impl = l,\ red\text{-}impl = r |\!)) \mid li\ ri\ l\ r.$
    $(li,l) \in \langle\langle Vi\rangle list\text{-}rel \times_r \langle Vi\rangle list\text{-}rel\rangle option\text{-}rel \wedge (ri,r) \in \langle Vi\rangle dflt\text{-}ahs\text{-}rel\}$

**consts**
  $i\text{-}bdfs\text{-}state\text{-}ext :: interface \Rightarrow interface$

**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-state-impl′-ext*, *bdfs-state-impl-ext*)
      $\in \langle\langle Vi\rangle list\text{-}rel \times_r \langle Vi\rangle list\text{-}rel\rangle option\text{-}rel \rightarrow \langle Vi\rangle dflt\text{-}ahs\text{-}rel \rightarrow unit\text{-}rel \rightarrow$
$\langle Vi\rangle bdfs\text{-}state\text{-}erel'$
    $(lasso\text{-}impl',\ lasso\text{-}impl) \in R \rightarrow \langle\langle Vi\rangle list\text{-}rel \times_r \langle Vi\rangle list\text{-}rel\rangle option\text{-}rel$
    $(red\text{-}impl',\ red\text{-}impl) \in R \rightarrow \langle Vi\rangle dflt\text{-}ahs\text{-}rel$
  ⟨*proof*⟩

**schematic-goal** *nested-dfs-code*:
  **assumes** *Vid*: $V = (Id :: ('v::hashable \times 'v)\ set)$
  **assumes** [*unfolded Vid*, *autoref-rules*]:
    $(Gi,\ G) \in \langle Rm,\ V\rangle g\text{-}impl\text{-}rel\text{-}ext$
    $(accpti,accpt) \in (V \rightarrow bool\text{-}rel)$
  **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*)
    $\in \langle\langle\langle V\rangle list\text{-}rel \times_r \langle V\rangle list\text{-}rel\rangle option\text{-}rel\rangle nres\text{-}rel$
  ⟨*proof*⟩

**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** $(\lambda\ G\ accpt.\ nres\text{-}of\ (nested\text{-}dfs\text{-}code\ G\ accpt), nested\text{-}dfs) \in$
    $\langle Rm,\ V\rangle g\text{-}impl\text{-}rel\text{-}ext \to (V \to bool\text{-}rel) \to$
    $\langle\langle\langle V\rangle list\text{-}rel \times_r \langle V\rangle list\text{-}rel\rangle option\text{-}rel\rangle nres\text{-}rel$
$\langle proof\rangle$

**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* $\langle proof\rangle$
  **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-path s v p w*
  **and** $w \in dom$ (*discovered s*)
  **and** $(v,w) \in E^*$ $(w,v) \in E^*$
  **shows** $w \in$ *lowlink-set s v*
⟨*proof*⟩

**lemma** *lowlink-set-discovered*:
  *lowlink-set s v* $\subseteq$ *dom* (*discovered s*)
  ⟨*proof*⟩

**lemma** *lowlink-set-finite*[*simp*, *intro*!]:
  *finite* (*lowlink-set s v*)
  ⟨*proof*⟩

**lemma** *lowlink-set-not-empty*:
  **assumes** $v \in dom$ (*discovered s*)
  **shows** *lowlink-set s v* $\neq \{\}$
  ⟨*proof*⟩

**lemma** *lowlink-path-single*:
  **assumes** $(v,w) \in$ *cross-edges s* $\cup$ *back-edges s*
  **shows** *lowlink-path s v* [*v*] *w*
  ⟨*proof*⟩

**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*⟩

**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*⟩

**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*⟩

**lemma** *lowlink-path-tree-prepend*:
  **assumes** *p*: *lowlink-path s v p w*
  **and** *tree-edges*: $(u,v) \in$ (*tree-edges s*)$^+$
  **and** *fin*: $u \in dom$ (*finished s*) $\vee$ (*stack s* $\neq [] \wedge u = hd$ (*stack s*))

