

A Framework for Verifying Depth-First Search Algorithms

Peter Lammich and René Neumann

March 17, 2025

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, '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 $\bigwedge$ *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

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

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

$st = v \# stack\ s;$

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

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

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

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

$finish\ u\ s \equiv let$

$f = (finished\ s)(u \mapsto counter\ s); c = counter\ s + 1;$

$st = tl\ (stack\ s)$

$in\ s \parallel finished := f, counter := c, stack := st \parallel$

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

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

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

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

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

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

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

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

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

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

definition *empty-state e*

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

lemma *empty-state-simps[simp]:*

$counter \ (empty-state \ e) = 0$
 $discovered \ (empty-state \ e) = Map.empty$
 $finished \ (empty-state \ e) = Map.empty$
 $pending \ (empty-state \ e) = \{\}$
 $stack \ (empty-state \ e) = []$
 $tree-edges \ (empty-state \ e) = \{\}$
 $back-edges \ (empty-state \ e) = \{\}$
 $cross-edges \ (empty-state \ e) = \{\}$
 $state.more \ (empty-state \ e) = e$
 $\langle proof \rangle$

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

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

definition *do-back-edge*

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

where

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

```

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

lemma $cond\text{-alt}$:

$cond = (\lambda s. (V0 \subseteq \text{dom } (discovered\ s) \longrightarrow \text{stack } s \neq [])) \wedge \neg \text{is-break param } s)$
 $\langle \text{proof} \rangle$

definition $get\text{-}pending ::$

$('v, 'es) \text{ state-scheme} \Rightarrow ('v \times 'v \text{ option} \times ('v, 'es) \text{ state-scheme}) \text{ nres}$

— Get topmost stack node and a pending edge if any. The pending edge is removed.

where $get\text{-}pending\ s \equiv \text{do } \{$
 $\text{let } u = \text{hd } (stack\ s);$
 $\text{let } Vs = \text{pending } s \text{ “ } \{u\};$

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

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

where

$step\ s \equiv$
 $\text{if is-empty-stack } s \text{ then do } \{$
 $v0 \leftarrow \text{get-new-root } s;$
 $\text{do-new-root } v0\ s$
 $\}$ $\text{else do } \{$
 $(u, Vs, s) \leftarrow \text{get-pending } s;$
 $\text{case } Vs \text{ of}$
 $\text{None} \Rightarrow \text{do-finish } u\ s$
 $| \text{Some } v \Rightarrow \text{do } \{$
 $\text{if is-discovered } v\ s \text{ then}$
 $\text{do-known-edge } u\ v\ s$
 else

