

A Framework for Verifying Depth-First Search Algorithms

Peter Lammich and René Neumann

April 10, 2026

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.

Contents

1	The DFS Framework	2
1.1	General DFS with Hooks	2
1.1.1	State and Parameterization	2
1.1.2	DFS operations	3
1.1.3	DFS Algorithm	8
1.1.4	Invariants	9
1.1.5	Basic Invariants	16
1.1.6	Total Correctness	20
1.1.7	Non-Failing Parameterization	20
1.2	Basic Invariant Library	22
1.2.1	Basic Timing Invariants	22
1.2.2	Paranthesis Theorem	24
1.2.3	Edge Types	25
1.2.4	White Path Theorem	34
1.3	Invariants for SCCs	35
1.4	Generic DFS and Refinement	38
1.4.1	Generic DFS Algorithm	38
1.4.2	Refinement Between DFS Implementations	43
1.5	Tail-Recursive Implementation	47
1.6	Recursive DFS Implementation	51
1.7	Simple Data Structures	57
1.7.1	Stack, Pending Stack, and Visited Set	57
1.7.2	Simple state without on-stack	63
1.7.3	Simple state without stack and on-stack	64
1.8	Restricting Nodes by Pre-Initializing Visited Set	66
1.9	Basic DFS Framework	70
2	Examples	72
2.1	Simple Cyclicity Checker	72
2.1.1	Framework Instantiation	72
2.1.2	Correctness Proof	74
2.1.3	Implementation	75
2.1.4	Synthesizing Executable Code	77

2.2	Finding a Path between Nodes	79
2.2.1	Including empty Path	80
2.2.2	Restricting the Graph	83
2.2.3	Path of Minimal Length One, with Restriction	84
2.2.4	Path of Minimal Length One, without Restriction	85
2.2.5	Implementation	85
2.2.6	Synthesis of Executable Code	87
2.2.7	Conclusion	90
2.3	Set of Reachable Nodes	90
2.3.1	Preliminaries	91
2.3.2	Framework Instantiation	91
2.3.3	Correctness	92
2.3.4	Synthesis of Executable Implementation	93
2.3.5	Conclusions	95
2.4	Find a Feedback Arc Set	95
2.4.1	Instantiation of the DFS-Framework	96
2.4.2	Correctness Proof	97
2.4.3	Implementation	97
2.4.4	Synthesis of Executable Code	98
2.4.5	Feedback Arc Set with Initialization	99
2.4.6	Conclusion	101
2.5	Nested DFS	102
2.5.1	Auxiliary Lemmas	102
2.5.2	Instantiation of the Framework	102
2.5.3	Correctness Proof	104
2.5.4	Interface	107
2.5.5	Implementation	107
2.5.6	Synthesis of Executable Code	110
2.5.7	Conclusion	111
2.6	Invariants for Tarjan's Algorithm	111
2.7	Tarjan's Algorithm	114
2.7.1	Preliminaries	114
2.7.2	Instantiation of the DFS-Framework	114
2.7.3	Correctness Proof	117
2.7.4	Interface	120

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  $\rightarrow$  nat — Discovered times of nodes
  finished :: 'v  $\rightarrow$  nat   — Finished times of nodes
  pending :: ('v  $\times$  '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 $s \equiv RETURN (state.more\ s)$

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 \wedge *more init. dflt-parametrization more init* \equiv (
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* \equiv $u \in \text{dom } (\text{discovered } s)$

definition *is-finished* :: *'v* ⇒ (*'v, 'es*) *state-scheme* ⇒ *bool*
where *is-finished u s* \equiv $u \in \text{dom } (\text{finished } s)$

definition *is-empty-stack* :: (*'v, 'es*) *state-scheme* ⇒ *bool*
where *is-empty-stack s* \equiv $\text{stack } s = []$

Effects on Basic State

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

definition *discover*

$:: 'v \Rightarrow 'v \Rightarrow ('v, 'es) \text{ state-scheme} \Rightarrow ('v, 'es) \text{ state-scheme}$

where

$discover\ u\ v\ s \equiv let$

$d = (discovered\ s)(v \mapsto counter\ s); c = counter\ s + 1;$

$st = v\#stack\ s;$

$p = pending\ s \cup \{v\} \times E''\{v\};$

$t = insert\ (u, v)\ (tree-edges\ s)$

$in\ s(| discovered := d, counter := c, stack := st, pending := p, tree-edges := t|)$

lemma *discover-simps[simp]*:

$counter\ (discover\ u\ v\ s) = Suc\ (counter\ s)$

$discovered\ (discover\ u\ v\ s) = (discovered\ s)(v \mapsto counter\ s)$

$finished\ (discover\ u\ v\ s) = finished\ s$

$stack\ (discover\ u\ v\ s) = v\#stack\ s$

$pending\ (discover\ u\ v\ s) = pending\ s \cup \{v\} \times E''\{v\}$

$tree-edges\ (discover\ u\ v\ s) = insert\ (u, v)\ (tree-edges\ s)$

$cross-edges\ (discover\ u\ v\ s) = cross-edges\ s$

$back-edges\ (discover\ u\ v\ s) = back-edges\ s$

$state.more\ (discover\ u\ v\ s) = state.more\ s$

$\langle proof \rangle$

definition *finish*

$:: 'v \Rightarrow ('v, 'es) \text{ state-scheme} \Rightarrow ('v, 'es) \text{ state-scheme}$

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]*:

$counter\ (finish\ u\ s) = Suc\ (counter\ s)$

$discovered\ (finish\ u\ s) = discovered\ s$

$finished\ (finish\ u\ s) = (finished\ s)(u \mapsto counter\ s)$

$stack\ (finish\ u\ s) = tl\ (stack\ s)$

$pending\ (finish\ u\ s) = pending\ s$

$tree-edges\ (finish\ u\ s) = tree-edges\ s$

$cross-edges\ (finish\ u\ s) = cross-edges\ s$

$back-edges\ (finish\ u\ s) = back-edges\ s$

$state.more\ (finish\ u\ s) = state.more\ s$

$\langle proof \rangle$

definition *back-edge*

$:: 'v \Rightarrow 'v \Rightarrow ('v, 'es) \text{ state-scheme} \Rightarrow ('v, 'es) \text{ state-scheme}$

where

$back-edge\ u\ v\ s \equiv let$

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

lemma *back-edge-simps*[simp]:

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

definition *cross-edge*

$:: 'v \Rightarrow 'v \Rightarrow ('v, 'es) \text{ state-scheme} \Rightarrow ('v, 'es) \text{ state-scheme}$

where

$\text{cross-edge } u \ v \ s \equiv \text{let}$
 $c = \text{insert } (u,v) \text{ (cross-edges } s)$
 $\text{in } s(\text{ cross-edges } := c)$

lemma *cross-edge-simps*[simp]:

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

definition *new-root*

$:: 'v \Rightarrow ('v, 'es) \text{ state-scheme} \Rightarrow ('v, 'es) \text{ state-scheme}$

where

$\text{new-root } v0 \ s \equiv \text{let}$
 $c = \text{Suc } (\text{counter } s);$
 $d = (\text{discovered } s)(v0 \mapsto \text{counter } s);$
 $p = \{v0\} \times E^{\{v0\}};$
 $st = [v0]$
 $\text{in } s(\text{counter } := c, \text{discovered } := d, \text{pending } := p, \text{stack } := st)$

lemma *new-root-simps*[simp]:

$\text{counter } (\text{new-root } v0 \ s) = \text{Suc } (\text{counter } s)$
 $\text{discovered } (\text{new-root } v0 \ s) = (\text{discovered } s)(v0 \mapsto \text{counter } s)$
 $\text{finished } (\text{new-root } v0 \ s) = \text{finished } s$

$stack (new-root\ v0\ s) = [v0]$
 $pending (new-root\ v0\ s) = (\{v0\} \times E^{\{v0\}})$
 $tree-edges (new-root\ v0\ s) = tree-edges\ s$
 $cross-edges (new-root\ v0\ s) = cross-edges\ s$
 $back-edges (new-root\ v0\ s) = back-edges\ s$
 $state.more (new-root\ v0\ s) = state.more\ s$
 <proof>

definition *empty-state e*

$\equiv (\mid counter = 0,$
 $discovered = Map.empty,$
 $finished = Map.empty,$
 $pending = \{\},$
 $stack = [],$
 $tree-edges = \{\},$
 $back-edges = \{\},$
 $cross-edges = \{\},$
 $\dots = e \mid)$

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*

$:: 'v \Rightarrow 'v \Rightarrow ('v, 'es)\ state-scheme \Rightarrow ('v, 'es)\ state-scheme\ nres$

where

$do-cross-edge\ u\ v\ s \equiv do\ \{$
 $let\ s = cross-edge\ u\ v\ s;$
 $e \leftarrow on-cross-edge\ param\ u\ v\ s;$
 $RETURN\ (s\{state.more := e\})$
 $\}$

definition *do-back-edge*

$:: 'v \Rightarrow 'v \Rightarrow ('v, 'es)\ state-scheme \Rightarrow ('v, 'es)\ state-scheme\ nres$

where

$do-back-edge\ u\ v\ s \equiv do\ \{$
 $let\ s = back-edge\ u\ v\ s;$

```

    e ← on-back-edge param u v s;
    RETURN (s(state.more := e))
}

```

definition *do-known-edge*

```

:: 'v ⇒ 'v ⇒ ('v, 'es) state-scheme ⇒ ('v, 'es) state-scheme nres

```

where

```

do-known-edge u v s ≡
  if is-finished v s then
    do-cross-edge u v s
  else
    do-back-edge u v s

```

definition *do-discover*

```

:: 'v ⇒ 'v ⇒ ('v, 'es) state-scheme ⇒ ('v, 'es) state-scheme nres

```

where

```

do-discover u v s ≡ do {
  let s = discover u v s;
  e ← on-discover param u v s;
  RETURN (s(state.more := e))
}

```

definition *do-finish*

```

:: 'v ⇒ ('v, 'es) state-scheme ⇒ ('v, 'es) state-scheme nres

```

where

```

do-finish u s ≡ do {
  let s = finish u s;
  e ← on-finish param u s;
  RETURN (s(state.more := e))
}

```

definition *get-new-root where*

```

get-new-root s ≡ SPEC (λv. v ∈ V0 ∧ ¬is-discovered v s)

```

definition *do-new-root where*

```

do-new-root v0 s ≡ do {
  let s = new-root v0 s;
  e ← 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* ≡ do {

```

  e ← 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 ($V0 \subseteq \{v. \text{is-discovered } v \text{ } s\} \longrightarrow \neg \text{is-empty-stack } s$)
 $\wedge \neg \text{is-break param } s$

lemma *cond-alt*:

cond = ($\lambda s. (V0 \subseteq \text{dom } (\text{discovered } s) \longrightarrow \text{stack } s \neq [])$) $\wedge \neg \text{is-break param } s$
 <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 \leftarrow RES *Vs*;

let *s* = *s* | *pending* := *pending* *s* - {(*u*,*v*)};

RETURN (*u*,Some *v*,*s*)

}

}

definition *step* :: ('v,'es) state-scheme \Rightarrow ('v,'es) state-scheme nres

where

step *s* \equiv

if *is-empty-stack* *s* then do {

v0 \leftarrow *get-new-root* *s*;

do-new-root *v0* *s*

} else do {

(*u*,*Vs*,*s*) \leftarrow *get-pending* *s*;

case *Vs* of

None \Rightarrow *do-finish* *u* *s*

| Some *v* \Rightarrow do {

if *is-discovered* *v* *s* then

do-known-edge *u* *v* *s*

else

```

    }
    }

```

definition $it\text{-}dfs \equiv init \gg \text{WHILE } cond \text{ step}$

definition $it\text{-}dfsT \equiv init \gg \text{WHILET } cond \text{ step}$

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 $G :: ('v, 'more) \text{ graph-rec-scheme}$
and $param :: ('v, 'es) \text{ parameterization}$
begin

definition $is\text{-}invar :: (('v, 'es) \text{ state-scheme} \Rightarrow bool) \Rightarrow bool$

— Predicate that states that I is an invariant.

where $is\text{-}invar I \equiv is\text{-}rwof\text{-}invar \text{ init } cond \text{ step } I$

end

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

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

lemma *make-invar-thm*: $is\text{-}invar I \Longrightarrow I s$

— Lemma to transfer an invariant into this locale

$\langle proof \rangle$

end

Establishing Invariants

context *param-DFS*
begin

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

lemmas *indep-invar-rules* =
leaf-True-rule[**where** *m=on-init param*]
leaf-True-rule[**where** *m=on-new-root param v0 s' for v0 s'*]
leaf-True-rule[**where** *m=on-discover param u v s' for u v s'*]
leaf-True-rule[**where** *m=on-finish param v s' for v s'*]
leaf-True-rule[**where** *m=on-cross-edge param u v s' for u v s'*]
leaf-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*]]
 $\implies \text{step } s \leq \text{SPEC } (\text{DFS-invar } G \text{ param})$
 — A step preserves the (best) invariant.
 ⟨*proof*⟩

lemma *DFS-invar-step'*: [[*nofail (step s); DFS-invar G param s; cond s*]]
 $\implies \text{step } s \leq \text{SPEC } (\text{DFS-invar } G \text{ param})$
 ⟨*proof*⟩

We define symbolic names for the preconditions of certain operations

definition *pre-is-break* $s \equiv \text{DFS-invar } G \text{ param } s$

definition *pre-on-new-root* $v0 \ s' \equiv \exists s.$
 $\text{DFS-invar } G \text{ param } s \wedge \text{cond } s \wedge$
 $\text{stack } s = [] \wedge v0 \in V0 \wedge v0 \notin \text{dom } (\text{discovered } s) \wedge$
 $s' = \text{new-root } v0 \ s$

definition *pre-on-finish* $u \ s' \equiv \exists s.$
 $\text{DFS-invar } G \text{ param } s \wedge \text{cond } s \wedge$
 $\text{stack } s \neq [] \wedge u = \text{hd } (\text{stack } s) \wedge \text{pending } s = \{u\} \wedge s' = \text{finish } u \ s$

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

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

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

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

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*: $\text{on-init param } \leq_n \text{SPEC } (\lambda x. I(\text{empty-state } x))$

assumes *new-root*: $\bigwedge s s' v0.$

$\llbracket \text{DFS-invar } G \text{ param } s; I s; \text{cond } s; \neg \text{is-break param } s;$
 $\text{stack } s = []; v0 \in V0; v0 \notin \text{dom}(\text{discovered } s);$
 $s' = \text{new-root } v0 s \rrbracket$

$\implies \text{on-new-root param } v0 s' \leq_n$

$\text{SPEC } (\lambda x. \text{DFS-invar } G \text{ param } (s'(\text{state.more} := x)))$
 $\longrightarrow I(s'(\text{state.more} := x))$

assumes *finish*: $\bigwedge s s' u.$

$\llbracket \text{DFS-invar } G \text{ param } s; I s; \text{cond } s; \neg \text{is-break param } s;$
 $\text{stack } s \neq []; u = \text{hd}(\text{stack } s);$
 $\text{pending } s \text{ “ } \{u\} = \{ \};$
 $s' = \text{finish } u s \rrbracket$

$\implies \text{on-finish param } u s' \leq_n$

$\text{SPEC } (\lambda x. \text{DFS-invar } G \text{ param } (s'(\text{state.more} := x)))$
 $\longrightarrow I(s'(\text{state.more} := x))$

assumes *cross-edge*: $\bigwedge s s' u v.$

$\llbracket \text{DFS-invar } G \text{ param } s; I s; \text{cond } s; \neg \text{is-break param } s;$
 $\text{stack } s \neq []; (u, v) \in \text{pending } s; u = \text{hd}(\text{stack } s);$
 $v \in \text{dom}(\text{discovered } s); v \in \text{dom}(\text{finished } s);$
 $s' = \text{cross-edge } u v (s(\text{pending} := \text{pending } s - \{(u,v)\})) \rrbracket$

$\implies \text{on-cross-edge param } u v s' \leq_n$

$\text{SPEC } (\lambda x. \text{DFS-invar } G \text{ param } (s'(\text{state.more} := x)))$
 $\longrightarrow I(s'(\text{state.more} := x))$

assumes *back-edge*: $\bigwedge s s' u v.$

$\llbracket \text{DFS-invar } G \text{ param } s; I s; \text{cond } s; \neg \text{is-break param } s;$
 $\text{stack } s \neq []; (u, v) \in \text{pending } s; u = \text{hd}(\text{stack } s);$
 $v \in \text{dom}(\text{discovered } s); v \notin \text{dom}(\text{finished } s);$
 $s' = \text{back-edge } u v (s(\text{pending} := \text{pending } s - \{(u,v)\})) \rrbracket$

$\implies \text{on-back-edge param } u v s' \leq_n$

$\text{SPEC } (\lambda x. \text{DFS-invar } G \text{ param } (s'(\text{state.more} := x)))$
 $\longrightarrow I(s'(\text{state.more} := x))$

assumes *discover*: $\bigwedge s s' u v.$
 $\llbracket \text{DFS-invar } G \text{ param } s; I s; \text{ cond } s; \neg \text{ is-break param } s;$
 $\text{ stack } s \neq []; (u, v) \in \text{ pending } s; u = \text{hd}(\text{stack } s);$
 $v \notin \text{dom}(\text{discovered } s);$
 $s' = \text{discover } u v (s(\text{pending} := \text{pending } s - \{(u,v)\})) \rrbracket$
 $\implies \text{on-discover param } u v s' \leq_n$
 $\text{SPEC } (\lambda x. \text{DFS-invar } G \text{ param } (s'(\text{state.more} := x)))$
 $\longrightarrow I (s'(\text{state.more} := x))$

shows *is-invar* I
 $\langle \text{proof} \rangle$

lemma *establish-invarI* [*case-names init new-root finish cross-edge back-edge discover*]:

— Establish a DFS invariant (symbolic preconditions).
assumes *init*: $\text{on-init param } \leq_n \text{SPEC } (\lambda x. I (\text{empty-state } x))$
assumes *new-root*: $\bigwedge s' v \theta. \text{pre-on-new-root } v \theta s'$
 $\implies \text{on-new-root param } v \theta s' \leq_n$
 $\text{SPEC } (\lambda x. \text{DFS-invar } G \text{ param } (s'(\text{state.more} := x)))$
 $\longrightarrow I (s'(\text{state.more} := x))$

assumes *finish*: $\bigwedge s' u. \text{pre-on-finish } u s'$
 $\implies \text{on-finish param } u s' \leq_n$
 $\text{SPEC } (\lambda x. \text{DFS-invar } G \text{ param } (s'(\text{state.more} := x)))$
 $\longrightarrow I (s'(\text{state.more} := x))$

assumes *cross-edge*: $\bigwedge s' u v. \text{pre-on-cross-edge } u v s'$
 $\implies \text{on-cross-edge param } u v s' \leq_n$
 $\text{SPEC } (\lambda x. \text{DFS-invar } G \text{ param } (s'(\text{state.more} := x)))$
 $\longrightarrow I (s'(\text{state.more} := x))$

assumes *back-edge*: $\bigwedge s' u v. \text{pre-on-back-edge } u v s'$
 $\implies \text{on-back-edge param } u v s' \leq_n$
 $\text{SPEC } (\lambda x. \text{DFS-invar } G \text{ param } (s'(\text{state.more} := x)))$
 $\longrightarrow I (s'(\text{state.more} := x))$

assumes *discover*: $\bigwedge s' u v. \text{pre-on-discover } u v s'$
 $\implies \text{on-discover param } u v s' \leq_n$
 $\text{SPEC } (\lambda x. \text{DFS-invar } G \text{ param } (s'(\text{state.more} := x)))$
 $\longrightarrow I (s'(\text{state.more} := x))$

shows *is-invar* I
 $\langle \text{proof} \rangle$

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*: $\bigwedge u v s. \text{on-discover param } u v s = \text{on-new-root param } v s$
assumes *init*: $\text{on-init param } \leq_n \text{SPEC } (\lambda x. I (\text{empty-state } x))$
assumes *new-discover*: $\bigwedge s s' v.$
 $\llbracket \text{DFS-invar } G \text{ param } s; I s; \text{ cond } s; \neg \text{ is-break param } s;$
 $v \notin \text{dom}(\text{discovered } s);$
 $\text{discovered } s' = (\text{discovered } s)(v \mapsto \text{counter } s); \text{finished } s' = \text{finished } s;$
 $\text{counter } s' = \text{Suc}(\text{counter } s); \text{stack } s' = v \# \text{stack } s;$
 $\text{back-edges } s' = \text{back-edges } s; \text{cross-edges } s' = \text{cross-edges } s;$

$tree\text{-}edges\ s' \supseteq tree\text{-}edges\ s;$
 $state.\text{more}\ s' = state.\text{more}\ s]$
 $\implies on\text{-}new\text{-}root\ param\ v\ s' \leq_n$
 $SPEC\ (\lambda x. DFS\text{-}invar\ G\ param\ (s'(\!|state.\text{more} := x\!|)))$
 $\longrightarrow I\ (s'(\!|state.\text{more} := x\!|))$

assumes *finish*: $\bigwedge s\ s'\ u.$
 $[[DFS\text{-}invar\ G\ param\ s; I\ s; cond\ s; \neg\ is\text{-}break\ param\ s;$
 $stack\ s \neq []; u = hd\ (stack\ s);$
 $pending\ s\ \text{“}\ \{u\} = \{\};$
 $s' = finish\ u\ s]]$
 $\implies on\text{-}finish\ param\ u\ s' \leq_n$
 $SPEC\ (\lambda x. DFS\text{-}invar\ G\ param\ (s'(\!|state.\text{more} := x\!|)))$
 $\longrightarrow I\ (s'(\!|state.\text{more} := x\!|))$

assumes *cross-edge*: $\bigwedge s\ s'\ u\ v.$
 $[[DFS\text{-}invar\ G\ param\ s; I\ s; cond\ s; \neg\ is\text{-}break\ param\ s;$
 $stack\ s \neq []; (u, v) \in pending\ s; u = hd\ (stack\ s);$
 $v \in dom\ (discovered\ s); v \in dom\ (finished\ s);$
 $s' = cross\text{-}edge\ u\ v\ (s(\!|pending := pending\ s - \{(u,v)\}\!|))]]$
 $\implies on\text{-}cross\text{-}edge\ param\ u\ v\ s' \leq_n$
 $SPEC\ (\lambda x. DFS\text{-}invar\ G\ param\ (s'(\!|state.\text{more} := x\!|)))$
 $\longrightarrow I\ (s'(\!|state.\text{more} := x\!|))$

assumes *back-edge*: $\bigwedge s\ s'\ u\ v.$
 $[[DFS\text{-}invar\ G\ param\ s; I\ s; cond\ s; \neg\ is\text{-}break\ param\ s;$
 $stack\ s \neq []; (u, v) \in pending\ s; u = hd\ (stack\ s);$
 $v \in dom\ (discovered\ s); v \notin dom\ (finished\ s);$
 $s' = back\text{-}edge\ u\ v\ (s(\!|pending := pending\ s - \{(u,v)\}\!|))]]$
 $\implies on\text{-}back\text{-}edge\ param\ u\ v\ s' \leq_n$
 $SPEC\ (\lambda x. DFS\text{-}invar\ G\ param\ (s'(\!|state.\text{more} := x\!|)))$
 $\longrightarrow I\ (s'(\!|state.\text{more} := x\!|))$

shows *is-invar* I
 $\langle proof \rangle$

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*: $\bigwedge u\ v\ s. on\text{-}back\text{-}edge\ param\ u\ v\ s = on\text{-}cross\text{-}edge\ param\ u\ v\ s$

assumes *init*: $on\text{-}init\ param \leq_n SPEC\ (\lambda x. I\ (empty\text{-}state\ x))$

assumes *new-root*: $\bigwedge s\ s'\ v0.$

$[[DFS\text{-}invar\ G\ param\ s; I\ s; cond\ s; \neg\ is\text{-}break\ param\ s;$
 $stack\ s = []; v0 \in V0; v0 \notin dom\ (discovered\ s);$
 $s' = new\text{-}root\ v0\ s]]$
 $\implies on\text{-}new\text{-}root\ param\ v0\ s' \leq_n$
 $SPEC\ (\lambda x. DFS\text{-}invar\ G\ param\ (s'(\!|state.\text{more} := x\!|)))$
 $\longrightarrow I\ (s'(\!|state.\text{more} := x\!|))$

assumes *finish*: $\bigwedge s\ s'\ u.$

$[[DFS\text{-}invar\ G\ param\ s; I\ s; cond\ s; \neg\ is\text{-}break\ param\ s;$
 $stack\ s \neq []; u = hd\ (stack\ s);$
 $pending\ s\ \text{“}\ \{u\} = \{\};$