**shows** $\exists\, p.$ *lowlink-path s u p w*
⟨*proof*⟩

**lemma** *lowlink-path-complex*:
**assumes** $(u,v) \in (\textit{tree-edges } s)^+$
**and** $u \in \textit{dom } (\textit{finished } s) \vee (\textit{stack } s \neq [] \wedge u = \textit{hd } (\textit{stack } s))$
**and** $(v,w) \in \textit{cross-edges } s \cup \textit{back-edges } s$
**shows** $\exists\, p.$ *lowlink-path s u p w*
⟨*proof*⟩

**lemma** *no-path-imp-no-lowlink-path*:
**assumes** *edges s* ‘‘ $\{v\} = \{\}$
**shows** $\neg$*lowlink-path s v p w*
⟨*proof*⟩

**context begin interpretation** *timing-syntax* ⟨*proof*⟩

**lemma** *LowLink-le-disc*:
**assumes** $v \in \textit{dom } (\textit{discovered } s)$
**shows** $\textit{LowLink } s\ v \leq \delta\ s\ v$
⟨*proof*⟩

**lemma** *LowLink-lessE*:
**assumes** $\textit{LowLink } s\ v < x$
**and** $v \in \textit{dom } (\textit{discovered } s)$
**obtains** $w$ **where** $\delta\ s\ w < x$ $w \in \textit{lowlink-set } s\ v$
⟨*proof*⟩

**lemma** *LowLink-lessI*:
**assumes** $y \in \textit{lowlink-set } s\ v$
**and** $\delta\ s\ y < \delta\ s\ v$
**shows** $\textit{LowLink } s\ v < \delta\ s\ v$
⟨*proof*⟩

**lemma** *LowLink-eqI*:
**assumes** *DFS-invar G param s′*
**assumes** *sub-m*: $\textit{discovered } s \subseteq_m \textit{discovered } s'$
**assumes** *sub*: $\textit{lowlink-set } s\ w \subseteq \textit{lowlink-set } s'\ w$
**and** *rev-sub*: $\textit{lowlink-set } s'\ w \subseteq \textit{lowlink-set } s\ w \cup X$
**and** *w-disc*: $w \in \textit{dom } (\textit{discovered } s)$
**and** *X*: $\bigwedge x.\ [\![x \in X;\ x \in \textit{lowlink-set } s'\ w]\!] \Longrightarrow \delta\ s'\ x \geq \textit{LowLink } s\ w$
**shows** $\textit{LowLink } s\ w = \textit{LowLink } s'\ w$
⟨*proof*⟩

**lemma** *LowLink-eq-disc-iff-scc-root*:
**assumes** $v \in \textit{dom } (\textit{finished } s) \vee (\textit{stack } s \neq [] \wedge v = \textit{hd } (\textit{stack } s) \wedge \textit{pending } s$ ‘‘ $\{v\} = \{\})$
**shows** $\textit{LowLink } s\ v = \delta\ s\ v \longleftrightarrow \textit{scc-root } s\ v\ (\textit{scc-of } E\ v)$

⟨*proof*⟩
**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*⟩

### 2.7.2 Instantiation of the DFS-Framework

**record** $'v\ tarjan\text{-}state = 'v\ state\ +$
$sccs :: 'v\ set\ set$
$lowlink :: 'v \rightharpoonup nat$
$tj\text{-}stack :: 'v\ list$

**type-synonym** $'v\ tarjan\text{-}param = ('v, ('v,unit)\ tarjan\text{-}state\text{-}ext)\ parameterization$

**abbreviation** $the\text{-}lowlink\ s\ v \equiv the\ (lowlink\ s\ v)$

**context** *timing-syntax*
**begin**
**notation** *the-lowlink* ($\zeta$)
**end**

**locale** *Tarjan-def* = *graph-defs G*
**for** $G :: ('v, 'more)\ graph\text{-}rec\text{-}scheme$
**begin**
**context begin interpretation** *timing-syntax* ⟨*proof*⟩

**definition** $tarjan\text{-}disc :: 'v \Rightarrow 'v\ tarjan\text{-}state \Rightarrow ('v,unit)\ tarjan\text{-}state\text{-}ext\ nres$
**where**
$tarjan\text{-}disc\ v\ s = RETURN\ (\!|\ sccs = sccs\ s,$