```

    }
    }

```

definition $it_dfs \equiv init \gg= WHILE\ cond\ step$

definition $it_dfsT \equiv init \gg= WHILET\ cond\ step$

end

1.1.4 Invariants

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

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

1. First, we prove the invariant wrt. the parameterization in the *param-DFS* locale.
2. Next, we transfer the invariant to the *DFS-invar*-locale.

locale *param-DFS* =

fb-graph $G + param-DFS-defs\ G\ param$

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

and $param :: ('v, 'es)\ parameterization$

begin

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

— Predicate that states that I is an invariant.

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

end

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

locale *DFS-invar* =

param-DFS $G\ param$

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

and $param :: ('v, 'es)\ parameterization$

+

fixes $s :: ('v, 'es)\ state-scheme$

assumes $rwof: rwof\ init\ cond\ step\ s$

begin

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

— 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*: $\llbracket \text{nofail it-dfs}; \text{DFS-invar } G \text{ param } s; \text{ cond } s \rrbracket$
 $\implies \text{step } s \leq \text{SPEC } (\text{DFS-invar } G \text{ param})$
 — A step preserves the (best) invariant.
<proof>

lemma *DFS-invar-step'*: $\llbracket \text{nofail } (\text{step } s); \text{DFS-invar } G \text{ param } s; \text{ cond } s \rrbracket$
 $\implies \text{step } s \leq \text{SPEC } (\text{DFS-invar } G \text{ param})$
<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* $u\ v\ s' \equiv \exists s. \text{pre-edge-selected } u\ v\ s \wedge$
 $v \in \text{dom}(\text{discovered } s) \wedge v \notin \text{dom}(\text{finished } s)$
 $\wedge s' = \text{back-edge } u\ v\ (s \setminus \text{pending} := \text{pending } s - \{(u, v)\})$

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

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

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

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

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

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

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

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

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

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

$tree_edges\ s' \supseteq tree_edges\ s;$
 $state.more\ s' = state.more\ s]$
 $\implies on_new_root\ param\ v\ s' \leq_n$
 $SPEC\ (\lambda x. DFS_invar\ G\ param\ (s'(|state.more := x|))$
 $\longrightarrow I\ (s'(|state.more := x|)))$
assumes *finish*: $\bigwedge s\ s'\ u.$
 $\llbracket DFS_invar\ G\ param\ s; I\ s; cond\ s; \neg is_break\ param\ s;$
 $stack\ s \neq []; u = hd\ (stack\ s);$
 $pending\ s\ \text{“}\{u\} = \{\};$
 $s' = finish\ u\ s\rrbracket$
 $\implies on_finish\ param\ u\ s' \leq_n$
 $SPEC\ (\lambda x. DFS_invar\ G\ param\ (s'(|state.more := x|))$
 $\longrightarrow I\ (s'(|state.more := x|)))$
assumes *cross-edge*: $\bigwedge s\ s'\ u\ v.$
 $\llbracket DFS_invar\ G\ param\ s; I\ s; cond\ s; \neg is_break\ param\ s;$
 $stack\ s \neq []; (u, v) \in pending\ s; u = hd\ (stack\ s);$
 $v \in dom\ (discovered\ s); v \in dom\ (finished\ s);$
 $s' = cross_edge\ u\ v\ (s(|pending := pending\ s - \{(u, v)\}|))\rrbracket$
 $\implies on_cross_edge\ param\ u\ v\ s' \leq_n$
 $SPEC\ (\lambda x. DFS_invar\ G\ param\ (s'(|state.more := x|))$
 $\longrightarrow I\ (s'(|state.more := x|)))$
assumes *back-edge*: $\bigwedge s\ s'\ u\ v.$
 $\llbracket DFS_invar\ G\ param\ s; I\ s; cond\ s; \neg is_break\ param\ s;$
 $stack\ s \neq []; (u, v) \in pending\ s; u = hd\ (stack\ s);$
 $v \in dom\ (discovered\ s); v \notin dom\ (finished\ s);$
 $s' = back_edge\ u\ v\ (s(|pending := pending\ s - \{(u, v)\}|))\rrbracket$
 $\implies on_back_edge\ param\ u\ v\ s' \leq_n$
 $SPEC\ (\lambda x. DFS_invar\ G\ param\ (s'(|state.more := x|))$
 $\longrightarrow I\ (s'(|state.more := x|)))$
shows *is-invar* I
 $\langle proof \rangle$

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*: $\bigwedge u\ v\ s. on_back_edge\ param\ u\ v\ s = on_cross_edge\ param\ u\ v\ s$

assumes *init*: $on_init\ param \leq_n SPEC\ (\lambda x. I\ (empty_state\ x))$

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

$\llbracket DFS_invar\ G\ param\ s; I\ s; cond\ s; \neg is_break\ param\ s;$
 $stack\ s = []; v0 \in V0; v0 \notin dom\ (discovered\ s);$
 $s' = new_root\ v0\ s\rrbracket$
 $\implies on_new_root\ param\ v0\ s' \leq_n$
 $SPEC\ (\lambda x. DFS_invar\ G\ param\ (s'(|state.more := x|))$
 $\longrightarrow I\ (s'(|state.more := x|)))$

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

$\llbracket DFS_invar\ G\ param\ s; I\ s; cond\ s; \neg is_break\ param\ s;$
 $stack\ s \neq []; u = hd\ (stack\ s);$
 $pending\ s\ \text{“}\{u\} = \{\};$

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

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*: $\bigwedge s \ s' \ v.$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

— Establish a DFS invariant not taking into account the parameterization.

assumes *init*: $\bigwedge e. I\ (empty\text{-}state\ e)$

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

$\llbracket I\ s; cond\ s; DFS\text{-}invar\ G\ param\ s; DFS\text{-}invar\ G\ param\ s';$

$stack\ s = []; v0 \notin dom\ (discovered\ s); v0 \in V0;$

$s' = new\text{-}root\ v0\ s \langle state.more := e \rangle \rrbracket$

$\implies I\ s'$

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

$\llbracket I\ s; cond\ s; DFS\text{-}invar\ G\ param\ s; DFS\text{-}invar\ G\ param\ s';$

$stack\ s \neq []; pending\ s\ \text{“}\{u\} = \{ \};$

$u = hd\ (stack\ s); s' = finish\ u\ s \langle state.more := e \rangle \rrbracket$

$\implies I\ s'$

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

$\llbracket I\ s; cond\ s; DFS\text{-}invar\ G\ param\ s; DFS\text{-}invar\ G\ param\ s';$

$stack\ s \neq []; v \in pending\ s\ \text{“}\{u\}; v \in dom\ (discovered\ s);$

$v \in dom\ (finished\ s);$

$u = hd\ (stack\ s);$

$s' = (cross\text{-}edge\ u\ v\ (s \langle pending := pending\ s - \{(u, v)\} \rangle)) \langle state.more := e \rangle \rrbracket$

$\implies I\ s'$

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

$\llbracket I\ s; cond\ s; DFS\text{-}invar\ G\ param\ s; DFS\text{-}invar\ G\ param\ s';$

$stack\ s \neq []; v \in pending\ s\ \text{“}\{u\}; v \in dom\ (discovered\ s); v \notin dom\ (finished$

$s);$

$u = hd\ (stack\ s);$

$s' = (back\text{-}edge\ u\ v\ (s \langle pending := pending\ s - \{(u, v)\} \rangle)) \langle state.more := e \rangle \rrbracket$

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

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

lemma *finished-imp-succ-discovered*:

v ∈ dom (finished s) ⇒ w ∈ succ v ⇒ w ∈ dom (discovered s)
 ⟨*proof*⟩

lemma *pending-reachable*: *pending s ⊆ reachable × reachable*

⟨*proof*⟩

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

⟨*proof*⟩

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

assumes *u ∈ dom (discovered s)*

and *pending s “ {u} = {}*

and *v ∈ succ u*

shows *v ∈ dom (discovered s)*

⟨*proof*⟩

lemma *nc-finished-eq-reachable*:

assumes *NC: ¬ cond s ¬ is-break param s*

shows *dom (finished s) = reachable*

⟨*proof*⟩

lemma *nc-V0-finished*:

assumes *NC: ¬ cond s ¬ is-break param s*

shows *V0 ⊆ dom (finished s)*

⟨*proof*⟩

lemma *nc-discovered-eq-finished*:

assumes *NC: ¬ cond s ¬ is-break param s*

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

⟨*proof*⟩

lemma *nc-discovered-eq-reachable*:

assumes *NC: ¬ cond s ¬ is-break param s*

shows *dom (discovered s) = reachable*

⟨*proof*⟩

lemma *nc-fin-closed*:

```

    assumes NC:  $\neg \text{cond } s$ 
    assumes NB:  $\neg \text{is-break param } s$ 
    shows  $E'' \text{dom (finished } s) \subseteq \text{dom (finished } s)$ 
    <proof>

end

```

1.1.6 Total Correctness

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

```

context param-DFS begin
  definition param-dfs-variant  $\equiv \text{inv-image}$ 
    (finite-psupset reachable <*<lex*> finite-psubset <*<lex*> less-than)
    ( $\lambda s. (\text{dom (discovered } s), \text{pending } s, \text{length (stack } s))$ )

  lemma param-dfs-variant-wf[simp, intro!]:
    assumes [simp, intro!]: finite reachable
    shows wf param-dfs-variant
    <proof>

  lemma param-dfs-variant-step:
    assumes A: DFS-invar G param s cond s nofail it-dfs
    shows step s  $\leq \text{SPEC } (\lambda s'. (s', s) \in \text{param-dfs-variant})$ 
    <proof>

end

```

```

context param-DFS begin
  lemma it-dfsT-eq-it-dfs:
    assumes [simp, intro!]: finite reachable
    shows it-dfsT = it-dfs
    <proof>
end

```

1.1.7 Non-Failing Parameterization

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