$s' = \text{finish } u \ s]$
 $\implies \text{on-finish param } u \ s' \leq_n$
 $\text{SPEC } (\lambda x. \text{DFS-invar } G \text{ param } (s'(\text{state.more} := x)))$
 $\longrightarrow I (s'(\text{state.more} := x))$

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

assumes *discover*: $\bigwedge s \ s' \ u \ v.$
 $\llbracket \text{DFS-invar } G \text{ param } s; I \ s; \text{cond } s; \neg \text{is-break param } s;$
 $\text{stack } s \neq []; (u, v) \in \text{pending } s; u = \text{hd } (\text{stack } s);$
 $v \notin \text{dom } (\text{discovered } s);$
 $s' = \text{discover } u \ v \ (s(\text{pending} := \text{pending } s - \{(u, v)\})) \rrbracket$
 $\implies \text{on-discover param } u \ v \ s' \leq_n$
 $\text{SPEC } (\lambda x. \text{DFS-invar } G \text{ param } (s'(\text{state.more} := x)))$
 $\longrightarrow I (s'(\text{state.more} := x))$

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*:

$\bigwedge u \ v \ s. \text{on-discover param } u \ v \ s = \text{on-new-root param } v \ s$

$\bigwedge u \ v \ s. \text{on-back-edge param } u \ v \ s = \text{on-cross-edge param } u \ v \ s$

assumes *init*: $\text{on-init param } \leq_n \text{SPEC } (\lambda x. I (\text{empty-state } x))$

assumes *new-discover*: $\bigwedge s \ s' \ v.$

$\llbracket \text{DFS-invar } G \text{ param } s; I \ s; \text{cond } s; \neg \text{is-break param } s;$

$v \notin \text{dom } (\text{discovered } s);$

$\text{discovered } s' = (\text{discovered } s)(v \rightarrow \text{counter } s); \text{finished } s' = \text{finished } s;$

$\text{counter } s' = \text{Suc } (\text{counter } s); \text{stack } s' = v \# \text{stack } s;$

$\text{back-edges } s' = \text{back-edges } s; \text{cross-edges } s' = \text{cross-edges } s;$

$\text{tree-edges } s' \supseteq \text{tree-edges } s;$

$\text{state.more } s' = \text{state.more } s \rrbracket$

$\implies \text{on-new-root param } v \ s' \leq_n$

$\text{SPEC } (\lambda x. \text{DFS-invar } G \text{ param } (s'(\text{state.more} := x)))$

$\longrightarrow I (s'(\text{state.more} := x))$

assumes *finish*: $\bigwedge s \ s' \ u.$

$\llbracket \text{DFS-invar } G \text{ param } s; I \ s; \text{cond } s; \neg \text{is-break param } s;$

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

assumes $cross\text{-}back\text{-}edge: \bigwedge s\ s'\ u\ v.$

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

shows $is\text{-}invar\ I$

(proof)

lemma $is\text{-}invarI\text{-}full$ [*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 $new\text{-}root: \bigwedge s\ s'\ v0\ e.$

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

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

$\llbracket I\ s; cond\ s; DFS\text{-}invar\ G\ param\ s; DFS\text{-}invar\ G\ param\ s';$
 $stack\ s \neq []; pending\ s\ \text{“}\ \{u\} = \{\};$
 $u = hd\ (stack\ s); s' = finish\ u\ s\ (state.more := e)\ \rrbracket$
 $\implies I\ s'$

and $cross\text{-}edge: \bigwedge s\ s'\ u\ v\ e.$

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

and $back\text{-}edge: \bigwedge s\ s'\ u\ v\ e.$

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

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

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'*: $\bigwedge e. I (\text{empty-state } e)$
and *new-root'*: $\bigwedge s s' v0 e.$
 $\llbracket I s; \text{cond } s; \text{DFS-invar } G \text{ param } s; \text{DFS-invar } G \text{ param } s';$
 $\text{stack } s = []; v0 \notin \text{dom } (\text{discovered } s); v0 \in V0;$
 $s' = \text{new-root } v0 s (\text{state.more} := e) \rrbracket$
 $\implies I s'$
and *finish'*: $\bigwedge s s' u e.$
 $\llbracket I s; \text{cond } s; \text{DFS-invar } G \text{ param } s; \text{DFS-invar } G \text{ param } s';$
 $\text{stack } s \neq []; \text{pending } s \text{ `` } \{u\} = \{ \};$
 $u = \text{hd } (\text{stack } s); s' = \text{finish } u s (\text{state.more} := e) \rrbracket$
 $\implies I s'$
and *visited'*: $\bigwedge s s' u v e c b.$
 $\llbracket I s; \text{cond } s; \text{DFS-invar } G \text{ param } s; \text{DFS-invar } G \text{ param } s';$
 $\text{stack } s \neq []; v \in \text{pending } s \text{ `` } \{u\}; v \in \text{dom } (\text{discovered } s);$
 $u = \text{hd } (\text{stack } s);$
 $\text{cross-edges } s \subseteq c; \text{back-edges } s \subseteq b;$
 $s' = s(\text{pending} := \text{pending } s - \{(u,v)\},$
 $\text{state.more} := e,$
 $\text{cross-edges} := c,$
 $\text{back-edges} := b) \rrbracket$
 $\implies I s'$
and *discover'*: $\bigwedge s s' u v e.$
 $\llbracket I s; \text{cond } s; \text{DFS-invar } G \text{ param } s; \text{DFS-invar } G \text{ param } s';$
 $\text{stack } s \neq []; v \in \text{pending } s \text{ `` } \{u\}; v \notin \text{dom } (\text{discovered } s);$
 $u = \text{hd } (\text{stack } s);$
 $s' = (\text{discover } u v (s(\text{pending} := \text{pending } s - \{(u,v)\}))) (\text{state.more} := e) \rrbracket$
 $\implies I s'$
shows *is-invar* I
 $\langle \text{proof} \rangle$

end

1.1.5 Basic Invariants

We establish some basic invariants

context *param-DFS* **begin**

definition *basic-invar* $s \equiv$

$set (stack\ s) = dom (discovered\ s) - dom (finished\ s) \wedge$
 $distinct (stack\ s) \wedge$
 $(stack\ s \neq [] \implies last (stack\ s) \in V0) \wedge$
 $dom (finished\ s) \subseteq dom (discovered\ s) \wedge$
 $Domain (pending\ s) \subseteq dom (discovered\ s) - dom (finished\ s) \wedge$
 $pending\ s \subseteq E$

lemma *i-basic-invar*: *is-invar basic-invar*

$\langle proof \rangle$

end

context *DFS-invar* **begin**

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

lemma *pending-ssE*: $pending\ s \subseteq E$

$\langle proof \rangle$

lemma *pendingD*:

$(u,v) \in pending\ s \implies (u,v) \in E \wedge u \in dom (discovered\ s)$
 $\langle proof \rangle$

lemma *stack-set-def*:

$set (stack\ s) = dom (discovered\ s) - dom (finished\ s)$
 $\langle proof \rangle$

lemma *stack-discovered*:

$set (stack\ s) \subseteq dom (discovered\ s)$
 $\langle proof \rangle$

lemma *stack-distinct*:

$distinct (stack\ s)$
 $\langle proof \rangle$

lemma *last-stack-in-V0*:

$stack\ s \neq [] \implies last (stack\ s) \in V0$
 $\langle proof \rangle$

lemma *stack-not-finished*:

$x \in set (stack\ s) \implies x \notin dom (finished\ s)$
 $\langle proof \rangle$

lemma *discovered-not-stack-imp-finished*:

$x \in dom (discovered\ s) \implies x \notin set (stack\ s) \implies x \in dom (finished\ s)$
 $\langle proof \rangle$

lemma *finished-discovered*:

$dom (finished\ s) \subseteq dom (discovered\ s)$
 $\langle proof \rangle$

lemma *finished-no-pending*:

$v \in dom (finished\ s) \implies pending\ s \text{ “ } \{v\} = \{\}$
 $\langle proof \rangle$

lemma *discovered-eq-finished-un-stack*:

$dom (discovered\ s) = dom (finished\ s) \cup set (stack\ s)$
 $\langle proof \rangle$

lemma *pending-on-stack*:

$(v,w) \in pending\ s \implies v \in set (stack\ s)$
 $\langle proof \rangle$

lemma *empty-stack-imp-empty-pending*:

$stack\ s = [] \implies pending\ s = \{\}$
 $\langle proof \rangle$

end

context *param-DFS* **begin**

lemma *i-discovered-reachable*:

$is-invar (\lambda s. dom (discovered\ s) \subseteq reachable)$
 $\langle proof \rangle$

definition *discovered-closed* $s \equiv$

$E \text{ “ } dom (finished\ s) \subseteq dom (discovered\ s)$
 $\wedge (E - pending\ s) \text{ “ } set (stack\ s) \subseteq dom (discovered\ s)$

lemma *i-discovered-closed*: $is-invar\ discovered-closed$

$\langle proof \rangle$

lemma *i-discovered-finite*: $is-invar (\lambda s. finite (dom (discovered\ s)))$

$\langle proof \rangle$

end

context *DFS-invar*

begin

lemmas *discovered-reachable* =

$i-discovered-reachable$ [THEN *make-invar-thm*]

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

$\langle proof \rangle$

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*) \subseteq *dom* (*discovered* *s*) ”
⟨*proof*⟩

lemma *finished-imp-succ-discovered*:

$v \in \text{dom}(\text{finished } s) \implies w \in \text{succ } v \implies w \in \text{dom}(\text{discovered } s)$
⟨*proof*⟩

lemma *pending-reachable*: *pending* *s* \subseteq *reachable* \times *reachable*

⟨*proof*⟩

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

⟨*proof*⟩

lemma *no-pending-imp-succ-discovered*:

assumes $u \in \text{dom}(\text{discovered } s)$

and *pending* *s* “ $\{u\} = \{\}$ ”

and $v \in \text{succ } u$

shows $v \in \text{dom}(\text{discovered } s)$

⟨*proof*⟩

lemma *nc-finished-eq-reachable*:

assumes *NC*: $\neg \text{cond } s \neg \text{is-break param } s$

shows *dom* (*finished* *s*) = *reachable*

⟨*proof*⟩

lemma *nc-V0-finished*:

assumes *NC*: $\neg \text{cond } s \neg \text{is-break param } s$

shows $V0 \subseteq \text{dom}(\text{finished } s)$

⟨*proof*⟩

lemma *nc-discovered-eq-finished*:

assumes *NC*: $\neg \text{cond } s \neg \text{is-break param } s$

shows *dom* (*discovered* *s*) = *dom* (*finished* *s*)

⟨*proof*⟩

lemma *nc-discovered-eq-reachable*:

assumes *NC*: $\neg \text{cond } s \neg \text{is-break param } s$

shows *dom* (*discovered* *s*) = *reachable*

⟨*proof*⟩

lemma *nc-fin-closed*:

```

assumes NC:  $\neg cond\ s$ 
assumes NB:  $\neg is\ break\ param\ s$ 
shows  $E''dom\ (finished\ s) \subseteq dom\ (finished\ s)$ 
   $\langle proof \rangle$ 

```

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  $\equiv inv\ image$ 
  (finite-psupset reachable  $\langle *lex* \rangle$  finite-psubset  $\langle *lex* \rangle$  less-than)
  ( $\lambda s. (dom\ (discovered\ s), pending\ s, length\ (stack\ s))$ )

```

lemma *param-dfs-variant-wf*[*simp*, *intro!*]:

```

assumes [simp, intro!]: finite reachable
shows wf param-dfs-variant
   $\langle proof \rangle$ 

```

lemma *param-dfs-variant-step*:

```

assumes A: DFS-invar G param s cond s nofail it-dfs
shows step s  $\leq SPEC\ (\lambda s'. (s',s) \in param\ dfs\ variant)$ 
   $\langle proof \rangle$ 

```

end

context *param-DFS* **begin**

```

lemma it-dfsT-eq-it-dfs:
assumes [simp, intro!]: finite reachable
shows it-dfsT = it-dfs
   $\langle proof \rangle$ 

```

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 \implies *nofail* (*on-new-root param v0 s*)

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

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

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

assumes *nofail-on-discover*:
pre-on-discover u v s \implies *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*)
 \langle *proof* \rangle

lemma *it-dfs-eq-spec*: *it-dfs* = *SPEC* ($\lambda s. \text{DFS-invar } G \text{ param } s \wedge \neg \text{cond } s$)
 \langle *proof* \rangle

lemma *it-dfs-correct*: *it-dfs* \leq *SPEC* ($\lambda s. \text{DFS-invar } G \text{ param } s \wedge \neg \text{cond } s$)
 \langle *proof* \rangle

lemma *it-dfs-SPEC*:
assumes $\bigwedge s. \llbracket \text{DFS-invar } G \text{ param } s; \neg \text{cond } s \rrbracket \implies P \ s$
shows *it-dfs* \leq *SPEC* *P*
 \langle *proof* \rangle

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

lemma *it-dfsT-SPEC*:
assumes *finite reachable*
assumes $\bigwedge s. \llbracket \text{DFS-invar } G \text{ param } s; \neg \text{cond } s \rrbracket \implies P \ s$
shows *it-dfsT* \leq *SPEC* *P*
 \langle *proof* \rangle

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* $\langle \delta \rangle$

notation *the-finished* $\langle \varphi \rangle$

end

context *param-DFS* **begin context begin interpretation** *timing-syntax* $\langle proof \rangle$

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

$\langle proof \rangle$

end end

context *DFS-invar* **begin context begin interpretation** *timing-syntax* $\langle proof \rangle$

lemmas *s-timing-common-inv* =

timing-common-inv[*THEN make-invar-thm*]

lemma *timing-less-counter*:

$v \in \text{dom} (\text{discovered } s) \implies \delta s v < \text{counter } s$

$v \in \text{dom} (\text{finished } s) \implies \varphi s v < \text{counter } s$

$\langle \text{proof} \rangle$

lemma *disc-lt-fin*:

$v \in \text{dom} (\text{finished } s) \implies \delta s v < \varphi s v$

$\langle \text{proof} \rangle$

lemma *disc-unequal*:

assumes $v \in \text{dom} (\text{discovered } s) \ w \in \text{dom} (\text{discovered } s)$

and $v \neq w$

shows $\delta s v \neq \delta s w$

$\langle \text{proof} \rangle$

lemma *fin-unequal*:

assumes $v \in \text{dom} (\text{finished } s) \ w \in \text{dom} (\text{finished } s)$

and $v \neq w$

shows $\varphi s v \neq \varphi s w$

$\langle \text{proof} \rangle$

lemma *finished-succ-fin*:

assumes $v \in \text{dom} (\text{finished } s)$

and $w \in \text{succ } v$

shows $\delta s w < \varphi s v$

$\langle \text{proof} \rangle$

end end

context *param-DFS* **begin context begin interpretation** *timing-syntax* $\langle \text{proof} \rangle$

lemma *i-prev-stack-discover-all*:

$\text{is-invar } (\lambda s. \forall n < \text{length} (\text{stack } s). \forall v \in \text{set} (\text{drop} (\text{Suc } n) (\text{stack } s)).$

$\delta s (\text{stack } s ! n) > \delta s v)$

$\langle \text{proof} \rangle$

end end

context *DFS-invar* **begin context begin interpretation** *timing-syntax* $\langle \text{proof} \rangle$

lemmas *prev-stack-discover-all*

$= \text{i-prev-stack-discover-all} [\text{THEN } \text{make-invar-thm}]$

lemma *prev-stack-discover*:

$\llbracket n < \text{length} (\text{stack } s); v \in \text{set} (\text{drop} (\text{Suc } n) (\text{stack } s)) \rrbracket$

$\implies \delta s (\text{stack } s ! n) > \delta s v$

$\langle \text{proof} \rangle$

lemma *Suc-stack-discover*:

assumes $n: n < (\text{length} (\text{stack } s)) - 1$

shows $\delta s (\text{stack } s ! n) > \delta s (\text{stack } s ! \text{Suc } n)$

<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* $s \equiv$

$\forall v \in dom\ (discovered\ s). \forall w \in dom\ (discovered\ s).$

$\delta\ s\ v < \delta\ s\ w \wedge v \in dom\ (finished\ s) \longrightarrow ($

$\varphi\ s\ v < \delta\ s\ w \text{ — disjoint}$

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

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 $\varphi\ s\ v < \delta\ s\ w \text{ — disjoint}$

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

<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 $v \in \text{dom } (\text{finished } s)$
and $w \in \text{succ } v$
and $\delta s v < \delta s w$
shows $w \in \text{dom } (\text{finished } s) \wedge \varphi s w < \varphi s v$
 $\langle \text{proof} \rangle$

end end

1.2.3 Edge Types

context *param-DFS*

begin

abbreviation $\text{edges } s \equiv \text{tree-edges } s \cup \text{cross-edges } s \cup \text{back-edges } s$

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

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

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

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

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

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

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

end

context *DFS-invar* **begin**

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

lemma *edges-ss-reachable-edges*:
edges $s \subseteq E \cap \text{reachable} \times \text{UNIV}$
<proof>

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

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

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

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

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

lemma *tree-edge-imp-discovered*:
 $(v,w) \in \text{tree-edges } s \implies v \in \text{dom } (\text{discovered } s)$
 $(v,w) \in \text{tree-edges } s \implies w \in \text{dom } (\text{discovered } s)$
<proof>

lemma *back-edge-imp-discovered*:
 $(v,w) \in \text{back-edges } s \implies v \in \text{dom } (\text{discovered } s)$
 $(v,w) \in \text{back-edges } s \implies w \in \text{dom } (\text{discovered } s)$
<proof>

lemma *cross-edge-imp-discovered*:
 $(v,w) \in \text{cross-edges } s \implies v \in \text{dom } (\text{discovered } s)$
 $(v,w) \in \text{cross-edges } s \implies w \in \text{dom } (\text{discovered } s)$
<proof>

lemma *edge-imp-discovered*:
 $(v,w) \in \text{edges } s \implies v \in \text{dom } (\text{discovered } s)$
 $(v,w) \in \text{edges } s \implies w \in \text{dom } (\text{discovered } s)$
 $\langle \text{proof} \rangle$

lemma *tree-edges-finite*[*simp, intro!*]: *finite* (*tree-edges* *s*)
 $\langle \text{proof} \rangle$

lemma *cross-edges-finite*[*simp, intro!*]: *finite* (*cross-edges* *s*)
 $\langle \text{proof} \rangle$

lemma *back-edges-finite*[*simp, intro!*]: *finite* (*back-edges* *s*)
 $\langle \text{proof} \rangle$

lemma *edges-finite*: *finite* (*edges* *s*)
 $\langle \text{proof} \rangle$

end

Properties of the DFS Tree

context *DFS-invar* **begin context** **begin interpretation** *timing-syntax* $\langle \text{proof} \rangle$

lemma *tree-edge-disc-lt-fin*:
 $(v,w) \in \text{tree-edges } s \implies v \in \text{dom } (\text{finished } s) \implies \delta s w < \varphi s v$
 $\langle \text{proof} \rangle$

lemma *back-edge-disc-lt-fin*:
 $(v,w) \in \text{back-edges } s \implies v \in \text{dom } (\text{finished } s) \implies \delta s w < \varphi s v$
 $\langle \text{proof} \rangle$

lemma *cross-edge-disc-lt-fin*:
 $(v,w) \in \text{cross-edges } s \implies v \in \text{dom } (\text{finished } s) \implies \delta s w < \varphi s v$
 $\langle \text{proof} \rangle$

end end

context *param-DFS* **begin**

lemma *i-stack-is-tree-path*:
 $\text{is-invar } (\lambda s. \text{stack } s \neq [] \longrightarrow (\exists v0 \in V0. \text{path } (\text{tree-edges } s) v0 (\text{rev } (\text{tl } (\text{stack } s))) (\text{hd } (\text{stack } s))))$
 $\langle \text{proof} \rangle$

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 $\neq [] \implies \exists v0 \in V0. \text{path } E \ v0 \ (\text{rev } (\text{tl } (\text{stack } s))) \ (\text{hd } (\text{stack } s))$
<proof>

lemma *hd-succ-stack-is-path*:

assumes *ne*: *stack s* $\neq []$
and *succ*: $v \in \text{succ } (\text{hd } (\text{stack } s))$
shows $\exists v0 \in V0. \text{path } E \ v0 \ (\text{rev } (\text{stack } s)) \ v$
<proof>

lemma *tl-stack-hd-tree-path*:

assumes *stack s* $\neq []$
and $v \in \text{set } (\text{tl } (\text{stack } s))$
shows $(v, \text{hd } (\text{stack } s)) \in (\text{tree-edges } s)^+$
<proof>

end

context *param-DFS begin*

definition *tree-discovered-inv s* \equiv

$(\text{tree-edges } s = \{\}) \longrightarrow \text{dom } (\text{discovered } s) \subseteq V0 \wedge (\text{stack } s = []$
 $\vee (\exists v0 \in V0. \text{stack } s = [v0]))$
 $\wedge (\text{tree-edges } s \neq \{\}) \longrightarrow (\text{tree-edges } s)^+ \text{ “ } V0 \cup V0 = \text{dom}$
 $(\text{discovered } s) \cup V0$

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*:

$v \notin V0 \implies v \in \text{dom } (\text{discovered } s) \iff (\exists v0 \in V0. (v0, v) \in (\text{tree-edges } s)^+)$
<proof>

lemma *i-tree-one-predecessor*:

is-invar $(\lambda s. \forall (v, v') \in \text{tree-edges } s. \forall y. y \neq v \longrightarrow (y, v') \notin \text{tree-edges } s)$
<proof>

lemma (**in** *DFS-invar*) *tree-one-predecessor*:

assumes $(v, w) \in \text{tree-edges } s$
and $a \neq v$
shows $(a, w) \notin \text{tree-edges } s$
<proof>

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

$\llbracket (v,w) \in \text{tree-edges } s; (u,w) \in \text{tree-edges } s \rrbracket \implies v=u$
 $\langle \text{proof} \rangle$

context begin interpretation *timing-syntax* $\langle \text{proof} \rangle$

lemma *i-tree-edge-disc:*

is-invar $(\lambda s. \forall (v,v') \in \text{tree-edges } s. \delta s v < \delta s v')$

$\langle \text{proof} \rangle$

end end

context *DFS-invar* **begin context begin interpretation** *timing-syntax* $\langle \text{proof} \rangle$

lemma *tree-edge-disc:*

$(v,w) \in \text{tree-edges } s \implies \delta s v < \delta s w$

$\langle \text{proof} \rangle$

lemma *tree-path-disc:*

$(v,w) \in (\text{tree-edges } s)^+ \implies \delta s v < \delta s w$

$\langle \text{proof} \rangle$

lemma *no-loop-in-tree:*

$(v,v) \notin (\text{tree-edges } s)^+$

$\langle \text{proof} \rangle$

lemma *tree-acyclic:*

acyclic $(\text{tree-edges } s)$

$\langle \text{proof} \rangle$

lemma *no-self-loop-in-tree:*

$(v,v) \notin \text{tree-edges } s$

$\langle \text{proof} \rangle$

lemma *tree-edge-unequal:*

$(v,w) \in \text{tree-edges } s \implies v \neq w$

$\langle \text{proof} \rangle$

lemma *tree-path-unequal:*

$(v,w) \in (\text{tree-edges } s)^+ \implies v \neq w$

$\langle \text{proof} \rangle$

lemma *tree-subpath':*

assumes $x: (x,v) \in (\text{tree-edges } s)^+$

and $y: (y,v) \in (\text{tree-edges } s)^+$

and $x \neq y$

shows $(x,y) \in (\text{tree-edges } s)^+ \vee (y,x) \in (\text{tree-edges } s)^+$

$\langle \text{proof} \rangle$

lemma *tree-subpath:*

assumes $(x,v) \in (\text{tree-edges } s)^+$

and $(y,v) \in (\text{tree-edges } s)^+$
and $\delta: \delta s x < \delta s y$
shows $(x,y) \in (\text{tree-edges } s)^+$
 $\langle \text{proof} \rangle$

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

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

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

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

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

end end

context *param-DFS* **begin context** **begin interpretation** *timing-syntax* $\langle \text{proof} \rangle$

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