$lowlink = (lowlink\ s)(v \mapsto \delta\ s\ v),$
$tj\text{-}stack = v\#tj\text{-}stack\ s\|)$

**definition** *tj-stack-pop* :: $'v\ list \Rightarrow 'v \Rightarrow ('v\ list \times 'v\ set)\ nres$ **where**
$tj\text{-}stack\text{-}pop\ tjs\ u = RETURN\ (tl\ (dropWhile\ ((\neq)\ u)\ tjs),\ insert\ u\ (set\ (takeWhile\ ((\neq)\ u)\ tjs)))$

**lemma** *tj-stack-pop-set*:
$tj\text{-}stack\text{-}pop\ tjs\ u \leq SPEC\ (\lambda(tjs',scc).\ u \in set\ tjs \longrightarrow set\ tjs = set\ tjs' \cup scc \wedge u \in scc)$
⟨*proof*⟩

**lemmas** *tj-stack-pop-set-leof-rule* = *weaken-SPEC*[*OF tj-stack-pop-set*, *THEN leof-lift*]

**definition** *tarjan-fin* :: $'v \Rightarrow 'v\ tarjan\text{-}state \Rightarrow ('v,unit)\ tarjan\text{-}state\text{-}ext\ nres$ **where**
$tarjan\text{-}fin\ v\ s = do\ \{$
$\quad let\ ll = (if\ stack\ s = []\ then\ lowlink\ s$
$\qquad else\ let\ u = hd\ (stack\ s)\ in$
$\qquad\quad (lowlink\ s)(u \mapsto min\ (\zeta\ s\ u)\ (\zeta\ s\ v)));$
$\quad let\ s' = s(\!|\ lowlink := ll\ |\!);$

$\quad ASSERT\ (v \in set\ (tj\text{-}stack\ s));$
$\quad ASSERT\ (distinct\ (tj\text{-}stack\ s));$
$\quad if\ \zeta\ s\ v = \delta\ s\ v\ then\ do\ \{$
$\qquad ASSERT\ (scc\text{-}root'\ E\ s\ v\ (scc\text{-}of\ E\ v));$
$\qquad (tjs,scc) \leftarrow tj\text{-}stack\text{-}pop\ (tj\text{-}stack\ s)\ v;$
$\qquad RETURN\ (state.more\ (s'(\!|\ tj\text{-}stack := tjs,\ sccs := insert\ scc\ (sccs\ s)|\!)))$
$\quad\}\ else\ do\ \{$
$\qquad ASSERT\ (\neg\ scc\text{-}root'\ E\ s\ v\ (scc\text{-}of\ E\ v));$
$\qquad RETURN\ (state.more\ s')$
$\quad\}\}$

**definition** *tarjan-back* :: $'v \Rightarrow 'v \Rightarrow 'v\ tarjan\text{-}state \Rightarrow ('v,unit)\ tarjan\text{-}state\text{-}ext\ nres$ **where**
$tarjan\text{-}back\ u\ v\ s = ($
$\quad if\ \delta\ s\ v < \delta\ s\ u \wedge v \in set\ (tj\text{-}stack\ s)\ then$
$\qquad let\ ul' = min\ (\zeta\ s\ u)\ (\delta\ s\ v)$
$\qquad in\ RETURN\ (state.more\ (s(\!|\ lowlink := (lowlink\ s)(u \mapsto ul')\ |\!)))$
$\quad else\ NOOP\ s)$

**end**

**definition** *tarjan-params* :: $'v\ tarjan\text{-}param$ **where**
$tarjan\text{-}params = (\!|$
$\quad on\text{-}init = RETURN\ (\!|\ sccs = \{\},\ lowlink = Map.empty,\ tj\text{-}stack = []\ |\!),$
$\quad on\text{-}new\text{-}root = tarjan\text{-}disc,$
$\quad on\text{-}discover = \lambda u.\ tarjan\text{-}disc,$
$\quad on\text{-}finish = tarjan\text{-}fin,$

*on-back-edge* = *tarjan-back*,
*on-cross-edge* = *tarjan-back*,
*is-break* = $\lambda s$. *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*
⟨*proof*⟩

**sublocale** *param-DFS-defs G tarjan-params* ⟨*proof*⟩
**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*
⟨*proof*⟩