```

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

```

$\text{nofail } (\text{on-init param})$

assumes *nofail-on-new-root*:
 $\text{pre-on-new-root } v0 \ s \implies \text{nofail } (\text{on-new-root param } v0 \ s)$

assumes *nofail-on-finish*:
 $\text{pre-on-finish } u \ s \implies \text{nofail } (\text{on-finish param } u \ s)$

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

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

assumes *nofail-on-discover*:
 $\text{pre-on-discover } u \ v \ s \implies \text{nofail } (\text{on-discover param } u \ v \ s)$

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*: $\text{init} \leq_n \text{SPEC } (\text{DFS-invar } G \text{ param})$
 $\langle \text{proof} \rangle$

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

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

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

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

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 $\text{it-dfsT} \leq \text{SPEC } P$
 $\langle \text{proof} \rangle$

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* $s \equiv$

$\neg \delta\ s\ v < \varphi\ s\ v$

$(\forall v \in dom\ (finished\ s). \delta\ s\ v < \varphi\ s\ v)$

$\neg v \neq w \longrightarrow \delta\ s\ v \neq \delta\ s\ w \wedge \varphi\ s\ v \neq \varphi\ s\ w$

$\neg$ Can't use *card dom* = *card ran* as the maps may be infinite ...

$\wedge (\forall v \in dom\ (discovered\ s). \forall w \in dom\ (discovered\ s). v \neq w \longrightarrow \delta\ s\ v \neq \delta\ s\ w)$

$\wedge (\forall v \in dom\ (finished\ s). \forall w \in dom\ (finished\ s). v \neq w \longrightarrow \varphi\ s\ v \neq \varphi\ s\ w)$

$\neg \delta\ s\ v < counter \wedge \varphi\ s\ v < counter$

$\wedge (\forall v \in dom\ (discovered\ s). \delta\ s\ v < counter\ s)$

$\wedge (\forall v \in dom\ (finished\ s). \varphi\ s\ v < counter\ s)$

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

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*

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

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

$\langle \text{proof} \rangle$

lemma *tl-lt-stack-hd-discover*:

assumes *notempty*: $\text{stack } s \neq []$
and $x \in \text{set } (\text{tl } (\text{stack } s))$
shows $\delta s x < \delta s (\text{hd } (\text{stack } s))$

$\langle \text{proof} \rangle$

lemma *stack-nth-order*:

assumes $l: i < \text{length } (\text{stack } s) \ j < \text{length } (\text{stack } s)$
shows $\delta s (\text{stack } s ! i) < \delta s (\text{stack } s ! j) \longleftrightarrow i > j$ (**is** $\delta s ?i < \delta s ?j \longleftrightarrow -$)

$\langle \text{proof} \rangle$

end end

1.2.2 Paranthesis Theorem

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

definition *parenthesis* $s \equiv$

$\forall v \in \text{dom } (\text{discovered } s). \forall w \in \text{dom } (\text{discovered } s).$
 $\delta s v < \delta s w \wedge v \in \text{dom } (\text{finished } s) \longrightarrow ($
 $\varphi s v < \delta s w \text{ — disjoint}$
 $\vee (\varphi s v > \delta s w \wedge w \in \text{dom } (\text{finished } s) \wedge \varphi s w < \varphi s v))$

lemma *i-parenthesis*: *is-invar parenthesis*

$\langle \text{proof} \rangle$

end end

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

lemma *parenthesis*:

assumes $v \in \text{dom } (\text{finished } s) \ w \in \text{dom } (\text{discovered } s)$
and $\delta s v < \delta s w$
shows $\varphi s v < \delta s w \text{ — disjoint}$
 $\vee (\varphi s v > \delta s w \wedge w \in \text{dom } (\text{finished } s) \wedge \varphi s w < \varphi s v)$

$\langle \text{proof} \rangle$

lemma *parenthesis-contained*:

assumes $v \in \text{dom } (\text{finished } s) \ w \in \text{dom } (\text{discovered } s)$
and $\delta s v < \delta s w \ \varphi s v > \delta s w$
shows $w \in \text{dom } (\text{finished } s) \wedge \varphi s w < \varphi s v$

$\langle \text{proof} \rangle$

lemma *parenthesis-disjoint*:

assumes $v \in \text{dom } (\text{finished } s) \ w \in \text{dom } (\text{discovered } s)$
and $\delta s v < \delta s w \ \varphi s w > \varphi s v$
shows $\varphi s v < \delta s w$

$\langle \text{proof} \rangle$

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

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*:
 $edges\ s \subseteq E \cap reachable \times UNIV$
 $\langle proof \rangle$

lemma *nc-edges-covered*:
assumes $\neg cond\ s\ \neg is-break\ param\ s$
shows $E \cap reachable \times UNIV = edges\ s$
 $\langle proof \rangle$

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

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

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

lemma *edges-disjoint*:
 $back-edges\ s \cap cross-edges\ s = \{\}$
 $back-edges\ s \cap tree-edges\ s = \{\}$
 $cross-edges\ s \cap tree-edges\ s = \{\}$
 $\langle proof \rangle$

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

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

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

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

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

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

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

assumes *stack s* $\neq []$
and $v \in set \ (tl \ (stack \ s))$
shows $(v, hd \ (stack \ s)) \in (tree-edges \ s)^+$
 $\langle proof \rangle$

end

context *param-DFS begin*

definition *tree-discovered-inv s* $\equiv$

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

lemma *i-tree-discovered-inv*:

is-invar tree-discovered-inv
 $\langle proof \rangle$

lemmas (**in** *DFS-invar*) *tree-discovered-inv* =

i-tree-discovered-inv[*THEN make-invar-thm*]

lemma (**in** *DFS-invar*) *discovered-iff-tree-path*:

$v \notin V0 \implies v \in dom \ (discovered \ s) \longleftrightarrow (\exists v0 \in V0. (v0, v) \in (tree-edges \ s)^+)$
 $\langle proof \rangle$

lemma *i-tree-one-predecessor*:

is-invar $(\lambda s. \forall (v, v') \in tree-edges \ s. \forall y. y \neq v \longrightarrow (y, v') \notin tree-edges \ s)$
 $\langle proof \rangle$

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

assumes $(v, w) \in tree-edges \ s$
and $a \neq v$
shows $(a, w) \notin tree-edges \ s$
 $\langle proof \rangle$

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
 $\langle \text{proof} \rangle$
end end

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

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 $(v, w) \in (\text{tree-edges } s)^+$
 $\langle \text{proof} \rangle$

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 $(v, w) \in (\text{tree-edges } s)^+$
 $\langle \text{proof} \rangle$

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

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 $(v, w) \in (\text{tree-edges } s)^+$
 $\langle \text{proof} \rangle$

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 $(v, w) \in (\text{tree-edges } s)^+$
 $\langle \text{proof} \rangle$
end end

Properties of Cross Edges

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

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

end end

context *DFS-invar* **begin context begin interpretation** *timing-syntax* $\langle \text{proof} \rangle$
lemmas *cross-edges-finished*
 $= i\text{-cross-edges-finished}[THEN \text{ make-invar-thm}]$

lemma *cross-edges-target-finished*:
 $(u,v) \in \text{cross-edges } s \implies v \in \text{dom } (\text{finished } s)$
 $\langle \text{proof} \rangle$

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

lemma *cross-edge-unequal*:
assumes *cross*: $(v,w) \in \text{cross-edges } s$
shows $v \neq w$
 $\langle \text{proof} \rangle$

end end

Properties of Back Edges

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

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

end end

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

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

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

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

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

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

$\langle proof \rangle$

lemma *back-edge-finished*:

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

and $w \in dom\ (finished\ s)$

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

$\langle proof \rangle$

end end

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

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

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

$v \in dom\ (discovered\ s) \wedge w \in dom\ (discovered\ s)$

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

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

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

$\langle proof \rangle$

end end

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

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

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

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

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

$\langle proof \rangle$

lemma *finished-imp-back-edge*:

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

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

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

$\langle proof \rangle$

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

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

$w \notin dom\ (finished\ s);$

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

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

$\langle proof \rangle$

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

assumes $v \in dom\ (finished\ s)$

and $(v, v) \in E$

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

$\langle proof \rangle$

end end

context *DFS-invar* **begin**

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

lemma *tree-cross-acyclic*:

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

end

lemma *cycle-contains-back-edge*:

assumes *cycle*: $(u, u) \in (\textit{edges } s)^+$

shows $\exists v w. (u, v) \in (\textit{edges } s)^* \wedge (v, w) \in \textit{back-edges } s \wedge (w, u) \in (\textit{edges } s)^*$

$\langle \text{proof} \rangle$

lemma *cycle-needs-back-edge*:

assumes *back-edges* $s = \{\}$

shows *acyclic* (*edges* s)

$\langle \text{proof} \rangle$

lemma *back-edge-closes-cycle*:

assumes *back-edges* $s \neq \{\}$

shows $\neg \textit{acyclic} (\textit{edges } s)$

$\langle \text{proof} \rangle$

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

back-edges $s \neq \{\} \implies \neg \textit{acyclic} (E \cap \textit{reachable} \times \textit{UNIV})$

$\langle \text{proof} \rangle$

lemma *cycle-iff-back-edges*:

acyclic (*edges* s) $\longleftrightarrow \textit{back-edges } s = \{\}$

$\langle \text{proof} \rangle$

end

1.2.4 White Path Theorem

context *DFS* **begin**

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

definition *white-path* **where**

white-path $s \ x \ y \equiv x \neq y$

$\longrightarrow (\exists p. \textit{path } E \ x \ p \ y \wedge$
 $(\delta \ s \ x < \delta \ s \ y \wedge (\forall v \in \textit{set } (tl \ p). \delta \ s \ x < \delta \ s \ v)))$

lemma *white-path*:

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

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

$\langle proof \rangle$
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

$scc-root'\ E\ s\ v\ scc \longleftrightarrow is-scc\ E\ scc$

$\wedge v \in scc$

$\wedge v \in dom\ (discovered\ s)$

$\wedge scc \cap dom\ (discovered\ s) \subseteq (tree-edges\ s)^* \text{ `` } \{v\}$

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 $scc \cap dom\ (discovered\ s) \subseteq (tree-edges\ s)^* \text{ `` } \{v\}$

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-rtrancI*:

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 $\varphi \ s \ v \geq \varphi \ s \ x$
 $\langle proof \rangle$

lemma *scc-root-is-Min-disc*:
assumes *scc-root* $s \ v \ scc$
shows $Min \ (\delta \ s \ ' \ (scc \cap dom \ (discovered \ s))) = \delta \ s \ v$ (**is** $Min \ ?S = -$)
 $\langle proof \rangle$

lemma *Min-disc-is-scc-root*:
assumes $v \in scc \ v \in dom \ (discovered \ s)$
and *is-scc* $E \ scc$
and *min*: $\delta \ s \ v = Min \ (\delta \ s \ ' \ (scc \cap dom \ (discovered \ s)))$
shows *scc-root* $s \ v \ scc$
 $\langle proof \rangle$

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

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

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

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

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

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* $\langle$ DFS framework: Unfolding theorems to prepare term for automatic refinement $\rangle$