```

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  $v \in \text{dom}(\text{discovered } s)$ 
  and  $w \in \text{dom}(\text{finished } s)$ 
  and  $\delta s v < \delta s w$ 
  and  $v \notin \text{dom}(\text{finished } s)$ 
  shows  $(v, w) \in (\text{tree-edges } s)^+$ 
  ⟨proof⟩

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

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

lemma no-pending-succ-impl-path-in-tree:
  assumes  $v: v \in \text{dom}(\text{discovered } s)$  pending  $s$  “ $\{v\} = \{\}$ ”
  and  $w: w \in \text{succ } v$ 
  and  $\delta: \delta s v < \delta s w$ 
  shows  $(v, w) \in (\text{tree-edges } s)^+$ 
  ⟨proof⟩

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

```

Properties of Cross Edges

```

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

```

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

lemma *cross-edges-finished-decr*:
 $\llbracket (u,v) \in \text{cross-edges } s; u \in \text{dom } (\text{finished } s) \rrbracket \implies \varphi s v < \varphi s u$
 ⟨*proof*⟩

lemma *cross-edge-unequal*:
assumes *cross*: $(v,w) \in \text{cross-edges } s$
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 $(\lambda s. \forall (v,w) \in \text{back-edges } s. (w,v) \in (\text{tree-edges } s)^+ \vee w = v)$
 ⟨*proof*⟩

end end

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

lemma *back-edge-impl-tree-path*:
 $\llbracket (v,w) \in \text{back-edges } s; v \neq w \rrbracket \implies (w,v) \in (\text{tree-edges } s)^+$
 ⟨*proof*⟩

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

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

lemma *back-edges-tree-edges-disjoint*:

$$\text{back-edges } s \cap (\text{tree-edges } s)^+ = \{\}$$

<proof>

lemma *back-edge-finished*:

assumes $(v,w) \in \text{back-edges } s$

and $w \in \text{dom } (\text{finished } s)$

shows $v \in \text{dom } (\text{finished } s) \wedge \varphi s v \leq \varphi s w$

<proof>

end end

context *param-DFS* **begin context begin interpretation** *timing-syntax* *<proof>*

lemma *i-disc-imp-back-edge-or-pending*:

is-invar $(\lambda s. \forall (v,w) \in E.$

$$v \in \text{dom } (\text{discovered } s) \wedge w \in \text{dom } (\text{discovered } s)$$

$$\wedge \delta s v \geq \delta s w$$

$$\wedge (w \in \text{dom } (\text{finished } s) \longrightarrow v \in \text{dom } (\text{finished } s) \wedge \varphi s w \geq \varphi s v)$$

$$\longrightarrow (v,w) \in \text{back-edges } s \vee (v,w) \in \text{pending } s$$

<proof>

end end

context *DFS-invar* **begin context begin interpretation** *timing-syntax* *<proof>*

lemma *disc-imp-back-edge-or-pending*:

$\llbracket w \in \text{succ } v; v \in \text{dom } (\text{discovered } s); w \in \text{dom } (\text{discovered } s); \delta s w \leq \delta s v;$

$(w \in \text{dom } (\text{finished } s) \implies v \in \text{dom } (\text{finished } s) \wedge \varphi s v \leq \varphi s w) \rrbracket$

$\implies (v, w) \in \text{back-edges } s \vee (v, w) \in \text{pending } s$

<proof>

lemma *finished-imp-back-edge*:

$\llbracket w \in \text{succ } v; v \in \text{dom } (\text{finished } s); w \in \text{dom } (\text{finished } s);$

$\delta s w \leq \delta s v; \varphi s v \leq \varphi s w \rrbracket$

$\implies (v, w) \in \text{back-edges } s$

<proof>

lemma *finished-not-finished-imp-back-edge*:

$\llbracket w \in \text{succ } v; v \in \text{dom } (\text{finished } s); w \in \text{dom } (\text{discovered } s);$

$w \notin \text{dom } (\text{finished } s);$

$\delta s w \leq \delta s v \rrbracket$

$\implies (v, w) \in \text{back-edges } s$

<proof>

lemma *finished-self-loop-in-back-edges*:

assumes $v \in \text{dom } (\text{finished } s)$

and $(v,v) \in E$

shows $(v,v) \in \text{back-edges } s$

<proof>

end end

context *DFS-invar* **begin**

context begin interpretation *timing-syntax* \langle *proof* \rangle

lemma *tree-cross-acyclic*:

acyclic (*tree-edges* $s \cup$ *cross-edges* s) (**is** *acyclic* $?E$)
 \langle *proof* \rangle

end

lemma *cycle-contains-back-edge*:

assumes *cycle*: $(u,u) \in (\text{edges } s)^+$
shows $\exists v w. (u,v) \in (\text{edges } s)^* \wedge (v,w) \in \text{back-edges } s \wedge (w,u) \in (\text{edges } s)^*$
 \langle *proof* \rangle

lemma *cycle-needs-back-edge*:

assumes *back-edges* $s = \{\}$
shows *acyclic* (*edges* s)
 \langle *proof* \rangle

lemma *back-edge-closes-cycle*:

assumes *back-edges* $s \neq \{\}$
shows \neg *acyclic* (*edges* s)
 \langle *proof* \rangle

lemma *back-edge-closes-reachable-cycle*:

back-edges $s \neq \{\} \implies \neg$ *acyclic* ($E \cap \text{reachable} \times UNIV$)
 \langle *proof* \rangle

lemma *cycle-iff-back-edges*:

acyclic (*edges* s) \longleftrightarrow *back-edges* $s = \{\}$
 \langle *proof* \rangle

end

1.2.4 White Path Theorem

context *DFS* **begin**

context begin interpretation *timing-syntax* \langle *proof* \rangle

definition *white-path* **where**

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

lemma *white-path*:

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

$white-path\ s\ x\ y \longleftrightarrow (x,y) \in (tree-edges\ s)^*$

$\langle proof \rangle$
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* $s\ v\ scc$

$\langle proof \rangle$

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

end

context *DFS-invar* **begin**

lemma *scc-root-is-discovered*:

scc-root $s\ v\ scc \implies v \in dom\ (discovered\ s)$

$\langle proof \rangle$

lemma *scc-root-scc-tree-rtrancl*:

assumes *scc-root* $s\ v\ scc$

and $x \in scc\ x \in dom\ (discovered\ s)$

shows $(v,x) \in (tree-edges\ s)^*$

$\langle proof \rangle$

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 $x \in scc \ x \in \text{dom}(\text{discovered } s) \ x \neq v$

shows $(v, x) \in (\text{tree-edges } s)^+$

<proof>

lemma *scc-root-unique-scc*:

scc-root s v scc \implies scc-root s v scc' \implies 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 $v \in \text{dom}(\text{finished } s)$

and *scc-root s v scc*

shows $scc \subseteq \text{dom}(\text{finished } s)$

<proof>

context begin interpretation *timing-syntax* *<proof>*

lemma *scc-root-disc-le*:

assumes *scc-root s v scc*

and $x \in scc \ x \in \text{dom}(\text{discovered } s)$

shows $\delta s v \leq \delta s x$

<proof>

lemma *scc-root-fin-ge*:

assumes *scc-root s v scc*

and $v \in \text{dom}(\text{finished } s)$

and $x \in scc$
shows $\varphi s v \geq \varphi s x$
 ⟨proof⟩

lemma *scc-root-is-Min-disc*:
assumes $scc\text{-root } s v scc$
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 $is\text{-scc } E scc$
and $min: \delta s v = Min (\delta s ' (scc \cap dom (discovered s)))$
shows $scc\text{-root } s v scc$
 ⟨proof⟩

lemma *scc-root-iff-Min-disc*:
assumes $is\text{-scc } E scc r \in scc r \in dom (discovered s)$
shows $scc\text{-root } s r scc \longleftrightarrow Min (\delta s ' (scc \cap dom (discovered s))) = \delta s r$ (**is**
 $?L \longleftrightarrow ?R$)
 ⟨proof⟩

lemma *scc-root-exists*:
assumes $is\text{-scc } E scc$
and $scc: scc \cap dom (discovered s) \neq \{\}$
shows $\exists r. scc\text{-root } s r scc$
 ⟨proof⟩

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

lemma *scc-root-transfer'*:
assumes $discovered s = discovered s' tree\text{-edges } s = tree\text{-edges } s'$
shows $scc\text{-root } s r scc \longleftrightarrow scc\text{-root } s' r scc$
 ⟨proof⟩

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

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 pre-
pare 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  $\Rightarrow$  bool
  gds-is-empty-stack :: 's  $\Rightarrow$  bool
  gds-new-root :: 'v  $\Rightarrow$  's  $\Rightarrow$  's nres
  gds-get-pending :: 's  $\Rightarrow$  ('v  $\times$  'v option  $\times$  's) nres
  gds-finish :: 'v  $\Rightarrow$  's  $\Rightarrow$  's nres
  gds-is-discovered :: 'v  $\Rightarrow$  's  $\Rightarrow$  bool
  gds-is-finished :: 'v  $\Rightarrow$  's  $\Rightarrow$  bool
  gds-back-edge :: 'v  $\Rightarrow$  'v  $\Rightarrow$  's  $\Rightarrow$  's nres
  gds-cross-edge :: 'v  $\Rightarrow$  'v  $\Rightarrow$  's  $\Rightarrow$  's nres
  gds-discover :: 'v  $\Rightarrow$  'v  $\Rightarrow$  's  $\Rightarrow$  's nres
```

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

```
definition gen-step s  $\equiv$ 
  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 {
    (u, Vs, s)  $\leftarrow$  gds-get-pending gds s;
    case Vs of
```

$$\begin{aligned}
& \text{None} \Rightarrow \text{gds-finish gds u s} \\
& | \text{Some v} \Rightarrow \text{do } \{ \\
& \quad \text{if gds-is-discovered gds v s then (} \\
& \quad \quad \text{if gds-is-finished gds v s then} \\
& \quad \quad \quad \text{gds-cross-edge gds u v s} \\
& \quad \quad \text{else} \\
& \quad \quad \quad \text{gds-back-edge gds u v s} \\
& \quad \quad \text{) else} \\
& \quad \quad \text{gds-discover gds u v s} \\
& \quad \} \\
& \}
\end{aligned}$$

definition *gen-cond s*

$$\begin{aligned}
& \equiv (V0 \subseteq \{v. \text{gds-is-discovered gds v s}\} \longrightarrow \neg \text{gds-is-empty-stack gds s}) \\
& \wedge \neg \text{gds-is-break gds s}
\end{aligned}$$

definition *gen-dfs*

$$\equiv \text{gds-init gds} \gg \text{WHILE gen-cond gen-step}$$

definition *gen-dfsT*

$$\equiv \text{gds-init gds} \gg \text{WHILET gen-cond gen-step}$$

abbreviation *gen-discovered s* $\equiv \{v. \text{gds-is-discovered gds v s}\}$

abbreviation *gen-rwof* $\equiv \text{rwof (gds-init gds) gen-cond gen-step}$

definition *pre-new-root v0 s* \equiv

$$\begin{aligned}
& \text{gen-rwof s} \wedge \text{gds-is-empty-stack gds s} \wedge \neg \text{gds-is-break gds s} \\
& \wedge v0 \in V0 - \text{gen-discovered s}
\end{aligned}$$

definition *pre-get-pending s* \equiv

$$\text{gen-rwof s} \wedge \neg \text{gds-is-empty-stack gds s} \wedge \neg \text{gds-is-break gds s}$$

definition *post-get-pending u Vs s0 s* $\equiv \text{pre-get-pending s0}$

$$\wedge \text{inres (gds-get-pending gds s0) (u, Vs, s)}$$

definition *pre-finish u s0 s* $\equiv \text{post-get-pending u None s0 s}$

definition *pre-cross-edge u v s0 s* \equiv

$$\begin{aligned}
& \text{post-get-pending u (Some v) s0 s} \wedge \text{gds-is-discovered gds v s} \\
& \wedge \text{gds-is-finished gds v s}
\end{aligned}$$

definition *pre-back-edge u v s0 s* \equiv

$$\begin{aligned}
& \text{post-get-pending u (Some v) s0 s} \wedge \text{gds-is-discovered gds v s} \\
& \wedge \neg \text{gds-is-finished gds v s}
\end{aligned}$$

definition *pre-discover u v s0 s* \equiv

$$\text{post-get-pending u (Some v) s0 s} \wedge \neg \text{gds-is-discovered gds v s}$$

lemmas *pre-defs = pre-new-root-def pre-get-pending-def post-get-pending-def*

pre-finish-def pre-cross-edge-def pre-back-edge-def pre-discover-def

definition *gen-step-assert* $s \equiv$
 if *gds-is-empty-stack* *gds* *s* then do {
 $v0 \leftarrow SPEC (\lambda v0. v0 \in V0 \wedge \neg gds\text{-is-discovered } gds\ v0\ s);$
 ASSERT (*pre-new-root* $v0\ s$);
 gds-new-root *gds* $v0\ s$
 } else do {
 ASSERT (*pre-get-pending* *s*);
 let $s0 = GHOST\ s$;
 $(u, Vs, s) \leftarrow gds\text{-get-pending } gds\ s$;
 case Vs of
 None \Rightarrow do {*ASSERT* (*pre-finish* $u\ s0\ s$); *gds-finish* *gds* $u\ s$ }
 | Some $v \Rightarrow$ do {
 if *gds-is-discovered* *gds* $v\ s$ then do {
 if *gds-is-finished* *gds* $v\ s$ then do {
 ASSERT (*pre-cross-edge* $u\ v\ s0\ s$);
 gds-cross-edge *gds* $u\ v\ s$
 } else do {
 ASSERT (*pre-back-edge* $u\ v\ s0\ s$);
 gds-back-edge *gds* $u\ v\ s$
 }
 } else do {
 ASSERT (*pre-discover* $u\ v\ s0\ s$);
 gds-discover *gds* $u\ v\ s$
 }
 }
 }

definition *gen-dfs-assert*
 $\equiv gds\text{-init } gds \ggg WHILE\ gen\text{-cond } gen\text{-step-assert}$

definition *gen-dfsT-assert*
 $\equiv gds\text{-init } gds \ggg WHILET\ gen\text{-cond } gen\text{-step-assert}$

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

lemma *gen-step-eq-assert*: $\llbracket gen\text{-cond } s; gen\text{-rwof } s \rrbracket$
 $\implies gen\text{-step } s = gen\text{-step-assert } s$
 ⟨*proof*⟩

lemma *gen-dfs-eq-assert*: $gen\text{-dfs} = gen\text{-dfs-assert}$
 ⟨*proof*⟩

lemma *gen-dfsT-eq-assert*: $gen\text{-dfsT} = gen\text{-dfsT-assert}$
 ⟨*proof*⟩

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

shows $gen\text{-}rwof = gen\text{-}rwof\text{-}assert$
<proof>

lemma $gen\text{-}dfs\text{-}le\text{-}gen\text{-}dfsT$: $gen\text{-}dfs \leq gen\text{-}dfsT$
<proof>

end

locale $gen\text{-}dfs = gen\text{-}dfs\text{-}defs$ gds $V0$
for $gds :: ('v, 's)$ $gen\text{-}dfs\text{-}struct$
and $V0 :: 'v$ set

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

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

begin

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

definition $do\text{-}init \equiv do$ {
 $e \leftarrow on\text{-}init$ $param$;
 $gbs\text{-}init$ gbs e
}

definition $do\text{-}new\text{-}root$ $v0$

≡ *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*]:
 $(gds\text{-is-finished } gdsi, gds\text{-is-finished } gds) \in V \rightarrow S \rightarrow \text{bool-rel}$
assumes *is-empty-stack-param*[*param*]:
 $(gds\text{-is-empty-stack } gdsi, gds\text{-is-empty-stack } gds) \in S \rightarrow \text{bool-rel}$
assumes *is-break-param*[*param*]:
 $(gds\text{-is-break } gdsi, gds\text{-is-break } gds) \in S \rightarrow \text{bool-rel}$
assumes *init-refine*[*refine*]:
 $gds\text{-init } gdsi \leq \Downarrow S (gds\text{-init } gds)$
assumes *new-root-refine*[*refine*]:
 $\llbracket a.\text{pre-new-root } v0\ s; (v0i, v0) \in V; (si, s) \in S \rrbracket$
 $\implies gds\text{-new-root } gdsi\ v0i\ si \leq \Downarrow S (gds\text{-new-root } gds\ v0\ s)$
assumes *get-pending-refine*[*refine*]:
 $\llbracket a.\text{pre-get-pending } s; (si, s) \in S \rrbracket$
 $\implies gds\text{-get-pending } gdsi\ si \leq \Downarrow (V \times_r \langle V \rangle \text{option-rel} \times_r S) (gds\text{-get-pending } gds\ s)$
assumes *finish-refine*[*refine*]:
 $\llbracket a.\text{pre-finish } v\ s0\ s; (vi, v) \in V; (si, s) \in S \rrbracket$
 $\implies gds\text{-finish } gdsi\ vi\ si \leq \Downarrow S (gds\text{-finish } gds\ v\ s)$
assumes *cross-edge-refine*[*refine*]:
 $\llbracket a.\text{pre-cross-edge } u\ v\ s0\ s; (ui, u) \in V; (vi, v) \in V; (si, s) \in S \rrbracket$
 $\implies gds\text{-cross-edge } gdsi\ ui\ vi\ si \leq \Downarrow S (gds\text{-cross-edge } gds\ u\ v\ s)$
assumes *back-edge-refine*[*refine*]:
 $\llbracket a.\text{pre-back-edge } u\ v\ s0\ s; (ui, u) \in V; (vi, v) \in V; (si, s) \in S \rrbracket$
 $\implies gds\text{-back-edge } gdsi\ ui\ vi\ si \leq \Downarrow S (gds\text{-back-edge } gds\ u\ v\ s)$
assumes *discover-refine*[*refine*]:
 $\llbracket a.\text{pre-discover } u\ v\ s0\ s; (ui, u) \in V; (vi, v) \in V; (si, s) \in S \rrbracket$
 $\implies gds\text{-discover } gdsi\ ui\ vi\ si \leq \Downarrow S (gds\text{-discover } gds\ u\ v\ s)$

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

lemma *select-v0-refine*[*refine*]:
assumes *s-param*: $(si, s) \in S$
shows *SPEC* $(\lambda v0. v0 \in V0i \wedge \neg gds\text{-is-discovered } gdsi\ v0\ si)$
 $\leq \Downarrow V (\text{SPEC } (\lambda v0. v0 \in V0 \wedge \neg gds\text{-is-discovered } gds\ v0\ s))$
 $\langle \text{proof} \rangle$

lemma *gen-rwof-refine*:
assumes *NF*: *nofail* (*a.gen-dfs*)
assumes *RW*: *c.gen-rwof* *s*
obtains *s'* **where** $(s, s') \in S$ **and** *a.gen-rwof* *s'*
 $\langle \text{proof} \rangle$

lemma *gen-step-refine*[*refine*]: $(si, s) \in S \implies c.\text{gen-step } si \leq \Downarrow S (a.\text{gen-step-assert } s)$
 $\langle \text{proof} \rangle$

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*: *bijjective 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*]: $(ei, e) \in ES \implies gbs-init\ gbsi\ ei \leq \Downarrow S (gbs-init\ gbs\ e)$

assumes *gbs-new-root-refine*[*refine*]:

$\llbracket a.pre-new-root\ v0\ s; (v0i, v0) \in V; (si, s) \in S \rrbracket$

$\implies gbs-new-root\ gbsi\ v0i\ si \leq \Downarrow S (gbs-new-root\ gbs\ v0\ s)$

assumes *gbs-get-pending-refine*[*refine*]:

$\llbracket a.pre-get-pending\ s; (si, s) \in S \rrbracket$

$\implies gbs-get-pending\ gbsi\ si$

$\leq \Downarrow (V \times_r \langle V \rangle option-rel \times_r S) (gbs-get-pending\ gbs\ s)$

assumes *gbs-finish-refine*[*refine*]:

$\llbracket a.pre-finish\ v\ s0\ s; (vi, v) \in V; (si, s) \in S \rrbracket$

$\implies gbs-finish\ gbsi\ vi\ si \leq \Downarrow S (gbs-finish\ gbs\ v\ s)$

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

$\llbracket a.\text{pre-cross-edge } u \ v \ s0 \ s; (ui, u) \in V; (vi, v) \in V; (si, s) \in S \rrbracket$
 $\implies \text{gbs-cross-edge gbsi ui vi si} \leq \Downarrow S (\text{gbs-cross-edge gbs } u \ v \ s)$

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

$\llbracket a.\text{pre-back-edge } u \ v \ s0 \ s; (ui, u) \in V; (vi, v) \in V; (si, s) \in S \rrbracket$
 $\implies \text{gbs-back-edge gbsi ui vi si} \leq \Downarrow S (\text{gbs-back-edge gbs } u \ v \ s)$

assumes *gbs-discover-refine*[*refine*]:

$\llbracket a.\text{pre-discover } u \ v \ s0 \ s; (ui, u) \in V; (vi, v) \in V; (si, s) \in S \rrbracket$
 $\implies \text{gbs-discover gbsi ui vi si} \leq \Downarrow S (\text{gbs-discover gbs } u \ v \ s)$

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* ≤ $\Downarrow ES$ (*on-init param*)

assumes *is-break-param*[*param*]:

(*is-break parami, is-break param*) ∈ *S* → *bool-rel*

assumes *on-new-root-refine*[*refine*]:

$\llbracket a.\text{pre-new-root } v0 \ s; (v0i, v0) \in V; (si, s) \in S;$
 $(si', s') \in S; \text{nf-inres (gbs-new-root gbs } v0 \ s) \ s' \rrbracket$
 $\implies \text{on-new-root parami } v0i \ si' \leq \Downarrow ES (\text{on-new-root param } v0 \ s')$

assumes *on-finish-refine*[*refine*]:

$\llbracket a.\text{pre-finish } v \ s0 \ s; (vi, v) \in V; (si, s) \in S; (si', s') \in S;$
 $\text{nf-inres (gbs-finish gbs } v \ s) \ s' \rrbracket$
 $\implies \text{on-finish parami } vi \ si' \leq \Downarrow ES (\text{on-finish param } v \ s')$

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

$\llbracket a.\text{pre-cross-edge } u \ v \ s0 \ s; (ui, u) \in V; (vi, v) \in V; (si, s) \in S;$
 $(si', s') \in S; \text{nf-inres (gbs-cross-edge gbs } u \ v \ s) \ s' \rrbracket$
 $\implies \text{on-cross-edge parami } ui \ vi \ si' \leq \Downarrow ES (\text{on-cross-edge param } u \ v \ s')$

assumes *on-back-edge-refine*[*refine*]:

$\llbracket a.\text{pre-back-edge } u \ v \ s0 \ s; (ui, u) \in V; (vi, v) \in V; (si, s) \in S;$
 $(si', s') \in S; \text{nf-inres (gbs-back-edge gbs } u \ v \ s) \ s' \rrbracket$
 $\implies \text{on-back-edge parami } ui \ vi \ si' \leq \Downarrow ES (\text{on-back-edge param } u \ v \ s')$

assumes *on-discover-refine*[*refine*]:

$\llbracket a.\text{pre-discover } u \ v \ s0 \ s; (ui, u) \in V; (vi, v) \in V; (si, s) \in S;$
 $(si', s') \in S; \text{nf-inres (gbs-discover gbs } u \ v \ s) \ s' \rrbracket$
 $\implies \text{on-discover parami } ui \ vi \ si' \leq \Downarrow ES (\text{on-discover param } u \ v \ s')$

```

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 ≡ 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
    }
  }
end

```

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

tailrec-implT \equiv do {

$s \leftarrow$ *gds-init* *gds*;

FOREACHci

(λ it *s*.

gen-rwof *s*

$\wedge (\neg$ *gds-is-break* *gds* *s* \longrightarrow *gds-is-empty-stack* *gds* *s*)

$\wedge \forall 0-it \subseteq$ *gen-discovered* *s*)

$\forall 0$

(*Not o* *gds-is-break* *gds*)

($\lambda v0$ *s*. do {

let — ghost: *s0* = *s*;

if *gds-is-discovered* *gds* *v0* *s* *then*

RETURN *s*

else do {

$s \leftarrow$ *gds-new-root* *gds* *v0* *s*;

WHILEIT

(λs . *gen-rwof* *s* \wedge *insert* *v0* (*gen-discovered* *s0*) \subseteq *gen-discovered* *s*)

(λs . \neg *gds-is-break* *gds* *s* \wedge \neg *gds-is-empty-stack* *gds* *s*)

tr-impl-while-body *s*

}

}) *s*

}

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

tailrec-impl \equiv do {

$s \leftarrow$ *gds-init* *gds*;

FOREACHci

(λ it *s*.

gen-rwof *s*

$\wedge (\neg$ *gds-is-break* *gds* *s* \longrightarrow *gds-is-empty-stack* *gds* *s*)

$\wedge \forall 0-it \subseteq$ *gen-discovered* *s*)

$\forall 0$

(*Not o* *gds-is-break* *gds*)

($\lambda v0$ *s*. do {

let — ghost: *s0* = *s*;

if *gds-is-discovered* *gds* *v0* *s* *then*

RETURN *s*

else do {

$s \leftarrow$ *gds-new-root* *gds* *v0* *s*;

WHILEI

(λs . *gen-rwof* *s* \wedge *insert* *v0* (*gen-discovered* *s0*) \subseteq *gen-discovered* *s*)

(λs . \neg *gds-is-break* *gds* *s* \wedge \neg *gds-is-empty-stack* *gds* *s*)

(λs . do {

(*u*, *Vs*, *s*) \leftarrow *gds-get-pending* *gds* *s*;

case *Vs* of

None \Rightarrow *gds-finish* *gds* *u* *s*

```

    | Some v  $\Rightarrow$  do {
      if gds-is-discovered gds v s then do {
        if gds-is-finished gds v s then
          gds-cross-edge gds u v s
        else
          gds-back-edge gds u v s
      } else
        gds-discover gds u v s
    }
  } s
}
} s
}

```

end

Implementation of general DFS with outer foreach-loop

```

locale tailrec-impl =
  fb-graph G + gen-dfs gds V0 + tailrec-impl-defs G gds
  for G :: ('v, 'more) graph-rec-scheme
  and gds :: ('v, 's) gen-dfs-struct
  +
  assumes init-empty-stack:
    gds-init gds  $\leq_n$  SPEC (gds-is-empty-stack gds)
  assumes new-root-discovered:
     $\llbracket$ pre-new-root v0 s $\rrbracket$ 
     $\Rightarrow$  gds-new-root gds v0 s  $\leq_n$  SPEC ( $\lambda s'$ .
      insert v0 (gen-discovered s)  $\subseteq$  gen-discovered s')
  assumes get-pending-incr:
     $\llbracket$ pre-get-pending s $\rrbracket$   $\Rightarrow$  gds-get-pending gds s  $\leq_n$  SPEC ( $\lambda(-, -, s')$ .
      gen-discovered s  $\subseteq$  gen-discovered s'
      gen-discovered s  $\subseteq$  gen-discovered s'
  assumes finish-incr:  $\llbracket$ pre-finish u s0 s $\rrbracket$ 
     $\Rightarrow$  gds-finish gds u s  $\leq_n$  SPEC ( $\lambda s'$ .
      gen-discovered s  $\subseteq$  gen-discovered s')
  assumes cross-edge-incr: pre-cross-edge u v s0 s
     $\Rightarrow$  gds-cross-edge gds u v s  $\leq_n$  SPEC ( $\lambda s'$ .
      gen-discovered s  $\subseteq$  gen-discovered s')
  assumes back-edge-incr: pre-back-edge u v s0 s
     $\Rightarrow$  gds-back-edge gds u v s  $\leq_n$  SPEC ( $\lambda s'$ .
      gen-discovered s  $\subseteq$  gen-discovered s')
  assumes discover-incr: pre-discover u v s0 s
     $\Rightarrow$  gds-discover gds u v s  $\leq_n$  SPEC ( $\lambda s'$ .
      gen-discovered s  $\subseteq$  gen-discovered s')
begin

```

context

assumes *nofail*:

$nofail (gds-init\ gds \ggg WHILE\ gen-cond\ gen-step)$
begin
lemma *gds-init-refine: gds-init gds*
 $\leq SPEC (\lambda s. gen-rwof\ s \wedge gds-is-empty-stack\ gds\ s)$
 $\langle proof \rangle$

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

lemma *get-pending-nofail:*
assumes *A: pre-get-pending s*
shows *nofail (gds-get-pending gds s)*
 $\langle proof \rangle$

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

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

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

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

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

$\wedge \text{gen-discovered } s \subseteq \text{gen-discovered } s')$
 ⟨proof⟩

end

lemma *gen-step-disc-incr*:

assumes *nofail gen-dfs*

assumes *gen-rwof s insert v0 (gen-discovered s0) ⊆ gen-discovered s*

assumes $\neg \text{gds-is-break gds s } \neg \text{gds-is-empty-stack gds s}$

shows *gen-step s ≤ SPEC (λs. insert v0 (gen-discovered s0) ⊆ gen-discovered s)*
 ⟨proof⟩

theorem *tailrec-impl: tailrec-impl ≤ gen-dfs*
 ⟨proof⟩

lemma *tr-impl-while-body-gen-step*:

assumes [*simp*]: $\neg \text{gds-is-empty-stack gds s}$

shows *tr-impl-while-body s ≤ gen-step s*

⟨proof⟩

lemma *tailrecT-impl: tailrec-implT ≤ 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* :: $'s \Rightarrow 'v \text{ rel}$

fixes *stack* :: $'s \Rightarrow 'v \text{ list}$

fixes *choose-pending* :: $'v \Rightarrow 'v \text{ option} \Rightarrow 's \Rightarrow 's \text{ nres}$

begin

definition *gen-step' s* ≡ *do { ASSERT (gen-rwof s);*

if gds-is-empty-stack gds s then do {

v0 ← SPEC (λv0. v0 ∈ V0 ∧ ¬ gds-is-discovered gds v0 s);

gds-new-root gds v0 s

} else do {

```

let u = hd (stack s);
Vs ← SELECT (λv. (u,v) ∈ pending s);
s ← choose-pending u Vs s;
case Vs of
  None ⇒ gds-finish gds u s
| Some v ⇒
  if gds-is-discovered gds v s
  then if gds-is-finished gds v s then gds-cross-edge gds u v s
       else gds-back-edge gds u v s
  else gds-discover gds u v s
}}

```

definition $gen\text{-}dfs' \equiv gds\text{-}init\ gds \gg \text{WHILE } gen\text{-}cond\ gen\text{-}step'$
abbreviation $gen\text{-}rwof' \equiv rwof\ (gds\text{-}init\ gds)\ gen\text{-}cond\ gen\text{-}step'$

definition $rec\text{-}impl$ **where** $[DFS\text{-}code\text{-}unfold]$:

```

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'

```

```

     $\wedge (\neg \text{gds-is-break gds } s' \longrightarrow$ 
       $\text{stack } s' = \text{stack } s$ 
       $\wedge \text{pending } s' = (\text{pending } s - \{u\} \times (E''\{u\} - \text{it}))$ 
       $\wedge \text{gen-discovered } s' \supseteq \text{gen-discovered } s \cup (E''\{u\} - \text{it})$ 
     $)$ 
     $(E''\{u\}) (\lambda s. \neg \text{gds-is-break gds } s)$ 
     $(\lambda v s. \text{do } \{$ 
       $s \leftarrow \text{choose-pending } u \text{ (Some } v) s;$ 
       $\text{if gds-is-discovered gds } v \text{ } s \text{ then do } \{$ 
         $\text{if gds-is-finished gds } v \text{ } s \text{ then}$ 
           $\text{gds-cross-edge gds } u \text{ } v \text{ } s$ 
         $\text{else}$ 
           $\text{gds-back-edge gds } u \text{ } v \text{ } s$ 
         $\} \text{ else do } \{$ 
           $s \leftarrow \text{gds-discover gds } u \text{ } v \text{ } s;$ 
           $\text{if gds-is-break gds } s \text{ then RETURN } s \text{ else } D (v, s)$ 
         $\}$ 
       $\}$ 
     $)$ 
     $s;$ 
     $\text{if gds-is-break gds } s \text{ then}$ 
       $\text{RETURN } s$ 
     $\text{else do } \{$ 
       $s \leftarrow \text{choose-pending } u \text{ (None) } s;$ 
       $s \leftarrow \text{gds-finish gds } u \text{ } s;$ 
       $\text{RETURN } s$ 
     $\}$ 
   $\}) (v0, s)$ 
 $\}$ 
 $\}$ 
 $\}$ 
 $\}) s$ 
 $\}$ 

```

definition *rec-impl-for-paper* **where** *rec-impl-for-paper* $\equiv \text{do } \{$

```

   $s \leftarrow \text{gds-init gds};$ 
   $\text{FOREACHc } V0 \text{ (Not o gds-is-break gds) } (\lambda v0 s. \text{do } \{$ 
     $\text{if gds-is-discovered gds } v0 \text{ } s \text{ then RETURN } s$ 
     $\text{else do } \{$ 
       $s \leftarrow \text{gds-new-root gds } v0 \text{ } s;$ 
       $\text{if gds-is-break gds } s \text{ then RETURN } s$ 
       $\text{else do } \{$ 
         $\text{REC } (\lambda D (u, s). \text{do } \{$ 
           $s \leftarrow \text{FOREACHc } (E''\{u\}) (\lambda s. \neg \text{gds-is-break gds } s) (\lambda v s. \text{do } \{$ 
             $s \leftarrow \text{choose-pending } u \text{ (Some } v) s;$ 
             $\text{if gds-is-discovered gds } v \text{ } s \text{ then do } \{$ 
               $\text{if gds-is-finished gds } v \text{ } s \text{ then gds-cross-edge gds } u \text{ } v \text{ } s$ 
               $\text{else gds-back-edge gds } u \text{ } v \text{ } s$ 
             $\} \text{ else do } \{$ 
               $s \leftarrow \text{gds-discover gds } u \text{ } v \text{ } s;$ 
               $\text{if gds-is-break gds } s \text{ then RETURN } s \text{ else } D (v, s)$ 
             $\}$ 
           $\}$ 
         $\}$ 
       $\}$ 
     $\}$ 
   $\}$ 

```

```

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

```

end

locale *rec-impl* =

fb-graph *G* + *gen-dfs* *gds* *V0* + *rec-impl-defs* *G* *gds* *pending* *stack* *choose-pending*

for *G* :: ('v, 'more) *graph-rec-scheme*

and *gds* :: ('v, 's) *gen-dfs-struct*

and *pending* :: 's ⇒ 'v *rel*

and *stack* :: 's ⇒ 'v *list*

and *choose-pending* :: 'v ⇒ 'v *option* ⇒ 's ⇒ 's *nres*

+

assumes [*simp*]: *gds-is-empty-stack* *gds* *s* ⇔ *stack* *s* = []

assumes *init-spec*:

gds-init *gds* ≤_n *SPEC* (λs. *stack* *s* = [] ∧ *pending* *s* = {})

assumes *new-root-spec*:

[[*pre-new-root* *v0* *s*]]

⇒ *gds-new-root* *gds* *v0* *s* ≤_n *SPEC* (λs'.

stack *s'* = [*v0*] ∧ *pending* *s'* = {*v0*} × E "{*v0*}" ∧

gen-discovered *s'* = *insert* *v0* (*gen-discovered* *s*))

assumes *get-pending-fmt*: [[*pre-get-pending* *s*]] ⇒

do {

let *u* = *hd* (*stack* *s*);

vo ← *SELECT* (λv. (u, v) ∈ *pending* *s*);

s ← *choose-pending* *u* *vo* *s*;

RETURN (*u*, *vo*, *s*)

}

≤ *gds-get-pending* *gds* *s*

assumes *choose-pending-spec*: [[*pre-get-pending* *s*; *u* = *hd* (*stack* *s*);

case *vo* of

None ⇒ *pending* *s* "{*u*} = {}

| *Some* *v* ⇒ *v* ∈ *pending* *s* "{*u*}

]] ⇒

choose-pending *u* *vo* *s* ≤_n *SPEC* (λs'.

(*case vo of*
 None \Rightarrow pending $s' =$ pending s
 | Some $v \Rightarrow$ pending $s' =$ pending $s - \{(u,v)\} \wedge$
 stack $s' =$ stack $s \wedge$
 $(\forall x. \text{gds-is-discovered gds } x \ s' = \text{gds-is-discovered gds } x \ s)$
 ~~$(\forall x. \text{gds-is-discovered gds } x \ s' = \text{gds-is-discovered gds } x \ s)$~~
)
assumes *finish-spec*: $\llbracket \text{pre-finish } u \ s0 \ s \rrbracket$
 \Longrightarrow *gds-finish* $\text{gds } u \ s \leq_n \text{SPEC } (\lambda s'.$
 pending $s' =$ pending $s \wedge$
 stack $s' = \text{tl } (\text{stack } s) \wedge$
 $(\forall x. \text{gds-is-discovered gds } x \ s' = \text{gds-is-discovered gds } x \ s)$
assumes *cross-edge-spec*: *pre-cross-edge* $u \ v \ s0 \ s$
 \Longrightarrow *gds-cross-edge* $\text{gds } u \ v \ s \leq_n \text{SPEC } (\lambda s'.$
 pending $s' =$ pending $s \wedge$ stack $s' =$ stack $s \wedge$
 $(\forall x. \text{gds-is-discovered gds } x \ s' = \text{gds-is-discovered gds } x \ s)$
assumes *back-edge-spec*: *pre-back-edge* $u \ v \ s0 \ s$
 \Longrightarrow *gds-back-edge* $\text{gds } u \ v \ s \leq_n \text{SPEC } (\lambda s'.$
 pending $s' =$ pending $s \wedge$ stack $s' =$ stack $s \wedge$
 $(\forall x. \text{gds-is-discovered gds } x \ s' = \text{gds-is-discovered gds } x \ s)$
assumes *discover-spec*: *pre-discover* $u \ v \ s0 \ s$
 \Longrightarrow *gds-discover* $\text{gds } u \ v \ s \leq_n \text{SPEC } (\lambda s'.$
 pending $s' =$ pending $s \cup (\{v\} \times E''\{v\}) \wedge$ stack $s' = v\#\text{stack } s \wedge$
 gen-discovered $s' = \text{insert } v \ (\text{gen-discovered } s)$

begin

lemma *gen-step'-refine*:

$\llbracket \text{gen-rwof } s; \text{gen-cond } s \rrbracket \Longrightarrow \text{gen-step}' \ s \leq \text{gen-step } s$
 $\langle \text{proof} \rangle$

lemma *gen-dfs'-refine*: $\text{gen-dfs}' \leq \text{gen-dfs}$

$\langle \text{proof} \rangle$

lemma *gen-rwof'-imp-rwof*:

assumes *NF*: *nofail* *gen-dfs*

assumes *A*: *gen-rwof'* s

shows *gen-rwof* s

$\langle \text{proof} \rangle$

lemma *reachable-invar*:

gen-rwof' $s \Longrightarrow \text{set } (\text{stack } s) \subseteq \text{reachable} \wedge \text{pending } s \subseteq E$
 $\wedge \text{set } (\text{stack } s) \subseteq \text{gen-discovered } s \wedge \text{distinct } (\text{stack } s)$
 $\wedge \text{pending } s \subseteq \text{set } (\text{stack } s) \times \text{UNIV}$

$\langle \text{proof} \rangle$

lemma *mk-spec-aux*:

$\llbracket m \leq_n \text{SPEC } \Phi; m \leq \text{SPEC } \text{gen-rwof}' \rrbracket \implies m \leq \text{SPEC } (\lambda s. \text{gen-rwof}' s \wedge \Phi$
s)
 $\langle \text{proof} \rangle$

definition *post-choose-pending* u *vo* $s0$ $s \equiv$

$\text{gen-rwof}' s0$
 $\wedge \text{gen-cond } s0$
 $\wedge \text{stack } s0 \neq []$
 $\wedge u = \text{hd } (\text{stack } s0)$
 $\wedge \text{inres } (\text{choose-pending } u \text{ vo } s0) s$
 $\wedge \text{stack } s = \text{stack } s0$
 $\wedge (\forall x. \text{gds-is-discovered } \text{gds } x s = \text{gds-is-discovered } \text{gds } x s0)$
 ~~$\wedge \text{gds-is-not-root } \text{gds } s \neq \text{gds-is-not-root } \text{gds } s0$~~
 $\wedge (\text{case vo of}$
 $\text{None} \Rightarrow \text{pending } s0 \text{ `` } \{u\} = \{ \}$ $\wedge \text{pending } s = \text{pending } s0$
 $| \text{Some } v \Rightarrow v \in \text{pending } s0 \text{ `` } \{u\} \wedge \text{pending } s = \text{pending } s0 - \{(u,v)\})$

context

assumes *nofail*:

$\text{nofail } (\text{gds-init } \text{gds} \gg \text{WHILE } \text{gen-cond } \text{gen-step}')$

assumes *nofail2*:

$\text{nofail } (\text{gen-dfs})$

begin

lemma *pcp-imp-pgp*:

$\text{post-choose-pending } u \text{ vo } s0 s \implies \text{post-get-pending } u \text{ vo } s0 s$
 $\langle \text{proof} \rangle$

schematic-goal *gds-init-refine*: $?prop$

$\langle \text{proof} \rangle$

schematic-goal *gds-new-root-refine*:

$\llbracket \text{pre-new-root } v0 s; \text{gen-rwof}' s \rrbracket \implies \text{gds-new-root } \text{gds } v0 s \leq \text{SPEC } ?\Phi$
 $\langle \text{proof} \rangle$

schematic-goal *gds-choose-pending-refine*:

assumes 1: *pre-get-pending* s

assumes 2: *gen-rwof'* s

assumes [*simp*]: $u = \text{hd } (\text{stack } s)$

assumes 3: *case vo of*

$\text{None} \Rightarrow \text{pending } s \text{ `` } \{u\} = \{ \}$

$| \text{Some } v \Rightarrow v \in \text{pending } s \text{ `` } \{u\}$

shows *choose-pending* u *vo* $s \leq \text{SPEC } (\text{post-choose-pending } u \text{ vo } s)$

$\langle \text{proof} \rangle$

schematic-goal *gds-finish-refine*:

$\llbracket \text{pre-finish } u \text{ } s0 \text{ } s; \text{ post-choose-pending } u \text{ None } s0 \text{ } s \rrbracket \implies \text{gds-finish gds } u \text{ } s \leq$
 $\text{SPEC } ?\Phi$
 <proof>

schematic-goal *gds-cross-edge-refine*:
 $\llbracket \text{pre-cross-edge } u \text{ } v \text{ } s0 \text{ } s; \text{ post-choose-pending } u \text{ (Some } v) \text{ } s0 \text{ } s \rrbracket \implies \text{gds-cross-edge}$
 $\text{gds } u \text{ } v \text{ } s \leq \text{SPEC } ?\Phi$
 <proof>

schematic-goal *gds-back-edge-refine*:
 $\llbracket \text{pre-back-edge } u \text{ } v \text{ } s0 \text{ } s; \text{ post-choose-pending } u \text{ (Some } v) \text{ } s0 \text{ } s \rrbracket \implies \text{gds-back-edge}$
 $\text{gds } u \text{ } v \text{ } s \leq \text{SPEC } ?\Phi$
 <proof>

schematic-goal *gds-discover-refine*:
 $\llbracket \text{pre-discover } u \text{ } v \text{ } s0 \text{ } s; \text{ post-choose-pending } u \text{ (Some } v) \text{ } s0 \text{ } s \rrbracket \implies \text{gds-discover}$
 $\text{gds } u \text{ } v \text{ } s \leq \text{SPEC } ?\Phi$
 <proof>
end

lemma *rec-impl-aux*: $\llbracket xd \notin \text{Domain } P \rrbracket \implies P - \{y\} \times (\text{succ } y - \text{ita}) - \{(y,$
 $xd)\} - \{xd\} \times \text{UNIV} =$
 $P - \text{insert } (y, xd) (\{y\} \times (\text{succ } y - \text{ita}))$
 <proof>

lemma *rec-impl*: $\text{rec-impl} \leq \text{gen-dfs}$
 <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 × 'v set) list
on-stack :: 'v set

visited :: 'v set

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

lemma *simple-state-relI*:

assumes

dom (*finished* *s'*) = *dom* (*discovered* *s'*) - *set* (*stack* *s'*)
set (*stack* *s'*) \subseteq *dom* (*discovered* *s'*)
(*m'*, *state.more* *s'*) \in *erel*

shows (|

ss-stack = *map* ($\lambda u. (u, \text{pending } s' \text{ `` } \{u\})$) (*stack* *s'*),
on-stack = *set* (*stack* *s'*),
visited = *dom* (*discovered* *s'*),
... = *m'*

) , *s'*) \in (*erel*) *simple-state-rel*

<proof>

lemma *simple-state-more-refine*[*param*]:

(*simple-state.more-update*, *state.more-update*)
 \in (*R* \rightarrow *R*) \rightarrow *<R>* *simple-state-rel* \rightarrow *<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*

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

definition *is-empty-stack-impl* *s* \equiv (*ss-stack* *s* = [])

definition *is-discovered-impl* *u* *s* \equiv (*u* \in *visited* *s*)

definition *is-finished-impl* *u* *s* \equiv (*u* \in *visited* *s* - (*on-stack* *s*))

definition *finish-impl* *u* *s* \equiv *do* {

ASSERT (*ss-stack* *s* \neq [] \wedge *u* \in *on-stack* *s*);

let *s* = *s*(*ss-stack* := *tl* (*ss-stack* *s*));

let *s* = *s*(*on-stack* := *on-stack* *s* - {*u*});

RETURN *s*

}

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

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

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

definition *gbs* \equiv (
gbs-init = *init-impl*,
gbs-is-empty-stack = *is-empty-stack-impl* ,
gbs-new-root = *new-root-impl* ,
gbs-get-pending = *get-pending-impl* ,
gbs-finish = *finish-impl* ,
gbs-is-discovered = *is-discovered-impl* ,
gbs-is-finished = *is-finished-impl* ,
gbs-back-edge = $(\lambda u v s. \text{RETURN } s)$,
gbs-cross-edge = $(\lambda u v s. \text{RETURN } s)$,
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  $\langle$ proof $\rangle$ 

  definition get-pending s  $\equiv \bigcup$  (set (map ( $\lambda(u, Vs). \{u\} \times Vs$ ) (ss-stack s)))
  definition get-stack s  $\equiv$  map fst (ss-stack s)
  definition choose-pending
    :: 'v  $\Rightarrow$  'v option  $\Rightarrow$  ('v, 'd) simple-state-scheme  $\Rightarrow$  ('v, 'd) simple-state-scheme
  nres
    where [DFS-code-unfold]:
  choose-pending u vo s  $\equiv$ 
    case vo of
      None  $\Rightarrow$  RETURN s
    | Some v  $\Rightarrow$  do {
      ASSERT (ss-stack s  $\neq$  []);
      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  $\langle$ proof $\rangle$ 
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 $S = \langle ES \rangle \text{simple-state-rel}$
begin

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

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

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

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

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

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

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

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

lemma *discover-impl*:

$$\llbracket a.gen\text{-}dfs.pre\text{-}discover\ u\ v\ s0\ s;\ (ui,\ u)\in Id;\ (vi,\ v)\in Id;\ (si,\ s)\in \langle ES \rangle simple\text{-}state\text{-}rel \rrbracket$$

$$\implies c.discover\text{-}impl\ ui\ vi\ si \leq \Downarrow(\langle ES \rangle simple\text{-}state\text{-}rel)\ (RETURN\ (a.discover\ u\ v\ s))$$

$$\langle proof \rangle$$

sublocale *gen-param-dfs-refine*
where $gbsi = c.gbs$
and $gbs = a.gbs$
and $upd\text{-}exti = simple\text{-}state.more\text{-}update$
and $upd\text{-}ext = state.more\text{-}update$
and $V0i = a.V0$
and $V0 = a.V0$
and $V = Id$
and $S = \langle ES \rangle simple\text{-}state\text{-}rel$
 $\langle proof \rangle$

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

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

Link with tail-recursive implementation

sublocale *tailrec-impl G c.gds*
 $\langle proof \rangle$

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

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

Link to recursive implementation

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

sublocale *rec-impl G c.gds get-pending get-stack choose-pending*
 $\langle proof \rangle$

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

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* \equiv
 $\{((ss\text{-}stack\text{-}impl = si, ss\text{-}on\text{-}stack\text{-}impl = osi, ss\text{-}visited\text{-}impl = visi, \dots = mi),$
 $(ss\text{-}stack = s, on\text{-}stack = os, visited = vis, \dots = m)) \mid$
 $si\ osi\ visi\ mi\ s\ os\ vis\ m.$
 $(si, s) \in s\text{-}rel \wedge$
 $(osi, os) \in vis\text{-}rel \wedge$
 $(visi, vis) \in vis\text{-}rel \wedge$
 $(mi, m) \in erel$
 $\}$

consts

i-simple-state :: *interface* \Rightarrow *interface* \Rightarrow *interface* \Rightarrow *interface*

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

term *simple-state-ext*

lemma [autoref-rules, param]:

fixes *s-rel ps-rel vis-rel erel*

defines $R \equiv \langle s\text{-}rel, vis\text{-}rel, erel \rangle ss\text{-}impl\text{-}rel$

shows

$(ss\text{-}stack\text{-}impl, ss\text{-}stack) \in R \rightarrow s\text{-}rel$

$(ss\text{-}on\text{-}stack\text{-}impl, on\text{-}stack) \in R \rightarrow vis\text{-}rel$

$(ss\text{-}visited\text{-}impl, visited) \in R \rightarrow vis\text{-}rel$

$(simple\text{-}state\text{-}impl.more, simple\text{-}state.more) \in R \rightarrow erel$

$(ss\text{-}stack\text{-}impl\text{-}update, ss\text{-}stack\text{-}update) \in (s\text{-}rel \rightarrow s\text{-}rel) \rightarrow R \rightarrow R$

$(ss\text{-}on\text{-}stack\text{-}impl\text{-}update, on\text{-}stack\text{-}update) \in (vis\text{-}rel \rightarrow vis\text{-}rel) \rightarrow R \rightarrow R$

$(ss\text{-}visited\text{-}impl\text{-}update, visited\text{-}update) \in (vis\text{-}rel \rightarrow vis\text{-}rel) \rightarrow R \rightarrow R$

$(simple\text{-}state\text{-}impl.more\text{-}update, simple\text{-}state.more\text{-}update) \in (erel \rightarrow erel) \rightarrow R$
 $\rightarrow R$

$(simple\text{-}state\text{-}impl\text{-}ext, simple\text{-}state\text{-}ext) \in s\text{-}rel \rightarrow vis\text{-}rel \rightarrow vis\text{-}rel \rightarrow erel \rightarrow R$
<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* \equiv
 $\{((\text{ssnos-stack-impl} = si, \text{ssnos-visited-impl} = visi, \dots = mi),$
 $(\text{ss-stack} = s, \text{on-stack} = os, \text{visited} = vis, \dots = m)) \mid$
 $si\ visi\ mi\ s\ os\ vis\ m.$
 $(si, s) \in s\text{-rel} \wedge$
 $(visi, vis) \in vis\text{-rel} \wedge$
 $(mi, m) \in erel$
 $\}$

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

definition *op-nos-on-stack-update*
 $:: (-\ \text{set} \Rightarrow -\ \text{set}) \Rightarrow (-,-)\text{simple-state-scheme} \Rightarrow -$
where *op-nos-on-stack-update* \equiv *on-stack-update*

context begin interpretation *autoref-syn* \langle *proof* \rangle
lemma [*autoref-op-pat-def*]: *op-nos-on-stack-update f s*
 $\equiv OP\ (op\text{-nos-on-stack-update}\ f)\$s\ \langle$ *proof* \rangle

end

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

lemma [*autoref-rules, param*]:
fixes *s-rel vis-rel erel*
defines $R \equiv \langle s\text{-rel}, vis\text{-rel}, erel \rangle \text{ssnos-impl-rel}$
shows
 $(\text{ssnos-stack-impl}, \text{ss-stack}) \in R \rightarrow s\text{-rel}$
 $(\text{ssnos-visited-impl}, \text{visited}) \in R \rightarrow vis\text{-rel}$
 $(\text{simple-state-nos-impl.more}, \text{simple-state.more}) \in R \rightarrow erel$
 $(\text{ssnos-stack-impl-update}, \text{ss-stack-update}) \in (s\text{-rel} \rightarrow s\text{-rel}) \rightarrow R \rightarrow R$
 $(\lambda x. x, \text{op-nos-on-stack-update}\ f) \in R \rightarrow R$
 $(\text{ssnos-visited-impl-update}, \text{visited-update}) \in (vis\text{-rel} \rightarrow vis\text{-rel}) \rightarrow R \rightarrow R$
 $(\text{simple-state-nos-impl.more-update}, \text{simple-state.more-update}) \in (erel \rightarrow erel) \rightarrow$
 $R \rightarrow R$
 $(\lambda ns - ps\ \text{vs.}\ \text{simple-state-nos-impl-ext}\ ns\ ps\ \text{vs.}\ \text{simple-state-ext})$
 $\in s\text{-rel} \rightarrow ANY\text{-rel} \rightarrow vis\text{-rel} \rightarrow erel \rightarrow R$
 \langle *proof* \rangle

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* =
 $\text{ssns-visited-impl} :: 'nsi$

definition [*to-relAPP*]: *ssns-impl-rel* ($R::('a \times 'b)\ \text{set}$) *vis-rel erel* \equiv

```

{((ssns-visited-impl = visi, ... = mi),
  (ss-stack = s, on-stack = os, visited = vis, ... = m)) |
  visi mi s os vis m.
  (visi, vis) ∈ 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 $R \equiv \langle ANY\text{-rel}, vis\text{-rel}, erel \rangle ssns\text{-impl-rel}$

shows

$(ssns\text{-visited-impl}, visited) \in R \rightarrow vis\text{-rel}$

$(simple\text{-state-ns-impl.more}, simple\text{-state.more}) \in R \rightarrow erel$

$\bigwedge f. (\lambda x. x, op\text{-ns-stack-update } f) \in R \rightarrow R$

$\bigwedge f. (\lambda x. x, op\text{-ns-on-stack-update } f) \in R \rightarrow R$

$(ssns\text{-visited-impl-update}, visited\text{-update}) \in (vis\text{-rel} \rightarrow vis\text{-rel}) \rightarrow R \rightarrow R$

$(simple\text{-state-ns-impl.more-update}, simple\text{-state.more-update}) \in (erel \rightarrow erel) \rightarrow R \rightarrow R$

$(\lambda - ps\ vs. simple\text{-state-ns-impl-ext } ps\ vs, simple\text{-state-ext})$

$\in ANY1\text{-rel} \rightarrow ANY2\text{-rel} \rightarrow vis\text{-rel} \rightarrow erel \rightarrow R$

<proof>

lemma [*refine-transfer-post-simp*]:

$\bigwedge a\ m. a(simple\text{-state-nos-impl.more} := m::unit) = a$

$\bigwedge a\ m. a(simple\text{-state-impl.more} := m::unit) = a$

$\bigwedge a\ m. a(simple\text{-state-ns-impl.more} := m::unit) = a$

<proof>

end

1.8 Restricting Nodes by Pre-Initializing Visited Set

theory *Restr-Impl*

imports *Simple-Impl*

begin

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

We now further refine the simple implementation in case that the graph has the form $G' = (rel\text{-restrict } E\ R, V0 - R)$ for some *fb-graph* $G = (E, V0)$. If, additionally, the parameterization is not "too sensitive" to the visited set, we can pre-initialize the visited set with R , and use the $V0$ and E of G . This may be a more efficient implementation than explicitly 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 $restr\text{-rel-stack}D: (ss\text{-stack } s, ss\text{-stack } s') \in \langle Id \times_r \{(U, U'). U-R = U'\} \rangle list\text{-rel}$
and $restr\text{-rel-vis-dj}D: visited\ s' \cap R = \{\}$
 $\langle proof \rangle$

context *fixes* $R :: 'v\ set$ **begin**

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

end

lemma *restr-simple-state-rel-combine*:

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

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* =

$graph\text{-defs } G +$
 $a: simple\text{-impl-defs } graph\text{-restrict } G\ R$
for $G :: ('v, 'more) graph\text{-rec-scheme}$
and R

begin

sublocale *pre-simple-impl* $G \langle proof \rangle$

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

definition $gbs' \equiv gbs \ \emptyset$

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*:
 [$(s, s') \in \text{restr-rel } R$]
 $\implies \text{is-break parami } s \longleftrightarrow \text{is-break parami } s'$

assumes *on-new-root-refine*:
 [$(s, s') \in \text{restr-rel } R$]
 $\implies \text{on-new-root parami } v0 s \leq \text{on-new-root parami } v0 s'$

assumes *on-finish-refine*:
 [$(s, s') \in \text{restr-rel } R$]
 $\implies \text{on-finish parami } u s \leq \text{on-finish parami } u s'$

assumes *on-discover-refine*:
 [$(s, s') \in \text{restr-rel } R$]
 $\implies \text{on-discover parami } u v s \leq \text{on-discover parami } u v s'$

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*) $\in \text{rel} \rightarrow \text{bool-rel}$
 <proof>

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

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

lemma *cross-back-id*[*simp*]:
 $do\text{-}cross\text{-}edge\ u\ v\ s = RETURN\ s$
 $do\text{-}back\text{-}edge\ u\ v\ s = RETURN\ s$
 $a.c.do\text{-}cross\text{-}edge\ u\ v\ s = RETURN\ s$
 $a.c.do\text{-}back\text{-}edge\ u\ v\ s = RETURN\ s$
 ⟨*proof*⟩

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

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

lemma *do-new-root-refine*[*refine*]:
 $\llbracket (v0i, v0) \in Id; (si, s) \in rel; v0 \notin R \rrbracket$
 $\implies do\text{-}new\text{-}root\ v0i\ si \leq \Downarrow rel (a.c.do\text{-}new\text{-}root\ v0\ s)$
 ⟨*proof*⟩

lemma *do-finish-refine*[*refine*]:
 $\llbracket (s, s') \in rel; (u, u') \in Id \rrbracket$
 $\implies do\text{-}finish\ u\ s \leq \Downarrow rel (a.c.do\text{-}finish\ u'\ s')$
 ⟨*proof*⟩

lemma *aux-cnv-pending*:
 $\llbracket (s, s') \in rel;$
 $\neg is\text{-}empty\text{-}stack\text{-}impl\ s; vs \in Vs; vs \notin R;$
 $hd\ (ss\text{-}stack\ s) = (u, Vs) \rrbracket \implies$
 $hd\ (ss\text{-}stack\ s') = (u, insert\ vs\ (Vs - R))$
 ⟨*proof*⟩

lemma *get-pending-refine*:
assumes $(s, s') \in rel\ gen\text{-}cond\ s \neg is\text{-}empty\text{-}stack\text{-}impl\ s$
shows
 $get\text{-}pending\text{-}impl\ s \leq (sup$

```

    (↓(Id ×r ⟨Id⟩option-rel ×r rel) (inf
      (get-pending-impl s')
      (SPEC (λ(-, Vs, -). case Vs of None ⇒ True | Some v ⇒ v ∈ R))))
    (↓(Id ×r ⟨Id⟩option-rel ×r rel) (
      SPEC (λ(u, Vs, s'). ∃ v. Vs = Some v ∧ v ∈ R ∧ s'' = s')
    )))
  ⟨proof⟩

```

lemma *do-discover-refine*[*refine*]:
 $\llbracket (s, s') \in \text{rel}; (u, u') \in \text{Id}; (v, v') \in \text{Id}; v' \notin R \rrbracket$
 $\implies \text{do-discover } u \ v \ s \leq \downarrow \text{rel } (\text{a.c.do-discover } u' \ v' \ s')$
 ⟨proof⟩

lemma *aux-R-node-discovered*: $\llbracket (s, s') \in \text{rel}; v \in R \rrbracket \implies \text{is-discovered-impl } v \ s$
 ⟨proof⟩

lemma *re-refine-aux*: $\text{gen-dfs} \leq \downarrow \text{rel a.c.gen-dfs}$
 ⟨proof⟩

theorem *re-refine-aux2*: $\text{gen-dfs} \leq \downarrow (\text{rel } O \langle ES \rangle \text{simple-state-rel}) \text{ a.a.it-dfs}$
 ⟨proof⟩

theorem *re-refine*: $\text{gen-dfs} \leq \downarrow (\langle ES \rangle \text{restr-simple-state-rel } R) \text{ a.a.it-dfs}$
 ⟨proof⟩

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

lemma *tailrec-refine*: $\text{tailrec-impl} \leq \downarrow (\langle ES \rangle \text{restr-simple-state-rel } R) \text{ a.a.it-dfs}$
 ⟨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*: $state.more\ s = state.more\ s' \implies break\ s = break\ s'$
 ⟨*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 \longleftrightarrow 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 ($\lambda s. break\ s \longleftrightarrow back-edges\ s \neq \{\}$)*
 ⟨*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 ($\lambda s. break\ s \longleftrightarrow back-edges\ s \neq \{\}$)
 ⟨*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*: $\neg \text{cond } s$
shows $\text{break } s \longleftrightarrow \neg \text{acyclic } (E \cap \text{reachable} \times \text{UNIV})$
 $\langle \text{proof} \rangle$

Again, we have a short two-line proof:

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

end

Finally, we define a specification for cyclicity checking, and prove that our cyclicity checker satisfies the specification:

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

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

The same for the total correct variant:

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

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

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* \equiv {
 (\langle *cycc-state-impl.break* = *b* \rangle), (\langle *cycc-state.break* = *b* \rangle) | *b. True* }

abbreviation *cycc-rel* \equiv \langle *cycc-erel* \rangle *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*

\equiv *dflt-parameterization simple-state.more* (*RETURN* (\langle *break* = *False* \rangle)) (\langle
on-back-edge := $\lambda u v s.$ *RETURN* (\langle *break* = *True* \rangle),
is-break := *break* \rangle)

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*) \in *cycc-rel*

\impl *cycc-state-impl.break si* = *cycc-state.break s*

\langle *proof* \rangle

interpretation *cycc-impl*: *simple-impl-defs G cycc-params-impl cycc-params*
for *G* \langle *proof* \rangle

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* \equiv *do* {
ASSERT (*fb-graph G*);
s \leftarrow *cycc-impl.tailrec-impl TYPE('a) G*;
RETURN (*break s*)
 $\}$

definition [*DFS-code-unfold*]: *cyc-checker-rec-impl G* \equiv *do* {
ASSERT (*fb-graph G*);
s \leftarrow *cycc-impl.rec-impl TYPE('a) G*;
RETURN (*break s*)
 $\}$

definition [*DFS-code-unfold*]: *cyc-checker-implT G* \equiv *do* {
ASSERT (*graph G* \wedge *finite (graph-defs.reachable G)*);

```

  s ← cycc-impl.tailrec-implT TYPE('a) G;
  RETURN (break s)
}

```

To show correctness of the implementation, we integrate the locale of the simple implementation into our cyclicity checker's locale:

```

context cycc begin
  sublocale simple-impl G cycc-params cycc-params-impl cycc-erel
  ⟨proof⟩

```

We get that our implementation refines the abstract DFS algorithm.

```

lemmas impl-refine = simple-tailrec-refine simple-rec-refine simple-tailrecT-refine

```

Unfortunately, the combination of locales and abbreviations gets to its limits here, so we state the above lemma a bit more readable:

```

lemma
  cycc-impl.tailrec-impl TYPE('more) G ≤ ↓ cycc-rel it-dfs
  cycc-impl.rec-impl TYPE('more) G ≤ ↓ cycc-rel it-dfs
  cycc-impl.tailrec-implT TYPE('more) G ≤ ↓ cycc-rel it-dfsT
  ⟨proof⟩

```

end

Finally, we get correctness of our cyclicity checker implementations

```

lemma cycc-checker-impl-refine: cycc-checker-impl G ≤ ↓Id (cycc-checker G)
  ⟨proof⟩

```

```

lemma cycc-checker-rec-impl-refine:
  cycc-checker-rec-impl G ≤ ↓Id (cycc-checker G)
  ⟨proof⟩

```

```

lemma cycc-checker-implT-refine: cycc-checker-implT G ≤ ↓Id (cycc-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 {
 $(\langle \text{break-impl} = bi, \dots = mi \rangle, \langle \text{break} = b, \dots = m \rangle) \mid bi\ mi\ b\ m.$
 $(bi, b) \in \text{bool-rel} \wedge (mi, m) \in \text{erel}$ }

And register it with the Autoref tool:

consts

i-cycc-state-ext :: *interface* \Rightarrow *interface*

lemmas [*autoref-rel-intf*] = *REL-INTFI*[of *cycc-state-erel i-cycc-state-ext*]

We show that the record operations on our extended state are parametric, and declare these facts to Autoref:

lemma [*autoref-rules*]:

fixes *ns-rel vis-rel erel*

defines $R \equiv \langle \text{ns-rel}, \text{vis-rel}, \langle \text{erel} \rangle \text{cycc-state-erel} \rangle \text{ss-impl-rel}$

shows

$(\text{cycc-state-impl}'\text{-ext}, \text{cycc-state-impl-ext}) \in \text{bool-rel} \rightarrow \text{erel} \rightarrow \langle \text{erel} \rangle \text{cycc-state-erel}$

$(\text{break-impl}, \text{cycc-state-impl.break}) \in R \rightarrow \text{bool-rel}$

$\langle \text{proof} \rangle$

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

schematic-goal *cyc-checker-impl*:

defines $V \equiv \text{Id} :: ('v \times 'v :: \text{hashable}) \text{set}$

assumes [*unfolded V-def, autoref-rules*]:

$(Gi, G) \in \langle \text{Rm}, V \rangle \text{g-impl-rel-ext}$

notes [*unfolded V-def, autoref-tyrel*] =

TYRELI[**where** $R = \langle V \rangle \text{dflt-ahs-rel}$]

TYRELI[**where** $R = \langle V \times_r \langle V \rangle \text{list-set-rel} \rangle \text{ras-rel}$]

shows $\text{nres-of } (?c :: ?'c \text{ dres}) \leq \Downarrow ?R (\text{cyc-checker-impl } G)$

$\langle \text{proof} \rangle$

concrete-definition *cyc-checker-code* **uses** *cyc-checker-impl*

export-code *cyc-checker-code* **checking** *SML*

Combining the refinement steps yields a correctness theorem for the cyclicity checker implementation:

theorem *cyc-checker-code-correct*:

assumes 1: *fb-graph* G

assumes 2: $(Gi, G) \in \langle \text{Rm}, \text{Id} \rangle \text{g-impl-rel-ext}$

assumes 4: *cyc-checker-code* $Gi = \text{dRETURN } x$

shows $x \longleftrightarrow (\neg \text{acyclic } (g\text{-E } G \cap ((g\text{-E } G)^* \text{ “ } g\text{-V0 } G) \times \text{UNIV}))$

$\langle \text{proof} \rangle$

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

schematic-goal *cyc-checker-rec-impl*:

defines $V \equiv \text{Id} :: ('v \times 'v :: \text{hashable}) \text{set}$

assumes [*unfolded V-def, autoref-rules*]:

$(Gi, G) \in \langle \text{Rm}, V \rangle \text{g-impl-rel-ext}$

```

notes [unfolding 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  $x \longleftrightarrow (\neg \text{acyclic } (g-E G \cap ((g-E G)^* \text{ `` } g-V0 G) \times UNIV))$ 
 $\langle$ 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 [unfolding V-def, autoref-rules]:
     $(Gi, G) \in \langle Rm, V \rangle g$ -impl-rel-ext
  notes [unfolding 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  $\longleftrightarrow (\neg \text{acyclic } (g-E G \cap ((g-E G)^* \text{ `` } g-V0 G) \times UNIV))$ 
 $\langle$ 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*]: $s(\text{state.more} := (\text{ppath} = \text{foo})) = s(\text{ppath} := \text{foo})$
 ⟨*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 = $\lambda v0 s.$ if *P v0* then *RETURN (a-path [] v0)* else *RETURN no-path*,
on-discover = $\lambda u v s.$ if *P v*
 then — *v* is already on the stack, so we need to pop it again
 RETURN (a-path (rev (tl (stack s))) v)
 else *RETURN no-path*,
on-finish = $\lambda u s.$ *RETURN (state.more s)*,
on-back-edge = $\lambda u v s.$ *RETURN (state.more s)*,
on-cross-edge = $\lambda u v s.$ *RETURN (state.more s)*,
is-break = $\lambda s.$ *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 (λs. ppath s = None → dom (discovered s) ∩ Collect P = {})
    ⟨proof⟩

  lemma i-path-to-P:
    is-invar (λs. ppath s = Some (vs,v) → P v)
    ⟨proof⟩

  lemma i-path-invar:
    is-invar (λs. ppath s = Some (vs,v) →
      (vs ≠ [] → hd vs ∈ V0 ∧ path E (hd vs) vs v)
      ∧ (vs = [] → v ∈ V0 ∧ path E v vs v)
      ∧ (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 $hd\ vs \in V0\ path\ E\ (hd\ vs)\ vs\ v$
 $\langle proof \rangle$

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

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

end

context *fp0* **begin**

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

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* $G\ P \equiv \lambda r. case\ r\ of$

$None \Rightarrow (g\text{-}E\ G)^* \text{ `` } g\text{-}V0\ G \cap Collect\ P = \{ \}$
 $| Some\ (vs, v) \Rightarrow P\ v \wedge distinct\ (vs@[v]) \wedge (\exists v0 \in g\text{-}V0\ G. path\ (g\text{-}E\ G)\ v0\ vs\ v)$

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 ⇒ ('v ⇒ bool) ⇒ 'v fp-result nres
where find-path0 G P ≡ do {
  ASSERT (fp0 G);
  s ← fp0.it-dfs TYPE('more) G P;
  RETURN (ppath s)
}

```

lemma *find-path0-correct*:
shows *find-path0 G P ≤ find-path0-spec G P*
<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 ≡ E “ R ⊆ R ∧ R ∩ Collect P = {}

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

lemma *restr-invar-imp-not-reachable*: *restr-invar E R P ⇒ E* “R ∩ Collect P = {}*
<proof>

type-synonym *'v fpr-result* = *'v set + ('v list × 'v)*

definition *find-path0-restr-pred G P R ≡ λr.*

case r of

Inl R' ⇒ R' = R ∪ (g-E G) “ g-V0 G ∧ restr-invar (g-E G) R' P*

| Inr (vs,v) ⇒ P v ∧ (∃ v0 ∈ 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 ≡ do {*

ASSERT (fb-graph G ∧ 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*
 $:: ('v, 'more) \text{ graph-rec-scheme} \Rightarrow ('v \Rightarrow \text{bool}) \Rightarrow 'v \text{ set} \Rightarrow 'v \text{ fpr-result nres}$
where *find-path0-restr* $G P R \equiv \text{do} \{$
 ASSERT (*fb-graph* G);
 ASSERT (*fp0* (*graph-restrict* $G R$));
 $s \leftarrow \text{fp0.it-dfs TYPE}('more) (\text{graph-restrict } G R) P;$
 case *ppath* s *of*
 None $\Rightarrow \text{do} \{$
 ASSERT (*dom* (*discovered* s) = *dom* (*finished* s));
 RETURN (*Inl* ($R \cup \text{dom} (\text{finished } s)$))
 $\}$
 | *Some* (vs, v) $\Rightarrow \text{RETURN} (\text{Inr } (vs, v))$
 $\}$

lemma *find-path0-restr-correct*:
shows *find-path0-restr* $G P R \leq \text{find-path0-restr-spec } G P R$
<proof>

2.2.3 Path of Minimal Length One, with Restriction

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

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* $G P R \equiv \text{do} \{$
 ASSERT (*fb-graph* $G \wedge \text{restr-invar } (g-E G) R P$);
 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*
 $:: ('v, 'more) \text{ graph-rec-scheme} \Rightarrow ('v \Rightarrow \text{bool}) \Rightarrow 'v \text{ set} \Rightarrow 'v \text{ fpr-result nres}$
where *find-path1-restr* $G P R \equiv$
 FOREACHc ($g-V0 G$) *is-Inl* ($\lambda v0 s. \text{do} \{$
 ASSERT (*is-Inl* s); — TODO: Add FOREACH-condition as precondition in
 autoref!
 let $R = \text{projl } s;$
 $f0 \leftarrow \text{find-path0-restr-spec } (G \langle \langle g-V0 := g-E G \text{ `` } \{v0\} \rangle \rangle) P R;$
 case $f0$ *of*

```

    Inl - => RETURN f0
  | Inr (vs,v) => RETURN (Inr (v0#vs,v))
} (Inl R)

```

definition *find-path1-tailrec-invar* $G P R0$ *it s* \equiv

```

  case s of
    Inl R => R = R0  $\cup$  (g-E G)+ “ (g-V0 G - it)  $\wedge$  restr-invar (g-E G) R P
  | Inr (vs, v) => 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* $G P R$
 \langle *proof* \rangle

definition *find-path1-pred* $G P \equiv \lambda r.$

```

  case r of
    None => (g-E G)+ “ g-V0 G  $\cap$  Collect P = {}
  | Some (vs, v) => 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 => ('v => bool) => 'v fp-result nres
where find-path1  $G P \equiv$  do {
  r  $\leftarrow$  find-path1-restr-spec G P {};
  case r of
    Inl - => RETURN None
  | Inr vsv => RETURN (Some vsv)
}

```

lemma *find-path1-correct*:

shows *find-path1* $G P \leq$ *find-path1-spec* $G P$
 \langle *proof* \rangle

2.2.5 Implementation

record 'v fp0-state-impl = 'v simple-state +
 ppath :: ('v list \times 'v) option

definition $fp0\text{-erel} \equiv \{ (\ () \text{ fp0-state-impl.ppath} = p \), (\ () \text{ fp0-state.ppath} = p \)) \mid p. \text{ True} \}$

abbreviation $fp0\text{-rel } R \equiv \langle fp0\text{-erel} \rangle \text{restr-simple-state-rel } R$

abbreviation $no\text{-path-impl} \equiv (\ () \text{ fp0-state-impl.ppath} = \text{None} \)$

abbreviation $a\text{-path-impl } p \ v \equiv (\ () \text{ fp0-state-impl.ppath} = \text{Some } (p, v) \)$

lemma $fp0\text{-rel-ppath-cong}[simp]$:

$(s, s') \in fp0\text{-rel } R \implies fp0\text{-state-impl.ppath } s = fp0\text{-state.ppath } s'$
 $\langle \text{proof} \rangle$

lemma $fp0\text{-ss-rel-ppath-cong}[simp]$:

$(s, s') \in \langle fp0\text{-erel} \rangle \text{simple-state-rel} \implies fp0\text{-state-impl.ppath } s = fp0\text{-state.ppath } s'$
 $\langle \text{proof} \rangle$

lemma $fp0i\text{-cong}[cong]$: $\text{simple-state.more } s = \text{simple-state.more } s'$

$\implies fp0\text{-state-impl.ppath } s = fp0\text{-state-impl.ppath } s'$
 $\langle \text{proof} \rangle$

lemma $fp0\text{-erelI}$: $p = p'$

$\implies (\ () \text{ fp0-state-impl.ppath} = p \), (\ () \text{ fp0-state.ppath} = p' \)) \in fp0\text{-erel}$
 $\langle \text{proof} \rangle$

definition $fp0\text{-params-impl}$

$:: - \Rightarrow ('v, 'v \text{ fp0-state-impl}, ('v, \text{unit}) \text{fp0-state-impl-ext}) \text{ gen-parameterization}$

where $fp0\text{-params-impl } P \equiv (\$

$\text{on-init} = \text{RETURN } no\text{-path-impl},$

$\text{on-new-root} = \lambda v \ 0 \ s.$

$\text{if } P \ v \ 0 \ \text{then } \text{RETURN } (a\text{-path-impl } [] \ v \ 0) \ \text{else } \text{RETURN } no\text{-path-impl},$

$\text{on-discover} = \lambda u \ v \ s.$

$\text{if } P \ v \ \text{then } \text{RETURN } (a\text{-path-impl } (\text{map } \text{fst } (\text{rev } (\text{tl } (\text{CAST } (\text{ss-stack } s)))))) \ v$
 $\text{else } \text{RETURN } no\text{-path-impl},$

$\text{on-finish} = \lambda u \ s. \ \text{RETURN } (\text{simple-state.more } s),$

$\text{on-back-edge} = \lambda u \ v \ s. \ \text{RETURN } (\text{simple-state.more } s),$

$\text{on-cross-edge} = \lambda u \ v \ s. \ \text{RETURN } (\text{simple-state.more } s),$

$\text{is-break} = \lambda s. \ \text{ppath } s \neq \text{None} \)$

lemmas $fp0\text{-params-impl-simp}[simp, \text{DFS-code-unfold}]$

$= \text{gen-parameterization.simps}[\text{mk-record-simp}, \text{OF } fp0\text{-params-impl-def}]$

interpretation $fp0\text{-impl}$:

$\text{restricted-impl-defs } fp0\text{-params-impl } P \ \text{fp0-params } P \ G \ R$

for $G \ P \ R \ \langle \text{proof} \rangle$

locale $fp0\text{-restr} = \text{fb-graph}$

begin

sublocale $fp0?$: $fp0 \ \text{graph-restrict } G \ R$

$\langle \text{proof} \rangle$

sublocale *impl*: *restricted-impl* G *fp0-params* P *fp0-params-impl* P
fp0-erel R
 \langle *proof* \rangle
end

definition *find-path0-restr-impl* G P $R \equiv$ *do* {
ASSERT (*fb-graph* G);
ASSERT (*fp0* (*graph-restrict* G R));
 $s \leftarrow$ *fp0-impl.tailrec-impl* *TYPE*('a) G R P ;
case *ppath* s *of*
 $None \Rightarrow$ *RETURN* (*Inl* (*visited* s))
 $|$ *Some* (vs, v) \Rightarrow *RETURN* (*Inr* (vs, v))
}

lemma *find-path0-restr-impl*[*refine*]:
shows *find-path0-restr-impl* G P R
 \leq \Downarrow ($\langle Id, Id \times_r Id \rangle$ *sum-rel*)
(*find-path0-restr* G P R)
 \langle *proof* \rangle

definition *find-path0-impl* G $P \equiv$ *do* {
ASSERT (*fp0* G);
 $s \leftarrow$ *fp0-impl.tailrec-impl* *TYPE*('a) G {} P ;
RETURN (*ppath* s)
}

lemma *find-path0-impl*[*refine*]: *find-path0-impl* G P
 \leq \Downarrow ($\langle Id \times_r Id \rangle$ *option-rel*) (*find-path0* G P)
 \langle *proof* \rangle

2.2.6 Synthesis of Executable Code

record ($'v, 'si, 'nsi$)*fp0-state-impl'* = ($'si, 'nsi$)*simple-state-nos-impl* +
ppath-impl :: ($'v$ *list* \times $'v$) *option*

definition [*to-relAPP*]: *fp0-state-erel* *erel* \equiv {
 (\langle *ppath-impl* = $pi, \dots = mi$ \rangle, \langle *ppath* = $p, \dots = m$ \rangle) | pi mi p m .
 (pi, p) \in ($\langle Id \rangle$ *list-rel* \times_r *Id*) *option-rel* \wedge (mi, m) \in *erel*}
}

consts
i-fp0-state-ext :: *interface* \Rightarrow *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 $R \equiv \langle ns\text{-rel}, vis\text{-rel}, \langle erel \rangle fp0\text{-state-erel} \rangle ssnos\text{-impl-rel}$
shows
 $(fp0\text{-state-impl}'\text{-ext}, fp0\text{-state-impl-ext})$
 $\in \langle \langle Id \rangle list\text{-rel} \times_r Id \rangle option\text{-rel} \rightarrow erel \rightarrow \langle erel \rangle fp0\text{-state-erel}$
 $(ppath\text{-impl}, fp0\text{-state-impl.ppath}) \in R \rightarrow \langle \langle Id \rangle list\text{-rel} \times_r Id \rangle option\text{-rel}$
 $\langle proof \rangle$

schematic-goal *find-path0-code:*

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

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 $(\lambda G P. nres\text{-of} (find\text{-path0-code} G P), find\text{-path0-spec})$
 $\in \langle Rm, Rv \rangle g\text{-impl-rel-ext} \rightarrow (Rv \rightarrow bool\text{-rel})$
 $\rightarrow \langle \langle \langle Rv \rangle list\text{-rel} \times_r Rv \rangle option\text{-rel} \rangle nres\text{-rel}$
 $\langle proof \rangle$

lemmas *find-path0-autoref*[*autoref-rules*] = *find-path0-autoref-aux*[*OF PREFER-id-D*]

schematic-goal *find-path0-restr-code:*

fixes $vis\text{-rel} :: ('v \times 'v) set \Rightarrow ('visi \times 'v) set\ set$
notes [*autoref-rel-intf*] = $REL\text{-INTFI}[of\ vis\text{-rel}\ i\text{-set}\ \mathbf{for}\ I]$
assumes [*autoref-rules*]: $(op\text{-vis-insert}, insert) \in Id \rightarrow \langle Id \rangle vis\text{-rel} \rightarrow \langle Id \rangle vis\text{-rel}$
assumes [*autoref-rules*]: $(op\text{-vis-memb}, (\in)) \in Id \rightarrow \langle Id \rangle vis\text{-rel} \rightarrow bool\text{-rel}$
assumes [*autoref-rules*]:
 $(Gi, G) \in \langle Rm, Id \rangle g\text{-impl-rel-ext}$
 $(Pi, P) \in Id \rightarrow bool\text{-rel}$
 $(Ri, R) \in \langle Id \rangle vis\text{-rel}$
shows $(nres\text{-of} (?c :: ?'c\ dres),$
 $find\text{-path0-restr-impl}$
 G
 P
 $(R :: :_r \langle Id \rangle vis\text{-rel})) \in ?R$
 $\langle proof \rangle$

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: $(op\text{-}vis\text{-}insert, insert) \in Rv \rightarrow \langle Rv \rangle vis\text{-}rel \rightarrow \langle Rv \rangle vis\text{-}rel$

assumes 2: $(op\text{-}vis\text{-}memb, (\in)) \in Rv \rightarrow \langle Rv \rangle vis\text{-}rel \rightarrow bool\text{-}rel$

assumes *Vid*: $Rv = Id$

shows $(\lambda G P R. nres\text{-}of (find\text{-}path0\text{-}restr\text{-}code\ op\text{-}vis\text{-}insert\ op\text{-}vis\text{-}memb\ G\ P\ R),$

find-path0-restr-spec)

$\in \langle Rm, Rv \rangle g\text{-}impl\text{-}rel\text{-}ext \rightarrow (Rv \rightarrow bool\text{-}rel) \rightarrow \langle Rv \rangle vis\text{-}rel \rightarrow$

$\langle \langle \langle Rv \rangle vis\text{-}rel, \langle Rv \rangle list\text{-}rel \times_r Rv \rangle sum\text{-}rel \rangle nres\text{-}rel$

<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* :: $('v \times 'v)\ set \Rightarrow ('visi \times 'v)\ set\ set$

notes [*autoref-rel-intf*] = *REL-INTFI*[*of vis-rel i-set for I*]

assumes [*autoref-rules*]: $(op\text{-}vis\text{-}insert, insert) \in Id \rightarrow \langle Id \rangle vis\text{-}rel \rightarrow \langle Id \rangle vis\text{-}rel$

assumes [*autoref-rules*]: $(op\text{-}vis\text{-}memb, (\in)) \in Id \rightarrow \langle Id \rangle vis\text{-}rel \rightarrow bool\text{-}rel$

assumes [*autoref-rules*]:

$(Gi, G) \in \langle Rm, Id \rangle g\text{-}impl\text{-}rel\text{-}ext$

$(Pi, P) \in Id \rightarrow bool\text{-}rel$

$(Ri, R) \in \langle Id \rangle vis\text{-}rel$

shows $(nres\text{-}of\ ?c, find\text{-}path1\text{-}restr\ G\ P\ R)$

$\in \langle \langle \langle Id \rangle vis\text{-}rel, \langle Id \rangle list\text{-}rel \times_r Id \rangle sum\text{-}rel \rangle nres\text{-}rel$

<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*: $(op\text{-}vis\text{-}insert, insert) \in V \rightarrow \langle V \rangle vis\text{-}rel \rightarrow \langle V \rangle vis\text{-}rel$

$(op\text{-}vis\text{-}memb, (\in)) \in V \rightarrow \langle V \rangle vis\text{-}rel \rightarrow bool\text{-}rel$

assumes *Vid*[*simp*]: $V = Id$

shows $(\lambda G P R. nres\text{-}of (find\text{-}path1\text{-}restr\text{-}code\ op\text{-}vis\text{-}insert\ op\text{-}vis\text{-}memb\ G\ P\ R), find\text{-}path1\text{-}restr\text{-}spec)$

$\in \langle Rm, V \rangle g\text{-}impl\text{-}rel\text{-}ext \rightarrow (V \rightarrow bool\text{-}rel) \rightarrow \langle V \rangle vis\text{-}rel \rightarrow$

$\langle \langle \langle V \rangle vis\text{-}rel, \langle V \rangle list\text{-}rel \times_r V \rangle sum\text{-}rel \rangle nres\text{-}rel$

<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*]:

$(Gi, G) \in \langle Rm, V \rangle g\text{-}impl\text{-}rel\text{-}ext$

$(Pi, P) \in V \rightarrow bool\text{-}rel$

notes [*autoref-tyrel*] = *TYRELI*[**where** $R = \langle \langle \text{Id} :: ('a \times 'a :: \text{hashable}) \text{set} \rangle \rangle \text{dflt-ahs-rel}$]
shows (*nres-of* ?*c*, *find-path1* *G P*)
 $\in \langle \langle \langle V \rangle \text{list-rel} \times_r V \rangle \text{option-rel} \rangle \text{nres-rel}$
<proof>
concrete-definition *find-path1-code* **uses** *find-path1-code*

export-code *find-path1-code* **checking** *SML*

lemma *find-path1-code-autoref-aux*:
assumes *Vid*: $V = (\text{Id} :: 'a :: \text{hashable rel})$
shows $(\lambda G P. \text{nres-of } (\text{find-path1-code } G P), \text{find-path1-spec})$
 $\in \langle Rm, V \rangle \text{g-impl-rel-ext} \rightarrow (V \rightarrow \text{bool-rel}) \rightarrow \langle \langle \langle V \rangle \text{list-rel} \times_r V \rangle \text{option-rel} \rangle \text{nres-rel}$
<proof>

lemmas *find-path1-autoref*[*autoref-rules*] = *find-path1-code-autoref-aux*[*OF PRE-FER-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:  $\bigwedge x. x \in S \implies \exists xi. (xi, x) \in R$ 
  shows  $\exists Si. (Si, S) \in \langle R \rangle Rs$ 
  <proof>

```

```

lemma obtain-finite-ahs: finite S  $\implies \exists x. (x, S) \in \langle Id \rangle dflt-ahs-rel$ 
  <proof>

```

2.3.2 Framework Instantiation

```

definition unit-parametrization  $\equiv dflt-parametrization (\lambda-. ()) (RETURN ())$ 

```

```

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-DFS[Pure.intro?, intro?]:
  assumes fb-graph G
  shows unit-DFS G
  <proof>

```

```

definition find-reachable G  $\equiv do \{$ 
  ASSERT (fb-graph G);
  s  $\leftarrow unit-dfs.it-dfs TYPE('a) G$ ;
  RETURN (dom (discovered s))
   $\}$ 

```

definition *find-reachableT* $G \equiv do \{$
 ASSERT (*fb-graph* G);
 $s \leftarrow unit\text{-dfs.it}\text{-dfsT } TYPE('a) \ G$;
 RETURN (*dom* (*discovered* s))
 $\}$

2.3.3 Correctness

context *unit-DFS* **begin**

lemma *find-reachable-correct*: *find-reachable* $G \leq SPEC (\lambda r. r = \text{reachable})$
 $\langle proof \rangle$

lemma *find-reachableT-correct*:

finite reachable $\implies find\text{-reachableT} \ G \leq SPEC (\lambda r. r = \text{reachable})$
 $\langle proof \rangle$

end

context *unit-DFS* **begin**

sublocale *simple-impl* $G \ unit\text{-parametrization} \ unit\text{-parametrization} \ unit\text{-rel}$
 $\langle proof \rangle$

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

interpretation *unit-simple-impl*:

simple-impl-defs $G \ unit\text{-parametrization} \ unit\text{-parametrization}$
 for $G \ \langle proof \rangle$

term *unit-simple-impl.tailrec-impl* **term** *unit-simple-impl.rec-impl*

definition [*DFS-code-unfold*]: *find-reachable-impl* $G \equiv do \{$
 ASSERT (*fb-graph* G);
 $s \leftarrow unit\text{-simple-impl.tailrec-impl} \ TYPE('a) \ G$;
 RETURN (*simple-state.visited* s)
 $\}$

definition [*DFS-code-unfold*]: *find-reachable-implT* $G \equiv do \{$
 ASSERT (*fb-graph* G);
 $s \leftarrow unit\text{-simple-impl.tailrec-implT} \ TYPE('a) \ G$;
 RETURN (*simple-state.visited* s)
 $\}$

definition [*DFS-code-unfold*]: *find-reachable-rec-impl* $G \equiv do \{$
 ASSERT (*fb-graph* G);
 $s \leftarrow unit\text{-simple-impl.rec-impl} \ TYPE('a) \ G$;
 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*]:
 $(Gi, G) \in \langle Rm, V \rangle g\text{-impl-rel-ext}$
notes [*unfolded V-def, autoref-tyrel*] =
 TYRELI[**where** $R = \langle V \rangle dflt\text{-ahs-rel}$]
 TYRELI[**where** $R = \langle V \times_r \langle V \rangle list\text{-set-rel} \rangle ras\text{-rel}$]
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: $(Gi, G) \in \langle Rm, Id \rangle g\text{-impl-rel-ext}$
assumes 4: *find-reachable-code* $Gi = dRETURN$ r
shows $(r, (g\text{-E } G)^* \text{ `` } g\text{-V0 } G) \in \langle Id \rangle dflt\text{-ahs-rel}$
{*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*]: $(eq, (=)) \in V \rightarrow V \rightarrow bool\text{-rel}$
assumes [*autoref-ga-rules*]: *is-valid-def-hm-size* *TYPE* ($'vi$) *sz*
assumes [*autoref-rules*]:
 $(Gi, G) \in \langle Rm, V \rangle g\text{-impl-rel-ext}$
notes [*autoref-tyrel*] =
 TYRELI[**where** $R = \langle V \rangle ahs\text{-rel } bhc$]
 TYRELI[**where** $R = \langle V \times_r \langle V \rangle list\text{-set-rel} \rangle ras\text{-rel}$]
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* $((g-E\ G)^* \text{ `` } g-V0\ G)$
assumes BHC : *is-bounded-hashcode* $V\ eq\ bhc$
assumes EQ : $(eq, (=)) \in V \rightarrow V \rightarrow bool\text{-rel}$
assumes VDS : *is-valid-def-hm-size* $TYPE\ ('vi)\ sz$
assumes \mathcal{Z} : $(Gi, G) \in \langle Rm, V \rangle g\text{-impl-rel-ext}$
shows $(find\text{-reachable-codeT}\ eq\ bhc\ sz\ Gi, (g-E\ G)^* \text{ `` } g-V0\ G) \in \langle V \rangle ahs\text{-rel}\ bhc$
 $\langle proof \rangle$