**lemma** *sccs-more-cong*[*cong*]:*state.more s* = *state.more s'* $\Longrightarrow$ *sccs s* = *sccs s'*
⟨*proof*⟩
**lemma** *lowlink-more-cong*[*cong*]:*state.more s* = *state.more s'* $\Longrightarrow$ *lowlink s* = *lowlink s'*
⟨*proof*⟩
**lemma** *tj-stack-more-cong*[*cong*]:*state.more s* = *state.more s'* $\Longrightarrow$ *tj-stack s* = *tj-stack s'*
⟨*proof*⟩

**lemma** [*simp*]:
*s*(| *state.more* := (|*sccs* = *sc*, *lowlink* = *l*, *tj-stack* = *t*|)|)
= *s*(| *sccs* := *sc*, *lowlink* := *l*, *tj-stack* := *t*|)
⟨*proof*⟩
**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* $\longleftrightarrow$ *Tarjan-invar G s* (**is** *?D* $\longleftrightarrow$ *?T*)

⟨*proof*⟩
**end**

### 2.7.3 Correctness Proof

**context** *Tarjan* **begin**
**lemma** *i-tj-stack-discovered*:
*is-invar* ($\lambda s.$ *set* (*tj-stack s*) $\subseteq$ *dom* (*discovered s*))
⟨*proof*⟩

**lemmas** (**in** *Tarjan-invar*) *tj-stack-discovered* =
*i-tj-stack-discovered*[*THEN make-invar-thm*]

**lemma** *i-tj-stack-distinct*:
*is-invar* ($\lambda s.$ *distinct* (*tj-stack s*))
⟨*proof*⟩

**lemmas** (**in** *Tarjan-invar*) *tj-stack-distinct* =
*i-tj-stack-distinct*[*THEN make-invar-thm*]

**context begin interpretation** *timing-syntax* ⟨*proof*⟩
**lemma** *i-tj-stack-incr-disc*:
*is-invar* ($\lambda s.$ $\forall$ *k*<*length* (*tj-stack s*). $\forall$ *j*<*k*. $\delta$ *s* (*tj-stack s* ! *j*) > $\delta$ *s* (*tj-stack s*
! *k*))
⟨*proof*⟩
**end end**

**context** *Tarjan-invar* **begin context begin interpretation** *timing-syntax* ⟨*proof*⟩
**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*)
⟨*proof*⟩

**lemma** *tjs-disc-dw-tw*:
**fixes** *u*
**defines** *dw* $\equiv$ *dropWhile* (($\neq$) *u*) (*tj-stack s*)
**defines** *tw* $\equiv$ *takeWhile* (($\neq$) *u*) (*tj-stack s*)
**assumes** $x \in$ *set dw* $y \in$ *set tw*
**shows** $\delta\ s\ x < \delta\ s\ y$
⟨*proof*⟩
**end end**

**context** *Tarjan* **begin context begin interpretation** *timing-syntax* ⟨*proof*⟩
**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-stack s*))
⟨*proof*⟩

**lemma** *i-tj-stack-ss-stack-finished*:

*is-invar* ($\lambda s.$ *set* (*tj-stack* *s*) $\subseteq$ *set* (*stack* *s*) $\cup$ *dom* (*finished* *s*))
⟨*proof*⟩

**lemma** *i-finished-ss-sccs-tj-stack*:
*is-invar* ($\lambda s.$ *dom* (*finished* *s*) $\subseteq \bigcup$(*sccs* *s*) $\cup$ *set* (*tj-stack* *s*))
⟨*proof*⟩
**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*)
⟨*proof*⟩

**lemma** *stack-ss-tj-stack*:
*set* (*stack* *s*) $\subseteq$ *set* (*tj-stack* *s*)
⟨*proof*⟩

**lemma** *hd-stack-in-tj-stack*:
*stack* *s* $\neq$ [] $\Longrightarrow$ *hd* (*stack* *s*) $\in$ *set* (*tj-stack* *s*)
⟨*proof*⟩
**end**