```
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
```

```

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

```

definition *gen-cond s*
 $\equiv (V0 \subseteq \{v. \text{gds-is-discovered gds v s}\} \longrightarrow \neg \text{gds-is-empty-stack gds s})$
 $\wedge \neg \text{gds-is-break gds s}$

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* $\equiv$
 $\text{gen-rwof s} \wedge \text{gds-is-empty-stack gds s} \wedge \neg \text{gds-is-break gds s}$
 $\wedge v0 \in V0 - \text{gen-discovered s}$

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

definition *pre-back-edge u v s0 s* $\equiv$
 $\text{post-get-pending u (Some v) s0 s} \wedge \text{gds-is-discovered gds v s}$
 $\wedge \neg \text{gds-is-finished gds v s}$

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*: $\llbracket gen\text{-cond } s; gen\text{-rwof } s \rrbracket$
 $\implies gen\text{-step } s = gen\text{-step-assert } s$
 $\langle proof \rangle$

lemma *gen-dfs-eq-assert*: $gen\text{-dfs} = gen\text{-dfs-assert}$
 $\langle proof \rangle$

lemma *gen-dfsT-eq-assert*: $gen\text{-dfsT} = gen\text{-dfsT-assert}$
 $\langle proof \rangle$

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

```

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

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

end

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

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

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

  definition do-action bf ef s ≡ do {
    s ← bf s;
    e ← ef s;
    RETURN (upd-ext (λ-. e) s)
  }

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

  definition do-new-root v0

```

```

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

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

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

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

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

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

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

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

sublocale gen-dfs-defs gds V0 ⟨proof⟩
end

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

context param-DFS-defs begin

```

```

definition gbs ≡ (|
gbs-init = RETURN o empty-state,
gbs-is-empty-stack = is-empty-stack ,

```