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

lemma *all-unit-refine[simp]*:

$((), x) \in all\text{-unit-rel} \langle proof \rangle$

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

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

lemmas $[autoref\text{-rel-intf}] = REL\text{-INTFI}$ $[of\ unit\text{-list-rel}\ i\text{-list}]$

lemma $[autoref\text{-rules}]$:

$((), []) \in \langle R \rangle unit\text{-list-rel}$
 $(\lambda\cdot. (), tl) \in \langle R \rangle unit\text{-list-rel} \rightarrow \langle R \rangle unit\text{-list-rel}$
 $(\lambda\cdot. (), (\#)) \in R \rightarrow \langle R \rangle unit\text{-list-rel} \rightarrow \langle R \rangle unit\text{-list-rel}$
 $\langle proof \rangle$

schematic-goal *find-reachable-rec-impl*:

defines $V \equiv Id :: ('v \times 'v::hashable)$ *set*

assumes $[unfolded\ V\text{-def}, autoref\text{-rules}]$:

$(Gi, G) \in \langle Rm, V \rangle g\text{-impl-rel-ext}$

notes $[unfolded\ V\text{-def}, autoref\text{-tyrel}] =$

$TYRELI$ $[where\ R = \langle V \rangle dflt\text{-ahs-rel}]$

shows $nres\text{-of}\ (?c::?'c\ dres) \leq \Downarrow ?R\ (find\text{-reachable-rec-impl}\ G)$

$\langle proof \rangle$

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: $(Gi, G) \in \langle Rm, Id \rangle g\text{-impl-rel-ext}$
assumes 4: *find-reachable-rec-code* $Gi = dRETURN\ r$
shows $(r, (g-E\ G)^* \text{ “ } g-V0\ G) \in \langle Id \rangle dflt\text{-ahs-rel}$
 $\langle proof \rangle$

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

context begin interpretation *autoref-syn* $\langle proof \rangle$

lemma *autoref-op-reachable*[*autoref-rules*]:
fixes $V :: ('vi \times 'v)\ set$
assumes $G: SIDE\text{-}PRECOND\ (graph\ G)$
assumes $FR: SIDE\text{-}PRECOND\ (finite\ ((g-E\ G)^* \text{ “ } g-V0\ G))$
assumes $BHC: SIDE\text{-}GEN\text{-}ALGO\ (is\text{-}bounded\text{-}hashcode\ V\ eq\ bhc)$
assumes $EQ: GEN\text{-}OP\ eq\ (=)\ (V \rightarrow V \rightarrow bool\text{-}rel)$
assumes $VDS: SIDE\text{-}GEN\text{-}ALGO\ (is\text{-}valid\text{-}def\text{-}hm\text{-}size\ TYPE\ ('vi)\ sz)$
assumes 2: $(Gi, G) \in \langle Rm, V \rangle g\text{-impl-rel-ext}$
shows $(find\text{-}reachable\text{-}codeT\ eq\ bhc\ sz\ Gi,$
 $(OP\ op\text{-}reachable\ ::\ \langle Rm, V \rangle g\text{-impl-rel-ext} \rightarrow \langle V \rangle ahs\text{-}rel\ bhc)\$G) \in \langle V \rangle ahs\text{-}rel$
 bhc
 $\langle proof \rangle$

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
 $\dots/DFS\text{-}Framework$
 $CAVA\text{-}Automata.Digraph\text{-}Impl$
 $Reachable\text{-}Nodes$
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* :: $('v, 'more)\ graph\text{-}rec\text{-}scheme \Rightarrow 'v\ rel \Rightarrow bool$ **where**
 $is\text{-}fas\ G\ EC \equiv \neg(\exists\ u \in (g-E\ G)^* \text{ “ } g-V0\ G. (u, u) \in (g-E\ G - EC)^+)$

lemma *is-fas-alt*:

is-fas G $EC = acyclic ((g-E G \cap ((g-E G)^* \text{“} g-V0 G \times UNIV) - EC))$
 ⟨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' \implies fas s = fas s'*
 ⟨proof⟩

lemma [*simp*]: *s*(*state.more* := (*fas = foo* \Downarrow)) = *s* (*fas* := *foo* \Downarrow)
 ⟨proof⟩

definition *fas-params* :: (*'v*, (*'v*, *unit*) *fas-state-ext*) parameterization
where *fas-params* \equiv *dflt-parameterization state.more*
 (*RETURN* (*fas = {}* \Downarrow)) (*on-back-edge* := $\lambda u v s. RETURN$ (*fas = insert* (*u,v*) (*fas s*) \Downarrow))

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* $G \equiv do$ {
ASSERT (*graph* G);
ASSERT (*finite* ($(g-E G)^* \text{“} g-V0 G$));
s \leftarrow *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* ($(g-E G)^* \text{“} g-V0 G$)
begin

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

end

lemma *fasI*:
assumes *graph* G
assumes *finite* ($(g-E G)^* \text{“} g-V0 G$)
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* ($\lambda s. \text{fas-state.fas } s = \text{back-edges } s$)
<proof>

lemmas *fas-eq-back* = *i-fas-eq-back*[*THEN make-invar-thm*]

lemma *find-fas-correct-aux*:
 assumes *NC*: $\neg \text{cond } s$
 shows *is-fas* *G* (*fas-state.fas* *s*)
<proof>

end

lemma *find-fas-correct*:
 assumes *graph* *G*
 assumes *finite* ($(g-E \ G)^* \text{ `` } g-V0 \ G$)
 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* \equiv {
 ($\langle \text{fas-state-impl.fas} = f \ \rangle$, $\langle \text{fas-state.fas} = f \rangle$) | *f*. *True* }

abbreviation *fas-rel* \equiv $\langle \text{fas-erel} \rangle \text{simple-state-rel}$

definition *fas-params-impl*
 :: (*'v*, *'v fas-state-impl*, (*'v*, *unit*) *fas-state-impl-ext*) *gen-parameterization*

where *fas-params-impl*
 \equiv *dflt-parameterization simple-state.more* (*RETURN* ($\langle \text{fas} = \{\} \ \rangle$)) (\langle
 on-back-edge := $\lambda u \ v \ s. \text{RETURN } (\langle \text{fas} = \text{insert } (u,v) (\text{fas } s) \ \rangle)$ \rangle)

lemmas *fas-params-impl-simp*[*simp*, *DFS-code-unfold*] =
 gen-parameterization.simps[*mk-record-simp*, *OF fas-params-impl-def*[*simplified*]]

lemma *fas-impl*: (*si*, *s*) \in *fas-rel*
 \implies *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* $G \equiv \text{do } \{$
 $\text{ASSERT } (\text{graph } G);$
 $\text{ASSERT } (\text{finite } ((g-E \ G)^* \ \text{“ } g-V0 \ G));$
 $s \leftarrow \text{fas-impl.tailrec-implT } \text{TYPE}('a) \ G;$
 $\text{RETURN } (\text{fas } s)$
 $\}$

context *fas* **begin**

sublocale *simple-impl* G *fas-params* *fas-params-impl* *fas-erel*
 $\langle \text{proof} \rangle$

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

lemma *find-fas-impl-refine*: *find-fas-impl* $G \leq \Downarrow \text{Id } (\text{find-fas } G)$
 $\langle \text{proof} \rangle$

2.4.4 Synthesis of Executable Code

record $('si, 'nsi, 'fsi) \text{fas-state-impl}' = ('si, 'nsi) \text{simple-state-impl} +$
 $\text{fas-impl} :: 'fsi$

definition [*to-relAPP*]: *fas-state-erel* *frel* *erel* $\equiv \{$
 $(\downarrow \text{fas-impl} = fi, \dots = mi), (\downarrow \text{fas} = f, \dots = m)) \mid fi \ mi \ f \ m.$
 $(fi, f) \in \text{frel} \wedge (mi, m) \in \text{erel} \}$

consts

i-fas-state-ext :: *interface* \Rightarrow *interface* \Rightarrow *interface*

lemmas [*autoref-rel-intf*] = *REL-INTFI*[*of fas-state-erel i-fas-state-ext*]

term *fas-update*

term *fas-state-impl'.fas-impl-update*

lemma [*autoref-rules*]:

fixes *ns-rel* *vis-rel* *frel* *erel*

defines $R \equiv \langle \text{ns-rel}, \text{vis-rel}, \langle \text{frel}, \text{erel} \rangle \text{fas-state-erel} \rangle \text{ss-impl-rel}$

shows

$(\text{fas-state-impl}'\text{-ext}, \text{fas-state-impl-ext}) \in \text{frel} \rightarrow \text{erel} \rightarrow \langle \text{frel}, \text{erel} \rangle \text{fas-state-erel}$

$(\text{fas-impl}, \text{fas-state-impl.fas}) \in R \rightarrow \text{frel}$

$(\text{fas-state-impl}'\text{-fas-impl-update}, \text{fas-update}) \in (\text{frel} \rightarrow \text{frel}) \rightarrow R \rightarrow R$

$\langle \text{proof} \rangle$

schematic-goal *find-fas-impl*:

fixes $V :: ('vi \times 'v)$ *set*
assumes [*autoref-ga-rules*]: *is-bounded-hashcode* V *eq* *bhc*
assumes [*autoref-rules*]: $(eq, (=)) \in V \rightarrow V \rightarrow \text{bool-rel}$
assumes [*autoref-ga-rules*]: *is-valid-def-hm-size* *TYPE* $('vi)$ *sz*
assumes [*autoref-rules*]:
 $(Gi, G) \in \langle Rm, V \rangle g\text{-impl-rel-ext}$
notes [*autoref-tyrel*] =
 $TYRELI[\text{where } R = \langle V \rangle \text{ahs-rel } bhc]$
 $TYRELI[\text{where } R = \langle V \times_r V \rangle \text{ahs-rel } (prod\text{-bhc } bhc \text{ } bhc)]$
 $TYRELI[\text{where } R = \langle V \times_r \langle V \rangle \text{list-set-rel} \rangle \text{ras-rel}]$
shows *RETURN* $(?c :: ?'c) \leq \Downarrow ?R$ (*find-fas-impl* G)
 $\langle proof \rangle$

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 $(eq, (=)) \in V \rightarrow V \rightarrow \text{bool-rel}$
assumes *is-valid-def-hm-size* *TYPE* $('vi)$ *sz*
assumes $2: (Gi, G) \in \langle Rm, V \rangle g\text{-impl-rel-ext}$
shows *RETURN* $(find\text{-fas-code } eq \text{ } bhc \text{ } sz \text{ } Gi) \leq \Downarrow (\langle V \times_r V \rangle \text{ahs-rel } (prod\text{-bhc } bhc \text{ } bhc))$ (*find-fas* G)
 $\langle proof \rangle$

context begin interpretation *autoref-syn* $\langle proof \rangle$

Declare this algorithm to Autoref:

theorem *find-fas-code-autoref*[*autoref-rules*]:

fixes $V :: ('vi \times 'v)$ *set* **and** *bhc*
defines $RR \equiv \langle \langle V \times_r V \rangle \text{ahs-rel } (prod\text{-bhc } bhc \text{ } bhc) \rangle \text{nres-rel}$
assumes *BHC*: *SIDE-GEN-ALGO* (*is-bounded-hashcode* V *eq* *bhc*)
assumes *EQ*: *GEN-OP* *eq* $(=)$ $(V \rightarrow V \rightarrow \text{bool-rel})$
assumes *VDS*: *SIDE-GEN-ALGO* (*is-valid-def-hm-size* *TYPE* $('vi)$ *sz*)
assumes $2: (Gi, G) \in \langle Rm, V \rangle g\text{-impl-rel-ext}$
shows (*RETURN* $(find\text{-fas-code } eq \text{ } bhc \text{ } sz \text{ } Gi)$,
 $(OP \text{ } find\text{-fas}$
 $::: \langle Rm, V \rangle g\text{-impl-rel-ext} \rightarrow RR) \$ G) \in RR$
 $\langle proof \rangle$

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 ( $(g-E\ G)^* \text{ `` } g-V0\ G$ ));
  let  $nodes = (g-E\ G)^* \text{ `` } g-V0\ G$ ;
   $fas \leftarrow find-fas \ (\ \ g-V = g-V\ G, g-E = g-E\ G - FI, g-V0 = nodes \ )$ ;
  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)^* \text{ `` } g-V0\ G \)\ F1$
 $\langle proof \rangle$

lemma *graphI-init*:

assumes *graph* G
shows $graph\ (\ \ g-V = g-V\ G, g-E = g-E\ G - FI, g-V0 = (g-E\ G)^* \text{ `` } g-V0\ G \)$
 $\langle proof \rangle$

lemma *find-fas-init-correct*:

assumes [*simp*, *intro!*]: *graph* G
assumes [*simp*, *intro!*]: *finite* ($(g-E\ G)^* \text{ `` } g-V0\ G$)
shows $find-fas-init\ G\ FI \leq SPEC\ (\lambda fas. is-fas\ G\ fas \wedge FI \subseteq fas)$
 $\langle proof \rangle$

lemma *gen-cast-set*[*autoref-rules-raw*]:

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

lemma *gen-cast-fun-set-rel*[*autoref-rules-raw*]:

assumes *INS*: *GEN-OP* $mem\ (\in)\ (Rk \rightarrow \langle Rk \rangle Rs \rightarrow bool-rel)$
shows $(\lambda s\ x. mem\ x\ s, CAST) \in (\langle Rk \rangle Rs) \rightarrow (\langle Rk \rangle fun-set-rel)$
 $\langle proof \rangle$

lemma *find-fas-init-impl-aux-unfolds*:

$Let\ (E^* \text{ `` } V0) = Let\ (CAST\ (E^* \text{ `` } V0))$

$(\lambda S. RETURN (FI \cup S)) = (\lambda S. RETURN (FI \cup CAST S))$
 $\langle proof \rangle$

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

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* $\langle proof \rangle$

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

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  $s = \text{state.more } s' \implies \text{lasso } s = \text{lasso } s'$   
(proof)
```

```
lemma red-more-cong[cong]: state.more  $s = \text{state.more } s' \implies \text{red } s = \text{red } s'$   
(proof)
```

```
lemma [simp]:  $s \langle \text{state.more} := \langle \text{lasso} = \text{foo}, \text{red} = \text{bar} \rangle \rangle = s \langle \text{lasso} := \text{foo}, \text{red} := \text{bar} \rangle$   
(proof)
```

```
abbreviation dropWhileNot  $v \equiv \text{dropWhile } ((\neq) v)$ 
```

```
abbreviation takeWhileNot  $v \equiv \text{takeWhile } ((\neq) v)$ 
```

```
locale BlueDFS-defs = graph-defs  $G$ 
  for  $G :: ('v, 'more) \text{graph-rec-scheme} +$ 
  fixes accpt :: 'v  $\Rightarrow \text{bool}$ 
begin
```

```
definition blue  $s \equiv \text{dom } (\text{finished } s) - \text{red } s$ 
```

```
definition cyan  $s \equiv \text{set } (\text{stack } s)$ 
```

```
definition white  $s \equiv - \text{dom } (\text{discovered } s)$ 
```

abbreviation $red\text{-}dfs\ R\ ss\ x \equiv find\text{-}path1\text{-}restr\text{-}spec\ (G\ (\ g\text{-}V0 := \{x\} \))\ ss\ R$

definition $mk\text{-}blue\text{-}witness$

$:: 'v\ blue\text{-}dfs\text{-}state \Rightarrow 'v\ fpr\text{-}result \Rightarrow ('v, unit)\ blue\text{-}dfs\text{-}state\text{-}ext$

where

$mk\text{-}blue\text{-}witness\ s\ redS \equiv case\ redS\ of$

$\quad Inl\ R' \Rightarrow (\ lasso = None, red = (R' \setminus \# \setminus \# \setminus \#) \)$

$\quad | Inr\ (vs, v) \Rightarrow let\ rs = rev\ (stack\ s)\ in$

$\quad (\ lasso = Some\ (rs, vs@dropWhileNot\ v\ rs), red = red\ s)$

definition $run\text{-}red\text{-}dfs$

$:: 'v \Rightarrow 'v\ blue\text{-}dfs\text{-}state \Rightarrow ('v, unit)\ blue\text{-}dfs\text{-}state\text{-}ext\ nres$

where

$run\text{-}red\text{-}dfs\ u\ s \equiv case\ lasso\ s\ of\ None \Rightarrow do\ \{$

$\quad redS \leftarrow red\text{-}dfs\ (red\ s)\ (\lambda x. x = u \vee x \in cyan\ s)\ u;$

$\quad RETURN\ (mk\text{-}blue\text{-}witness\ s\ redS)$

$\quad \}$

$| - \Rightarrow NOOP\ s$

Schwoon-Esparza extension

definition $se\text{-}back\text{-}edge\ u\ v\ s \equiv case\ lasso\ s\ of$

$None \Rightarrow$

— 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

$if\ accept\ u\ then$

$\quad let\ rs = rev\ (tl\ (stack\ s));$

$\quad ur = rs;$

$\quad ul = u\#\dropWhileNot\ v\ rs$

$\quad in\ RETURN\ (\ lasso = Some\ (ur, ul), red = red\ s)$

$else\ if\ accept\ v\ then$

$\quad let\ rs = rev\ (stack\ s);$

$\quad vr = takeWhileNot\ v\ rs;$

$\quad vl = dropWhileNot\ v\ rs$

$\quad in\ RETURN\ (\ lasso = Some\ (vr, vl), red = red\ s)$

$else\ NOOP\ s$

$| - \Rightarrow NOOP\ s$

definition $blue\text{-}dfs\text{-}params :: 'v\ blue\text{-}dfs\text{-}param$

where $blue\text{-}dfs\text{-}params = (\$

$\quad on\text{-}init = RETURN\ (\ lasso = None, red = \{ \} \),$

$\quad on\text{-}new\text{-}root = \lambda v\ 0\ s. NOOP\ s,$

$\quad on\text{-}discover = \lambda u\ v\ s. NOOP\ s,$

$\quad on\text{-}finish = \lambda u\ s. if\ accept\ u\ then\ run\text{-}red\text{-}dfs\ u\ s\ else\ NOOP\ s,$

$\quad on\text{-}back\text{-}edge = se\text{-}back\text{-}edge,$

$\quad on\text{-}cross\text{-}edge = \lambda u\ v\ s. NOOP\ s,$

$\quad is\text{-}break = \lambda s. lasso\ s \neq None \)$

schematic-goal $blue\text{-}dfs\text{-}params\text{-}simps[simp]:$

```

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 accept + param-DFS G blue-dfs-params
for G :: ('v, 'more) graph-rec-scheme and accept :: 'v ⇒ bool

```