**context** *Tarjan* **begin context begin interpretation** *timing-syntax* ⟨*proof*⟩
**lemma** *i-no-finished-root*:
*is-invar* ($\lambda s.$ *scc-root* *s* *r* *scc* $\wedge$ *r* $\in$ *dom* (*finished* *s*) $\longrightarrow$ ($\forall x \in$ *scc*. $x \notin$ *set* (*tj-stack* *s*)))
⟨*proof*⟩
**end end**

**context** *Tarjan-invar* **begin**
**lemma** *no-finished-root*:
**assumes** *scc-root* *s* *r* *scc*
**and** *r* $\in$ *dom* (*finished* *s*)
**and** $x \in$ *scc*
**shows** $x \notin$ *set* (*tj-stack* *s*)
⟨*proof*⟩

**context begin interpretation** *timing-syntax* ⟨*proof*⟩

**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*⟩

**lemma** *tj-stack-reach-hd-stack*:
**assumes** $v \in set\ (tj\text{-}stack\ s)$
**shows** $(v,\ hd\ (stack\ s)) \in E^*$
⟨*proof*⟩

**lemma** *empty-stack-imp-empty-tj-stack*:
**assumes** $stack\ s = []$
**shows** $tj\text{-}stack\ s = []$
⟨*proof*⟩

**lemma** *stacks-eq-iff*: $stack\ s = [] \longleftrightarrow tj\text{-}stack\ s = []$
⟨*proof*⟩
**end end**

**context** *Tarjan* **begin context begin interpretation** *timing-syntax* ⟨*proof*⟩
**lemma** *i-sccs-are-sccs*:
$is\text{-}invar\ (\lambda s.\ \forall scc \in sccs\ s.\ is\text{-}scc\ E\ scc)$
⟨*proof*⟩
**end**

**lemmas** (**in** *Tarjan-invar*) *sccs-are-sccs* =
*i-sccs-are-sccs*[*THEN make-invar-thm*]

**context begin interpretation** *timing-syntax* ⟨*proof*⟩

**lemma** *i-lowlink-eq-LowLink*:
$is\text{-}invar\ (\lambda s.\ \forall x \in dom\ (discovered\ s).\ \zeta\ s\ x = LowLink\ s\ x)$
⟨*proof*⟩
**end end**

**context** *Tarjan-invar* **begin context begin interpretation** *timing-syntax* ⟨*proof*⟩

**lemmas** *lowlink-eq-LowLink* =
*i-lowlink-eq-LowLink*[*THEN make-invar-thm*, *rule-format*]

**lemma** *lowlink-eq-disc-iff-scc-root*:
**assumes** $v \in dom\ (finished\ s) \lor (stack\ s \neq [] \land v = hd\ (stack\ s) \land pending\ s$
`` $\{v\} = \{\})$
**shows** $\zeta\ s\ v = \delta\ s\ v \longleftrightarrow scc\text{-}root\ s\ v\ (scc\text{-}of\ E\ v)$
⟨*proof*⟩

**lemma** *nc-sccs-eq-reachable*:
**assumes** *NC*: $\neg\ cond\ s$
**shows** $reachable = \bigcup (sccs\ s)$
⟨*proof*⟩
**end end**

**context** *Tarjan* **begin**

**lemma** *tarjan-fin-nofail*:
  **assumes** *pre-on-finish u s'*
  **shows** *nofail* (*tarjan-fin u s'*)
⟨*proof*⟩

**sublocale** *DFS G tarjan-params*
  ⟨*proof*⟩
**end**

**interpretation** *tarjan*: *Tarjan-def* **for** *G* ⟨*proof*⟩

### 2.7.4 Interface

**definition** *tarjan G* ≡ *do* {
  *ASSERT* (*fb-graph G*);
  *s* ← *tarjan.it-dfs TYPE('a) G*;
  *RETURN* (*sccs s*) }

**definition** *tarjan-spec G* ≡ *do* {
  *ASSERT* (*fb-graph G*);
  *SPEC* ($\lambda$*sccs*. ($\forall$ *scc* ∈ *sccs*. *is-scc* (*g-E G*) *scc*)
            ∧ $\bigcup$ *sccs* = *tarjan.reachable TYPE('a) G*)}

**lemma** *tarjan-correct*:
  *tarjan G* ≤ *tarjan-spec G*
⟨*proof*⟩

**end**