```

    gbs-new-root = RETURN oo new-root ,
    gbs-get-pending = get-pending ,
    gbs-finish = RETURN oo finish ,
    gbs-is-discovered = is-discovered ,
    gbs-is-finished = is-finished ,
    gbs-back-edge = RETURN ooo back-edge ,
    gbs-cross-edge = RETURN ooo cross-edge ,
    gbs-discover = RETURN ooo discover
  ⌋

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

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

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

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

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

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

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

end

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

```

1.4.2 Refinement Between DFS Implementations

```

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

locale gen-dfs-refine =
  c: gen-dfs gdsi V0i + a: gen-dfs gds V0 + gen-dfs-refine-defs gdsi V0i gds V0
  for gdsi V0i gds V0 +
  fixes V S
  assumes BIJV[relator-props]: bijective V
  assumes V0-param[param]: (V0i, V0) ∈ ⟨V⟩set-rel
  assumes is-discovered-param[param]:
    (gds-is-discovered gdsi, gds-is-discovered gds) ∈ V → S → bool-rel

```

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

begin
term *gds-is-discovered gdsi*

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

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'*
 $\langle proof \rangle$

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

lemma *gen-dfs-refine*[*refine*]: $c.\text{gen-dfs} \leq \Downarrow S a.\text{gen-dfs}$
 $\langle \text{proof} \rangle$

lemma *gen-dfsT-refine*[*refine*]: $c.\text{gen-dfsT} \leq \Downarrow S a.\text{gen-dfsT}$
 $\langle \text{proof} \rangle$

end

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

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

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

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

assumes *is-break-param*[*param*]:
 $(\text{is-break } parami, \text{is-break } param) \in S \rightarrow \text{bool-rel}$

assumes *gbs-init-refine*[*refine*]: $(ei, e) \in ES \implies \text{gbs-init } gbsi \text{ ei} \leq \Downarrow S (\text{gbs-init } gbs \text{ e})$

assumes *gbs-new-root-refine*[*refine*]:
 $\llbracket a.\text{pre-new-root } v0 \text{ s}; (v0i, v0) \in V; (si, s) \in S \rrbracket$
 $\implies \text{gbs-new-root } gbsi \text{ v0i si} \leq \Downarrow S (\text{gbs-new-root } gbs \text{ v0 s})$

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

assumes *gbs-finish-refine*[*refine*]:
 $\llbracket a.\text{pre-finish } v \text{ s0 s}; (vi, v) \in V; (si, s) \in S \rrbracket$
 $\implies \text{gbs-finish } gbsi \text{ vi si} \leq \Downarrow S (\text{gbs-finish } gbs \text{ v s})$

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* {

$s \leftarrow \text{gds-init } \text{gds};$

FOREACHci

($\lambda it\ s.$

$\text{gen-rwof } s$

$\wedge (\neg \text{gds-is-break } \text{gds } s \longrightarrow \text{gds-is-empty-stack } \text{gds } s)$

$\wedge V0-it \subseteq \text{gen-discovered } s)$

$V0$

(*Not* $o\ \text{gds-is-break } \text{gds}$)

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

$\text{let } \text{---ghost}: s0 = s;$

if $\text{gds-is-discovered } \text{gds } v0\ s$ *then*

RETURN s

else do {

$s \leftarrow \text{gds-new-root } \text{gds } v0\ s;$

WHILEIT

($\lambda s.$ $\text{gen-rwof } s \wedge \text{insert } v0\ (\text{gen-discovered } s0) \subseteq \text{gen-discovered } s)$

($\lambda s.$ $\neg \text{gds-is-break } \text{gds } s \wedge \neg \text{gds-is-empty-stack } \text{gds } s)$

tr-impl-while-body s

}

}) s

}

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

tailrec-impl $\equiv$ *do* {

$s \leftarrow \text{gds-init } \text{gds};$

FOREACHci

($\lambda it\ s.$

$\text{gen-rwof } s$

$\wedge (\neg \text{gds-is-break } \text{gds } s \longrightarrow \text{gds-is-empty-stack } \text{gds } s)$

$\wedge V0-it \subseteq \text{gen-discovered } s)$

$V0$

(*Not* $o\ \text{gds-is-break } \text{gds}$)

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

$\text{let } \text{---ghost}: s0 = s;$

if $\text{gds-is-discovered } \text{gds } v0\ s$ *then*

RETURN s

else do {

$s \leftarrow \text{gds-new-root } \text{gds } v0\ s;$

WHILEI

($\lambda s.$ $\text{gen-rwof } s \wedge \text{insert } v0\ (\text{gen-discovered } s0) \subseteq \text{gen-discovered } s)$

($\lambda s.$ $\neg \text{gds-is-break } \text{gds } s \wedge \neg \text{gds-is-empty-stack } \text{gds } s)$

($\lambda s.$ *do* {

(u, Vs, s) $\leftarrow \text{gds-get-pending } \text{gds } s;$

case Vs *of*

$\text{None} \Rightarrow \text{gds-finish } \text{gds } u\ s$

```

    | 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:
     $\llbracket \text{pre-new-root } v0\ s \rrbracket$ 
     $\Rightarrow$  gds-new-root gds v0 s  $\leq_n$  SPEC ( $\lambda s'$ .
      insert v0 (gen-discovered s)  $\subseteq$  gen-discovered s')
  assumes get-pending-incr:
     $\llbracket \text{pre-get-pending } s \rrbracket \Rightarrow$  gds-get-pending gds s  $\leq_n$  SPEC ( $\lambda(-, s')$ .
      gen-discovered s  $\subseteq$  gen-discovered s')
   $\llbracket \text{pre-get-pending } s \rrbracket \Rightarrow$  gds-get-pending gds s  $\leq_n$  SPEC ( $\lambda(-, s')$ .
    gen-discovered s  $\subseteq$  gen-discovered s')
  assumes finish-incr:  $\llbracket \text{pre-finish } u\ s0\ s \rrbracket$ 
     $\Rightarrow$  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:

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

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

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

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

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

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

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

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

```

       $\wedge \text{gen-discovered } s \subseteq \text{gen-discovered } s')$ 
     $\langle \text{proof} \rangle$ 

  end

  lemma gen-step-disc-incr:
    assumes nofail gen-dfs
    assumes gen-rwof s insert v0 (gen-discovered s0)  $\subseteq$  gen-discovered s
    assumes  $\neg \text{gds-is-break gds s} \wedge \neg \text{gds-is-empty-stack gds s}$ 
    shows gen-step s  $\leq$  SPEC ( $\lambda s. \text{insert v0 (gen-discovered s0)  $\subseteq$  gen-discovered s$ )
     $\langle \text{proof} \rangle$ 

  theorem tailrec-impl: tailrec-impl  $\leq$  gen-dfs
     $\langle \text{proof} \rangle$ 

  lemma tr-impl-while-body-gen-step:
    assumes [simp]:  $\neg \text{gds-is-empty-stack gds s}$ 
    shows tr-impl-while-body s  $\leq$  gen-step s
     $\langle \text{proof} \rangle$ 

  lemma tailrecT-impl: tailrec-implT  $\leq$  gen-dfsT
     $\langle \text{proof} \rangle$ 

end
end