```

lemma BlueDFS:
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 accept 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

```

$fb\text{-graph } (G \ (| \ g\text{-}V0 := \{hd \ (stack \ s)\} \ |))$
and $fb\text{-graph } (G \ (| \ g\text{-}E := E \cap \text{UNIV} \times - \text{red } s, \ g\text{-}V0 := \{hd \ (stack \ s)\} \ |))$
and $restr\text{-invar } E \ (red \ s) \ (\lambda x. \ x \in \text{set} \ (stack \ s))$
 $\langle proof \rangle$

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

lemma $blue\text{-basic}\text{-invar}$: $is\text{-invar } blue\text{-basic}\text{-invar}$
 $\langle proof \rangle$

lemmas $(in \ BlueDFS\text{-invar}) \ s\text{-blue}\text{-basic}\text{-invar}$
 $= blue\text{-basic}\text{-invar} [THEN \ make\text{-invar}\text{-thm}]$

lemmas $(in \ BlueDFS\text{-invar}) \ red\text{-DFS}\text{-precond}$
 $= red\text{-DFS}\text{-precond}\text{-aux} [OF \ s\text{-blue}\text{-basic}\text{-invar}]$

sublocale $DFS \ G \ blue\text{-dfs}\text{-params}$
 $\langle proof \rangle$

end

context $BlueDFS\text{-invar}$
begin

context assumes $[simp]$: $lasso \ s = None$
begin

lemma $red\text{-closed}$:
 E “ $red \ s \subseteq red \ s$
 $\langle proof \rangle$

lemma $red\text{-stack}\text{-disjoint}$:
 $set \ (stack \ s) \cap red \ s = \{\}$
 $\langle proof \rangle$

lemma $red\text{-finished}$: $red \ s \subseteq dom \ (finished \ s)$
 $\langle proof \rangle$

lemma $all\text{-nodes}\text{-colored}$: $white \ s \cup blue \ s \cup cyan \ s \cup red \ s = UNIV$
 $\langle proof \rangle$

lemma *colors-disjoint*:

$$white\ s \cap (blue\ s \cup cyan\ s \cup red\ s) = \{\}$$

$$blue\ s \cap (white\ s \cup cyan\ s \cup red\ s) = \{\}$$

$$cyan\ s \cap (white\ s \cup blue\ s \cup red\ s) = \{\}$$

$$red\ s \cap (white\ s \cup blue\ s \cup cyan\ s) = \{\}$$

<proof>

end

lemma (**in** *BlueDFS*) *i-no-accept-cycle-in-finish*:

$$is_invar\ (\lambda s. \text{lasso } s = None \longrightarrow (\forall x. \text{accept } x \wedge x \in \text{dom } (\text{finished } s) \longrightarrow (x,x) \notin E^+))$$

<proof>

lemma *no-accept-cycle-in-finish*:

$$\llbracket \text{lasso } s = None; \text{accept } v; v \in \text{dom } (\text{finished } s) \rrbracket \Longrightarrow (v,v) \notin E^+$$

<proof>

end

context *BlueDFS*

begin

definition *lasso-inv* **where**

$$\text{lasso-inv } s \equiv \forall pr\ pl. \text{lasso } s = \text{Some } (pr,pl) \longrightarrow$$

$$pl \neq []$$

$$\wedge (\exists v0 \in V0. \text{path } E\ v0\ pr\ (\text{hd } pl))$$

$$\wedge \text{accept } (\text{hd } pl)$$

$$\wedge \text{path } E\ (\text{hd } pl)\ pl\ (\text{hd } pl)$$

lemma (**in** *BlueDFS-invar*) *se-back-edge-lasso-inv*:

assumes *b-inv*: *lasso-inv* *s*

and *ne*: *stack* *s* $\neq []$

and *R*: *lasso* *s* = *None*

and *p*: $(\text{hd } (\text{stack } s), v) \in \text{pending } s$

and *v*: $v \in \text{dom } (\text{discovered } s) \wedge v \notin \text{dom } (\text{finished } s)$

and *s'*: $s' = \text{back-edge } (\text{hd } (\text{stack } s))\ v\ (s \setminus \{\text{pending} := \text{pending } s - \{(u,v)\}\})$

shows *se-back-edge* $(\text{hd } (\text{stack } s))\ v\ s'$

$$\leq \text{SPEC } (\lambda e. \text{DFS-invar } G\ \text{blue-dfs-params } (s' \setminus \{\text{state.more} := e\}) \longrightarrow \text{lasso-inv } (s' \setminus \{\text{state.more} := e\}))$$

<proof>

lemma *lasso-inv*:

is-invar *lasso-inv*

<proof>

end

context *BlueDFS-invar*

begin

lemmas $s\text{-lasso-inv} = \text{lasso-inv}[\text{THEN make-invar-thm}]$

lemma

assumes $\text{lasso } s = \text{Some } (pr, pl)$
shows $\text{loop-nonempty}: pl \neq []$
and $\text{accept-loop}: \text{accept } (hd\ pl)$
and $\text{loop-is-path}: \text{path } E\ (hd\ pl)\ pl\ (hd\ pl)$
and $\text{loop-reachable}: \exists v0 \in V0. \text{path } E\ v0\ pr\ (hd\ pl)$
 $\langle \text{proof} \rangle$

lemma blue-dfs-correct :

assumes $NC: \neg \text{cond } s$
shows $\text{case } \text{lasso } s\ \text{of}$
 $\text{None} \Rightarrow \neg(\exists v0 \in V0. \exists v. (v0, v) \in E^* \wedge \text{accept } v \wedge (v, v) \in E^+)$
 $| \text{Some } (pr, pl) \Rightarrow (\exists v0 \in V0. \exists v.$
 $\text{path } E\ v0\ pr\ v \wedge \text{accept } v \wedge pl \neq [] \wedge \text{path } E\ v\ pl\ v)$
 $\langle \text{proof} \rangle$

end

2.5.4 Interface

interpretation $\text{BlueDFS-defs for } G\ \text{accept } \langle \text{proof} \rangle$

definition $\text{nested-dfs-spec } G\ \text{accept} \equiv \lambda r. \text{case } r\ \text{of}$

$\text{None} \Rightarrow \neg(\exists v0 \in g\text{-}V0\ G. \exists v. (v0, v) \in (g\text{-}E\ G)^* \wedge \text{accept } v \wedge (v, v) \in (g\text{-}E\ G)^+)$
 $| \text{Some } (pr, pl) \Rightarrow (\exists v0 \in g\text{-}V0\ G. \exists v.$
 $\text{path } (g\text{-}E\ G)\ v0\ pr\ v \wedge \text{accept } v \wedge pl \neq [] \wedge \text{path } (g\text{-}E\ G)\ v\ pl\ v)$

definition $\text{nested-dfs } G\ \text{accept} \equiv \text{do } \{$

$\text{ASSERT } (fb\text{-graph } G);$
 $s \leftarrow \text{it-dfs } \text{TYPE}('a)\ G\ \text{accept};$
 $\text{RETURN } (\text{lasso } s)$
 $\}$

theorem $\text{nested-dfs-correct}$:

assumes $fb\text{-graph } G$
shows $\text{nested-dfs } G\ \text{accept} \leq \text{SPEC } (\text{nested-dfs-spec } G\ \text{accept})$
 $\langle \text{proof} \rangle$

2.5.5 Implementation

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

definition $\text{bdfs-erel} \equiv \{(\langle \text{lasso-impl}=li, \text{red-impl}=ri \rangle, \langle \text{lasso}=l, \text{red}=r \rangle)$
 $| li\ ri\ l\ r. li=l \wedge ri=r\}$

abbreviation $\text{bdfs-rel} \equiv \langle \text{bdfs-erel} \rangle \text{simple-state-rel}$

definition *mk-blue-witness-impl*

$:: 'v \text{ bdfs-state-impl} \Rightarrow 'v \text{ fpr-result} \Rightarrow ('v, \text{unit}) \text{ bdfs-state-impl-ext}$

where

$\text{mk-blue-witness-impl } s \text{ redS} \equiv$

case redS of

$\text{Inl } R' \Rightarrow (\text{lasso-impl} = \text{None}, \text{red-impl} = (R' \text{ // red-impl}))$

$| \text{Inr } (vs, v) \Rightarrow \text{let}$

$rs = \text{rev } (\text{map fst } (\text{CAST } (\text{ss-stack } s)))$

$\text{in } (\text{lasso-impl} = \text{Some } (rs, \text{vs@dropWhileNot } v \text{ rs}),$

$\text{red-impl} = \text{red-impl } s)$

lemma *mk-blue-witness-impl[refine]*:

$\llbracket (si, s) \in \text{bdfs-rel}; (ri, r) \in (\text{Id}, \langle \text{Id} \rangle \text{list-rel} \times_r \text{Id}) \text{sum-rel} \rrbracket$

$\implies (\text{mk-blue-witness-impl } si \text{ ri}, \text{mk-blue-witness } s \text{ r}) \in \text{bdfs-erel}$

$\langle \text{proof} \rangle$

definition *cyan-impl* $s \equiv \text{on-stack } s$

lemma *cyan-impl[refine]*: $\llbracket (si, s) \in \text{bdfs-rel} \rrbracket \implies (\text{cyan-impl } si, \text{cyan } s) \in \text{Id}$

$\langle \text{proof} \rangle$

definition *run-red-dfs-impl*

$:: ('v, 'more) \text{ graph-rec-scheme} \Rightarrow 'v \Rightarrow 'v \text{ bdfs-state-impl} \Rightarrow ('v, \text{unit}) \text{ bdfs-state-impl-ext}$
nres

where

$\text{run-red-dfs-impl } G \text{ u } s \equiv \text{case lasso-impl } s \text{ of None} \Rightarrow \text{do } \{$

$\text{redS} \leftarrow \text{red-dfs TYPE('more) } G \text{ (red-impl } s) (\lambda x. x = u \vee x \in \text{cyan-impl}$

$s) \text{ u};$

$\text{RETURN } (\text{mk-blue-witness-impl } s \text{ redS})$

$\}$

$| - \Rightarrow \text{RETURN } (\text{simple-state.more } s)$

lemma *run-red-dfs-impl[refine]*: $\llbracket (Gi, G) \in \text{Id}; (ui, u) \in \text{Id}; (si, s) \in \text{bdfs-rel} \rrbracket$

$\implies \text{run-red-dfs-impl } Gi \text{ ui } si \leq \llbracket \text{bdfs-erel } (\text{run-red-dfs TYPE('a) } G \text{ u } s)$

$\langle \text{proof} \rangle$

definition *se-back-edge-impl* $\text{accept } u \text{ v } s \equiv \text{case lasso-impl } s \text{ of}$

$\text{None} \Rightarrow$

$\text{if accept } u \text{ then}$

$\text{let } rs = \text{rev } (\text{map fst } (\text{tl } (\text{CAST } (\text{ss-stack } s))));$

$ur = rs;$

$ul = u \# \text{dropWhileNot } v \text{ rs}$

$\text{in RETURN } (\text{lasso-impl} = \text{Some } (ur, ul), \text{red-impl} = \text{red-impl } s)$

$\text{else if accept } v \text{ then}$

$\text{let } rs = \text{rev } (\text{map fst } (\text{CAST } (\text{ss-stack } s)));$

$vr = \text{takeWhileNot } v \text{ rs};$

$vl = \text{dropWhileNot } v \text{ rs}$

$\text{in RETURN } (\text{lasso-impl} = \text{Some } (vr, vl), \text{red-impl} = \text{red-impl } s)$

else RETURN (simple-state.more s)
 | - ⇒ RETURN (simple-state.more s)

lemma *se-back-edge-impl*[refine]: $\llbracket (accpti, accpt) \in Id; (ui, u) \in Id; (vi, v) \in Id; (si, s) \in bdfs\text{-}rel$
 \rrbracket
 $\implies se\text{-}back\text{-}edge\text{-}impl\ accpt\ ui\ vi\ si \leq \Downarrow bdfs\text{-}erel (se\text{-}back\text{-}edge\ accpt\ u\ v\ s)$
 ⟨proof⟩

lemma *NOOP-impl*: $(si, s) \in bdfs\text{-}rel$
 $\implies RETURN (simple\text{-}state.more\ si) \leq \Downarrow bdfs\text{-}erel (NOOP\ s)$
 ⟨proof⟩

definition *bdfs-params-impl*
 $:: ('v, 'more)\ graph\text{-}rec\text{-}scheme \Rightarrow ('v \Rightarrow bool) \Rightarrow ('v, 'v\ bdfs\text{-}state\text{-}impl, ('v, unit)\ bdfs\text{-}state\text{-}impl\text{-}ext)$
gen-parameterization
where *bdfs-params-impl* *G* *accpt* $\equiv \langle$
on-init = RETURN (\langle lasso-impl = None, red-impl = $\{\}$ \rangle),
on-new-root = $\lambda v\ 0\ s.\ RETURN (simple\text{-}state.more\ s)$,
on-discover = $\lambda u\ v\ s.\ RETURN (simple\text{-}state.more\ s)$,
on-finish = $\lambda u\ s.$
 if *accpt* *u* then *run-red-dfs-impl* *G* *u* *s* else RETURN (simple-state.more s),
on-back-edge = *se-back-edge-impl* *accpt*,
on-cross-edge = $\lambda u\ v\ s.\ RETURN (simple\text{-}state.more\ s)$,
is-break = $\lambda s.\ lasso\text{-}impl\ s \neq None\ \rangle$

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* $\equiv do\ \{$
 ASSERT (*fb-graph* *G*);
s $\leftarrow impl.tailrec\text{-}impl\ TYPE('a)\ G\ accpt$;
 RETURN (*lasso-impl* *s*)
 $\}$

lemma *nested-dfs-impl*[*refine*]:
assumes $(Gi, G) \in Id$
assumes $(accpti, accpt) \in Id$
shows $nested-dfs-impl\ Gi\ accpti \leq \Downarrow (\langle Id \rangle list-rel \times_r \langle Id \rangle list-rel) option-rel$
 $(nested-dfs\ G\ accpt)$
 $\langle proof \rangle$

2.5.6 Synthesis of Executable Code

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

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

consts
 $i-bdfs-state-ext :: interface \Rightarrow interface$

lemmas [*autoref-rel-intf*] = *REL-INTFI*[*of bdfs-state-erel' i-bdfs-state-ext*]

lemma [*autoref-rules*]:
fixes $ns-rel\ vis-rel\ Vi$
defines $R \equiv \langle ns-rel, vis-rel, \langle Vi \rangle bdfs-state-erel' \rangle ss-impl-rel$
shows
 $(bdfs-state-impl'-ext, bdfs-state-impl-ext)$
 $\in \langle \langle \langle Vi \rangle list-rel \times_r \langle Vi \rangle list-rel \rangle option-rel \rightarrow \langle Vi \rangle dflt-ahs-rel \rightarrow unit-rel \rightarrow$
 $\langle Vi \rangle bdfs-state-erel'$
 $(lasso-impl', lasso-impl) \in R \rightarrow \langle \langle Vi \rangle list-rel \times_r \langle Vi \rangle list-rel \rangle option-rel$
 $(red-impl', red-impl) \in R \rightarrow \langle Vi \rangle dflt-ahs-rel$
 $\langle proof \rangle$

schematic-goal *nested-dfs-code*:
assumes $Vid: V = (Id :: ('v::hashable \times 'v) set)$
assumes [*unfolded Vid, autoref-rules*]:
 $(Gi, G) \in \langle Rm, V \rangle g-impl-rel-ext$
 $(accpti, accpt) \in (V \rightarrow bool-rel)$
notes [*unfolded Vid, autoref-tyrel*] =
 $TYRELI[\mathbf{where}\ R = \langle V \rangle dflt-ahs-rel]$
 $TYRELI[\mathbf{where}\ R = \langle V \rangle ras-rel]$
shows $(nres-of\ ?c, nested-dfs-impl\ G\ accpt)$
 $\in \langle \langle \langle V \rangle list-rel \times_r \langle V \rangle list-rel \rangle option-rel \rangle nres-rel$
 $\langle proof \rangle$

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

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-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-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-set s v)
```

```
end
```

```
end
```

```
context DFS-invar begin
```

lemma *lowlink-setI*:

assumes *lowlink-path s v p w*

and $w \in \text{dom } (\text{discovered } s)$

and $(v,w) \in E^* \ (w,v) \in E^*$

shows $w \in \text{lowlink-set } s \ v$

<proof>

lemma *lowlink-set-discovered*:

$\text{lowlink-set } s \ v \subseteq \text{dom } (\text{discovered } s)$

<proof>

lemma *lowlink-set-finite[simp, intro!]*:

$\text{finite } (\text{lowlink-set } s \ v)$

<proof>

lemma *lowlink-set-not-empty*:

assumes $v \in \text{dom } (\text{discovered } s)$

shows $\text{lowlink-set } s \ v \neq \{\}$

<proof>

lemma *lowlink-path-single*:

assumes $(v,w) \in \text{cross-edges } s \cup \text{back-edges } s$

shows $\text{lowlink-path } s \ v \ [v] \ w$

<proof>

lemma *lowlink-path-Cons*:

assumes $\text{lowlink-path } s \ v \ (x\#xs) \ w$

and $xs \neq []$

shows $\exists u. \text{lowlink-path } s \ u \ xs \ w$

<proof>

lemma *lowlink-path-in-tree*:

assumes $p: \text{lowlink-path } s \ v \ p \ w$

and $j: j < \text{length } p$

and $k: k < j$

shows $(p!k, p!j) \in (\text{tree-edges } s)^+$

<proof>

lemma *lowlink-path-finished*:

assumes $p: \text{lowlink-path } s \ v \ p \ w$

and $j: j < \text{length } p \ j > 0$

shows $p!j \in \text{dom } (\text{finished } s)$

<proof>

lemma *lowlink-path-tree-prepend*:

assumes $p: \text{lowlink-path } s \ v \ p \ w$

and $\text{tree-edges}: (u,v) \in (\text{tree-edges } s)^+$

and $\text{fin}: u \in \text{dom } (\text{finished } s) \vee (\text{stack } s \neq [] \wedge u = \text{hd } (\text{stack } s))$

shows $\exists p. \text{lowlink-path } s \ u \ p \ w$
 $\langle \text{proof} \rangle$

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

lemma *no-path-imp-no-lowlink-path*:
assumes $\text{edges } s \text{ “ } \{v\} = \{ \}$
shows $\neg \text{lowlink-path } s \ v \ p \ w$
 $\langle \text{proof} \rangle$

context begin interpretation *timing-syntax* $\langle \text{proof} \rangle$

lemma *LowLink-le-disc*:
assumes $v \in \text{dom } (\text{discovered } s)$
shows $\text{LowLink } s \ v \leq \delta \ s \ v$
 $\langle \text{proof} \rangle$

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

lemma *LowLink-lessI*:
assumes $y \in \text{lowlink-set } s \ v$
and $\delta \ s \ y < \delta \ s \ v$
shows $\text{LowLink } s \ v < \delta \ s \ v$
 $\langle \text{proof} \rangle$

lemma *LowLink-eqI*:
assumes *DFS-invar* $G \ \text{param } s'$
assumes *sub-m*: $\text{discovered } s \subseteq_m \text{discovered } s'$
assumes *sub*: $\text{lowlink-set } s \ w \subseteq \text{lowlink-set } s' \ w$
and *rev-sub*: $\text{lowlink-set } s' \ w \subseteq \text{lowlink-set } s \ w \cup X$
and *w-disc*: $w \in \text{dom } (\text{discovered } s)$
and $X: \bigwedge x. \llbracket x \in X; x \in \text{lowlink-set } s' \ w \rrbracket \implies \delta \ s' \ x \geq \text{LowLink } s \ w$
shows $\text{LowLink } s \ w = \text{LowLink } s' \ w$
 $\langle \text{proof} \rangle$

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

```

  ⟨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 ≡ dropWhile ((≠) u) tjs
  defines tw ≡ takeWhile ((≠) u) tjs
  assumes u ∈ set tjs
  shows set tjs = set (tl dw) ∪ insert u (set tw)
⟨proof⟩

```

2.7.2 Instantiation of the DFS-Framework

```

record 'v tarjan-state = 'v state +
  sccs :: 'v set set
  lowlink :: 'v → nat
  tj-stack :: 'v list

```

```

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

```

```

abbreviation the-lowlink s v ≡ the (lowlink s v)

```

```

context timing-syntax
begin
  notation the-lowlink ⟨ιζ⟩
end

```

```

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

```

```

  definition tarjan-disc :: 'v ⇒ 'v tarjan-state ⇒ ('v,unit) tarjan-state-ext nres
where

```

```

  tarjan-disc v s = RETURN (| sccs = sccs s,

```

$$\begin{aligned} \text{lowlink} &= (\text{lowlink } s)(v \mapsto \delta s v), \\ \text{tj-stack} &= v \# \text{tj-stack } s \end{aligned}$$

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

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

lemmas $\text{tj-stack-pop-set-leof-rule} = \text{weaken-SPEC}[\text{OF } \text{tj-stack-pop-set}, \text{THEN } \text{leof-lift}]$

definition $\text{tarjan-fin} :: 'v \Rightarrow 'v \text{ tarjan-state} \Rightarrow ('v, \text{unit}) \text{ tarjan-state-ext}$ **nres** **where**

```

tarjan-fin v s = do {
  let ll = (if stack s = [] then lowlink s
            else let u = hd (stack s) in
                (lowlink s)(u \mapsto min (\zeta s u) (\zeta s v)));
  let s' = s(| lowlink := ll |);

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

```

definition $\text{tarjan-back} :: 'v \Rightarrow 'v \Rightarrow 'v \text{ tarjan-state} \Rightarrow ('v, \text{unit}) \text{ tarjan-state-ext}$ **nres** **where**

```

tarjan-back u v s = (
  if \delta s v < \delta s u \wedge v \in set (tj-stack s) then
    let ul' = min (\zeta s u) (\delta s v)
    in RETURN (state.more (s(| lowlink := (lowlink s)(u \mapsto ul') |)))
  else NOOP s)

```

end

definition $\text{tarjan-params} :: 'v \text{ tarjan-param}$ **where**

```

tarjan-params = (|
  on-init = RETURN (| sccs = {}, lowlink = Map.empty, tj-stack = [] |),
  on-new-root = tarjan-disc,
  on-discover = \u. tarjan-disc,
  on-finish = tarjan-fin,

```

$on-back-edge = tarjan-back,$
 $on-cross-edge = tarjan-back,$
 $is-break = \lambda s. False$

schematic-goal $tarjan-params-simps[simp]:$

$on-init\ tarjan-params = ?OI$
 $on-new-root\ tarjan-params = ?ONR$
 $on-discover\ tarjan-params = ?OD$
 $on-finish\ tarjan-params = ?OF$
 $on-back-edge\ tarjan-params = ?OBE$
 $on-cross-edge\ tarjan-params = ?OCE$
 $is-break\ tarjan-params = ?IB$
 $\langle proof \rangle$

sublocale $param-DFS-defs\ G\ tarjan-params\ \langle proof \rangle$
end

locale $Tarjan = Tarjan-def\ G +$
 $param-DFS\ G\ tarjan-params$
for $G :: ('v, 'more)\ graph-rec-scheme$
begin

lemma $[simp]:$

$sccs\ (empty-state\ (\!sccs = s,\ lowlink = l,\ tj-stack = t\!)) = s$
 $lowlink\ (empty-state\ (\!sccs = s,\ lowlink = l,\ tj-stack = t\!)) = l$
 $tj-stack\ (empty-state\ (\!sccs = s,\ lowlink = l,\ tj-stack = t\!)) = t$
 $\langle proof \rangle$

lemma $sccs-more-cong[cong]:state.more\ s = state.more\ s' \implies sccs\ s = sccs\ s'$
 $\langle proof \rangle$

lemma $lowlink-more-cong[cong]:state.more\ s = state.more\ s' \implies lowlink\ s =$
 $lowlink\ s'$
 $\langle proof \rangle$

lemma $tj-stack-more-cong[cong]:state.more\ s = state.more\ s' \implies tj-stack\ s =$
 $tj-stack\ s'$
 $\langle proof \rangle$

lemma $[simp]:$

$s(\!state.more := (\!sccs = sc,\ lowlink = l,\ tj-stack = t\!))$
 $= s(\!sccs := sc,\ lowlink := l,\ tj-stack := t\!)$
 $\langle proof \rangle$

end

locale $Tarjan-invar = Tarjan +$
 $DFS-invar\ \mathbf{where}\ param = tarjan-params$

context $Tarjan-def\ \mathbf{begin}$

lemma $Tarjan-invar-eq[simp]:$

$DFS-invar\ G\ tarjan-params\ s \longleftrightarrow Tarjan-invar\ G\ s\ (\mathbf{is}\ ?D \longleftrightarrow ?T)$

⟨proof⟩
end

2.7.3 Correctness Proof

context Tarjan begin

lemma *i-tj-stack-discovered*:

is-invar ($\lambda s. \text{set } (tj\text{-stack } s) \subseteq \text{dom } (discovered\ s)$)

⟨proof⟩

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

lemma *i-tj-stack-distinct*:

is-invar ($\lambda s. \text{distinct } (tj\text{-stack } s)$)

⟨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 ($\lambda s. \forall k < \text{length } (tj\text{-stack } s). \forall j < k. \delta\ s\ (tj\text{-stack } s\ !\ j) > \delta\ s\ (tj\text{-stack } s\ !\ k)$)

⟨proof⟩

end end

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

lemma *tj-stack-incr-disc*:

assumes $k < \text{length } (tj\text{-stack } s)$

and $j < k$

shows $\delta\ s\ (tj\text{-stack } s\ !\ j) > \delta\ s\ (tj\text{-stack } s\ !\ k)$

⟨proof⟩

lemma *tjs-disc-dw-tw*:

fixes u

defines $dw \equiv \text{dropWhile } ((\neq)\ u)\ (tj\text{-stack } s)$

defines $tw \equiv \text{takeWhile } ((\neq)\ u)\ (tj\text{-stack } s)$

assumes $x \in \text{set } dw\ y \in \text{set } tw$

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 ($\lambda s. \bigcup (\text{sccs } s) \subseteq \text{dom } (finished\ s) \wedge \text{set } (stack\ s) \subseteq \text{set } (tj\text{-stack } s)$)

⟨proof⟩

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

$is\text{-invar } (\lambda s. set (tj\text{-stack } s) \subseteq set (stack\ s) \cup dom (finished\ s))$
 <proof>

lemma *i-finished-ss-sccs-tj-stack*:
 $is\text{-invar } (\lambda s. dom (finished\ s) \subseteq \bigcup (sccs\ s) \cup set (tj\text{-stack } s))$
 <proof>

end end

context *Tarjan-invar* **begin**
lemmas *finished-ss-sccs-tj-stack* =
 $i\text{-finished-ss-sccs-tj-stack}[THEN\ make\text{-invar}\text{-thm}]$

lemmas *tj-stack-ss-stack-finished* =
 $i\text{-tj-stack-ss-stack-finished}[THEN\ make\text{-invar}\text{-thm}]$

lemma *sccs-finished*:
 $\bigcup (sccs\ s) \subseteq dom (finished\ s)$
 <proof>

lemma *stack-ss-tj-stack*:
 $set (stack\ s) \subseteq set (tj\text{-stack } s)$
 <proof>

lemma *hd-stack-in-tj-stack*:
 $stack\ s \neq [] \implies hd (stack\ s) \in set (tj\text{-stack } s)$
 <proof>

end

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

end end

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

context **begin interpretation** *timing-syntax* <proof>

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

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

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

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

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

context begin interpretation *timing-syntax* $\langle proof \rangle$

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

context *Tarjan-invar* **begin context begin interpretation** *timing-syntax* $\langle proof \rangle$

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

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

lemma *nc-sccs-eq-reachable*:
assumes *NC*: $\neg \text{cond } s$
shows $\text{reachable} = \bigcup (sccs\ s)$
 $\langle proof \rangle$
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* \equiv *do* {
ASSERT (fb-graph G);
s \leftarrow *tarjan.it-dfs TYPE('a) G*;
RETURN (sccs s) }

definition *tarjan-spec G* \equiv *do* {
ASSERT (fb-graph G);
SPEC (λ *sccs.* (\forall *scc* \in *sccs.* *is-scc (g-E G) scc*)
 $\wedge \bigcup$ *sccs* = *tarjan.reachable TYPE('a) G*})

lemma *tarjan-correct*:
tarjan G \leq *tarjan-spec G*
<proof>

end