```

1.6 Recursive DFS Implementation

```

theory Rec-Impl
imports General-DFS-Structure
begin

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

    definition gen-step' s  $\equiv$  do { ASSERT (gen-rwof s);
      if gds-is-empty-stack gds s then do {
        v0  $\leftarrow$  SPEC ( $\lambda v0. v0 \in V0 \wedge \neg \text{gds-is-discovered gds v0 s}$ );
        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 $gen\text{-}dfs' \equiv gds\text{-}init\ gds \ggg WHILE\ gen\text{-}cond\ gen\text{-}step'$
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'

```

```

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

```

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

```

    }
  })
  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* (λ*s*. *stack* *s* = [] ∧ *pending* *s* = {})
assumes *new-root-spec*:
[[*pre-new-root* *v0* *s*]]
⇒ *gds-new-root* *gds* *v0* *s* ≤_n *SPEC* (λ*s'*.
stack *s'* = [*v0*] ∧ *pending* *s'* = {*v0*} × E “ {*v0*} ∧
gen-discovered *s'* = insert *v0* (*gen-discovered* *s*))
assumes *get-pending-fmt*: [[*pre-get-pending* *s*]] ⇒
do {
 let *u* = hd (*stack* *s*);
vo ← SELECT (λ*v*. (i*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* (λ*s'*.

(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: $\llbracket \text{pre-finish } u \ s0 \ s \rrbracket$
 $\Rightarrow$ gds-finish gds $u \ s \leq_n \text{SPEC } (\lambda s'.$
 pending $s' =$ pending $s \wedge$
 stack $s' =$ tl (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 $u \ v \ s0 \ s$
 $\Rightarrow$ gds-cross-edge 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 $u \ v \ s0 \ s$
 $\Rightarrow$ gds-back-edge 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 $u \ v \ s0 \ s$
 $\Rightarrow$ gds-discover gds $u \ v \ s \leq_n \text{SPEC } (\lambda s'.$
 pending $s' =$ pending $s \cup (\{v\} \times E''\{v\}) \wedge$ stack $s' = v \#$ stack $s \wedge$
 gen-discovered $s' =$ insert v (gen-discovered s)

begin

lemma gen-step'-refine:

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

lemma gen-dfs'-refine: gen-dfs' $\leq$ gen-dfs

<proof>

lemma gen-rwof'-imp-rwof:

assumes NF: nofail gen-dfs

assumes A: gen-rwof' s

shows gen-rwof s

<proof>

lemma reachable-invar:

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

$\langle \text{proof} \rangle$

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* $u \text{ vo } s0 \ s \equiv$

$\text{gen-rwof}' s0$
 $\wedge \text{gen-cond } s0$
 $\wedge \text{stack } s0 \neq []$
 $\wedge u = \text{hd } (\text{stack } s0)$
 $\wedge \text{inres } (\text{choose-pending } u \text{ vo } s0) \ s$
 $\wedge \text{stack } s = \text{stack } s0$
 $\wedge (\forall x. \text{gds-is-discovered } \text{gds } x \ s = \text{gds-is-discovered } \text{gds } x \ s0)$
 ~~$\wedge (\text{gds-is-break } \text{gds } s \neq \text{gds-is-break } \text{gds } s0)$~~
 $\wedge (\text{case } \text{vo} \text{ of}$
 $\quad \text{None} \Rightarrow \text{pending } s0 \text{ ``}\{u\} = \{\}$ $\wedge \text{pending } s = \text{pending } s0$
 $\quad | \text{Some } v \Rightarrow v \in \text{pending } s0 \text{ ``}\{u\} \wedge \text{pending } s = \text{pending } s0 - \{(u, v)\})$

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: $\text{pre-get-pending } s$

assumes 2: $\text{gen-rwof}' s$

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

assumes 3: $\text{case } \text{vo} \text{ of}$

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

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

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

$\langle \text{proof} \rangle$

schematic-goal *gds-finish-refine*:

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

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

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

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

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

lemma *rec-impl*: $\text{rec-impl} \leq \text{gen-dfs}$
 $\langle \text{proof} \rangle$

end

end

1.7 Simple Data Structures

theory *Simple-Impl*
imports
 $\dots / \text{Structural} / \text{Rec-Impl}$
 $\dots / \text{Structural} / \text{Tailrec-Impl}$
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* =
 $\text{ss-stack} :: ('v \times 'v \text{ set}) \text{ list}$
 $\text{on-stack} :: 'v \text{ set}$

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 ($\langle$

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

$\rangle, s') \in \langle \text{erel} \rangle \text{simple-state-rel}$

$\langle \text{proof} \rangle$

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

(*simple-state.more-update*, *state.more-update*)
 $\in (R \rightarrow R) \rightarrow \langle R \rangle \text{simple-state-rel} \rightarrow \langle R \rangle \text{simple-state-rel}$
 $\langle \text{proof} \rangle$

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*

$\equiv \text{RETURN } (\text{ss-stack} = [], \text{on-stack} = \{\}, \text{visited} = \{\}, \dots = e)$

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

definition *is-discovered-impl* *u s* $\equiv (u \in \text{visited } s)$

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

definition *finish-impl* *u s* $\equiv \text{do } \{$

ASSERT (*ss-stack* *s* $\neq [] \wedge u \in \text{on-stack } s$);
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 $s = s \uparrow \text{ss-stack} := (u, Vs) \# \text{tl } (\text{ss-stack } s)$;
 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 $s = s \uparrow \text{ss-stack} := (v, E''\{v\}) \# \text{ss-stack } s$;
 let $s = s \uparrow \text{on-stack} := \text{insert } v (\text{on-stack } s)$;
 let $s = s \uparrow \text{visited} := \text{insert } v (\text{visited } s)$;
 RETURN s
}

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

  definition get-pending s  $\equiv \bigcup (\text{set } (\text{map } (\lambda(u, Vs). \{u\} \times Vs) (\text{ss-stack } s)))$ 
  definition get-stack s  $\equiv \text{map fst } (\text{ss-stack } s)$ 
  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  $\langle \text{proof} \rangle$ 
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*: $\text{inres } (a.\text{get-pending } s0) \ (v, \text{Some } Vs, s)$
 $\longleftrightarrow v = \text{hd } (\text{stack } s) \wedge s = s0 \setminus \{\text{hd } (\text{stack } s0)\} \cup \{\text{hd } (\text{stack } s)\}$
 $\wedge (\text{hd } (\text{stack } s0), Vs) \in \text{pending } s0$
 $\langle \text{proof} \rangle$

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*:

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

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

$$\langle \text{proof} \rangle$$

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

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

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

Link with tail-recursive implementation

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

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

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

Link to recursive implementation

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

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

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

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* $\equiv$
 $\{((\langle ss_stack_impl = si, ss_on_stack_impl = osi, ss_visited_impl = visi, \dots = mi \rangle),$
 $\langle ss_stack = s, on_stack = os, visited = vis, \dots = m \rangle) \mid$
 $si\ osi\ visi\ mi\ s\ os\ vis\ m.$
 $(si, s) \in s_rel \wedge$
 $(osi, os) \in vis_rel \wedge$
 $(visi, vis) \in vis_rel \wedge$
 $(mi, m) \in erel$
 $\}$

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 $R \equiv \langle s_rel, vis_rel, erel \rangle ss_impl_rel$

shows

$(ss_stack_impl, ss_stack) \in R \rightarrow s_rel$
 $(ss_on_stack_impl, on_stack) \in R \rightarrow vis_rel$
 $(ss_visited_impl, visited) \in R \rightarrow vis_rel$
 $(simple_state_impl.more, simple_state.more) \in R \rightarrow erel$
 $(ss_stack_impl_update, ss_stack_update) \in (s_rel \rightarrow s_rel) \rightarrow R \rightarrow R$
 $(ss_on_stack_impl_update, on_stack_update) \in (vis_rel \rightarrow vis_rel) \rightarrow R \rightarrow R$
 $(ss_visited_impl_update, visited_update) \in (vis_rel \rightarrow vis_rel) \rightarrow R \rightarrow R$
 $(simple_state_impl.more_update, simple_state.more_update) \in (erel \rightarrow erel) \rightarrow R$
 $\rightarrow R$
 $(simple_state_impl_ext, simple_state_ext) \in s_rel \rightarrow vis_rel \rightarrow vis_rel \rightarrow erel \rightarrow R$
 $\langle proof \rangle$

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

lemmas *[autoref-rel-intf]* = $REL\text{-}INTFI[of\ ssnos\text{-}impl\text{-}rel\ i\text{-}simple\text{-}state]$

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

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

end

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

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

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 $(\text{'}si, \text{'}nsi)\text{simple-state-ns-impl} =$
 $ssns\text{-}visited\text{-}impl :: \text{'}nsi$

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

```

{((|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 $R \equiv \langle ANY\text{-}rel, vis\text{-}rel, erel \rangle ssns\text{-}impl\text{-}rel$

shows

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

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

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

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

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

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

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

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

$\langle proof \rangle$

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

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

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

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

$\langle proof \rangle$

end

1.8 Restricting Nodes by Pre-Initializing Visited Set

theory *Restr-Impl*

imports *Simple-Impl*

begin

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

We now further refine the simple implementation in case that the graph has the form $G' = (rel\text{-}restrict\ E\ R, V0 - R)$ 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 $restr\text{-}rel\text{-}stackD: (ss\text{-}stack\ s, ss\text{-}stack\ s') \in \langle Id \times_r \{(U, U').\ U - R = U'\} \rangle list\text{-}rel$
and $restr\text{-}rel\text{-}vis\text{-}djD: visited\ s' \cap R = \{\}$
 $\langle proof \rangle$

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

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

end

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

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

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

locale *restricted-impl-defs* =

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

begin

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

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

definition $gbs' \equiv gbs\ \emptyset$

```

    gbs-init := λe. 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 = (λu v s. RETURN (simple-state.more
  s))
  assumes [simp]: on-back-edge parami = (λu v s. RETURN (simple-state.more
  s))

  assumes is-break-refine:
    [| (s,s')∈restr-rel R |]
    ⇒ is-break parami s ↔ is-break parami s'

  assumes on-new-root-refine:
    [| (s,s')∈restr-rel R |]
    ⇒ on-new-root parami v0 s ≤ on-new-root parami v0 s'

  assumes on-finish-refine:
    [| (s,s')∈restr-rel R |]
    ⇒ on-finish parami u s ≤ on-finish parami u s'

  assumes on-discover-refine:
    [| (s,s')∈restr-rel R |]
    ⇒ on-discover parami u v s ≤ on-discover parami u v s'

begin

  lemmas rel-def = restr-rel-def[where R=R]
  sublocale gen-param-dfs gbs' parami simple-state.more-update V0 ⟨proof⟩

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

```

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

lemma *gen-cond-param*: $(gen-cond, a.c.gen-cond) \in rel \rightarrow bool-rel$
 $\langle proof \rangle$

lemma *cross-back-id*[*simpl*]:
 $do-cross-edge\ u\ v\ s = RETURN\ s$
 $do-back-edge\ u\ v\ s = RETURN\ s$
 $a.c.do-cross-edge\ u\ v\ s = RETURN\ s$
 $a.c.do-back-edge\ u\ v\ s = RETURN\ s$
 $\langle proof \rangle$

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

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

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

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

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

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

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

lemma *do-discover-refine*[*refine*]:
 $\llbracket (s, s') \in rel; (u, u') \in Id; (v, v') \in Id; v' \notin R \rrbracket$
 $\implies do-discover\ u\ v\ s \leq \Downarrow rel\ (a.c.do-discover\ u'\ v'\ s')$
 $\langle proof \rangle$

lemma *aux-R-node-discovered*: $\llbracket (s, s') \in rel; v \in R \rrbracket \implies is-discovered-impl\ v\ s$
 $\langle proof \rangle$

lemma *re-refine-aux*: $gen-dfs \leq \Downarrow rel\ a.c.gen-dfs$
 $\langle proof \rangle$

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

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

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

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

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*: $state.more\ s = state.more\ s' \implies break\ s = break\ s'$
 $\langle proof \rangle$

lemma [*simp*]: $s \langle state.more := \langle break = foo \rangle \rangle = s \langle break := foo \rangle$
 $\langle proof \rangle$

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* $\equiv$ *dflt-parametrization* *state.more*

(*RETURN* ($\langle break = False \rangle$) ($\langle$
on-back-edge := λ - - . *RETURN* ($\langle break = True \rangle$),
is-break := *break* $\rangle$)

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* $\langle proof \rangle$

We now can define our cyclicity checker. The partially correct version asserts a finitely branching graph:

definition *cyc-checker* *G* $\equiv$ *do* {
ASSERT (*fb-graph* *G*);
s $\leftarrow$ *cycc.it-dfs* *TYPE*(*'a*) *G*;
RETURN (*break* *s*)
}

The total correct variant asserts finitely many reachable nodes.

definition *cyc-checkerT* *G* $\equiv$ *do* {
ASSERT (*graph* *G* $\wedge$ *finite* (*graph-defs.reachable* *G*));
s $\leftarrow$ *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*
 $\langle proof \rangle$

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*
 $\langle proof \rangle$

lemma *cyccI'*:
 assumes *graph* *G*
 and *FR*: *finite* (*graph-defs.reachable* *G*)
 shows *cycc* *G*
 $\langle proof \rangle$

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*
 $\langle proof \rangle$

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* ($\lambda s. \text{break } s \longleftrightarrow \text{back-edges } s \neq \{\}$)
 $\langle proof \rangle$

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 ($\lambda s. \text{break } s \longleftrightarrow \text{back-edges } s \neq \{\}$)
 $\langle proof \rangle$

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

```

```

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

```

The same for the total correct variant:

```

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

```

```

theorem cyc-checkerT-correct: cyc-checkerT G  $\leq$  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* $\equiv \{$
 $(\langle \text{cycc-state-impl.break} = b \rangle, \langle \text{cycc-state.break} = b \rangle) \mid b. \text{True} \}$
abbreviation *cycc-rel* $\equiv \langle \text{cycc-erel} \rangle \text{simple-state-rel}$

Implementation of the parameters

definition *cycc-params-impl*
 $:: ('v, 'v \text{ cycc-state-impl}, \text{unit } \text{cycc-state-impl-ext}) \text{ gen-parameterization}$
where *cycc-params-impl*
 $\equiv \text{dflt-parametrization simple-state.more } (\text{RETURN } \langle \text{break} = \text{False} \rangle) \langle$
 $\text{on-back-edge} := \lambda u \ v \ s. \text{RETURN } \langle \text{break} = \text{True} \rangle,$
 $\text{is-break} := \text{break} \rangle$
lemmas *cycc-params-impl-simp*[*simp, DFS-code-unfold*] =
 $\text{gen-parameterization.simps}[mk\text{-record-simp, OF } \text{cycc-params-impl-def}[\text{simplified}]]$

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*: $(si, s) \in \text{cycc-rel}$
 $\implies \text{cycc-state-impl.break } si = \text{cycc-state.break } s$
 $\langle \text{proof} \rangle$

interpretation *cycc-impl*: *simple-impl-defs* *G* *cycc-params-impl* *cycc-params*
for *G* $\langle \text{proof} \rangle$

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

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

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

```

     $s \leftarrow \text{cycc-impl.tailrec-implT TYPE('a) } G;$ 
    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)  $G \leq \Downarrow$  cycc-rel it-dfs
  cycc-impl.rec-impl TYPE('more)  $G \leq \Downarrow$  cycc-rel it-dfs
  cycc-impl.tailrec-implT TYPE('more)  $G \leq \Downarrow$  cycc-rel it-dfsT
  <proof>

```

```

end

```

Finally, we get correctness of our cyclicity checker implementations

```

lemma cyc-checker-impl-refine: cyc-checker-impl  $G \leq \Downarrow Id$  (cyc-checker  $G$ )
  <proof>

```

```

lemma cyc-checker-rec-impl-refine:
  cyc-checker-rec-impl  $G \leq \Downarrow Id$  (cyc-checker  $G$ )
  <proof>

```

```

lemma cyc-checker-implT-refine: cyc-checker-implT  $G \leq \Downarrow Id$  (cyc-checkerT  $G$ )
  <proof>

```

2.1.4 Synthesizing Executable Code

Our algorithm's implementation is still abstract, as it uses abstract data structures like sets and relations. In a last step, we use the Autoref tool to derive an implementation with efficient data structures.

Again, we derive our state implementation from the template provided by the framework. The break-flag is implemented by a Boolean flag. Note that, in general, the user-defined state extensions may be data-refined in this step.

```

record ('si,'nsi,'psi)cycc-state-impl' = ('si,'nsi)simple-state-impl +
  break-impl :: bool

```

We define the refinement relation for the state extension

definition $[to-relAPP]$: $cycc-state-erel\ erel \equiv \{$
 $(\langle break-impl = bi, \dots = mi \rangle, \langle break = b, \dots = m \rangle) \mid bi\ mi\ b\ m.$
 $(bi, b) \in bool-rel \wedge (mi, m) \in erel\}$

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

shows

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

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

$\langle proof \rangle$

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

schematic-goal $cyc-checker-impl$:

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

assumes $[unfolded\ V-def, autoref-rules]$:

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

notes $[unfolded\ V-def, autoref-tyrel] =$

$TYRELI[where\ R = \langle V \rangle dflt-ahs-rel]$

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

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

$\langle proof \rangle$

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

assumes $4: cyc-checker-code\ Gi = dRETURN\ x$

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

$\langle proof \rangle$

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

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

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

assumes $[unfolded\ V-def, autoref-rules]$:

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

```

notes [unfolded V-def, autoref-tyrel] =
  TYRELI[where R= $\langle V \rangle$  dflt-ahs-rel]
  TYRELI[where R= $\langle V \times_r \langle V \rangle$  list-set-rel  $\rangle$  ras-rel]
shows nres-of ( $?c::?'c$  dres)  $\leq \Downarrow ?R$  (cyc-checker-rec-impl G)
 $\langle$ proof $\rangle$ 
concrete-definition cyc-checker-rec-code uses cyc-checker-rec-impl
prepare-code-thms cyc-checker-rec-code-def
export-code cyc-checker-rec-code checking SML

```

```

lemma cyc-checker-rec-code-correct:
  assumes 1: fb-graph G
  assumes 2:  $(Gi, G) \in \langle Rm, Id \rangle g\text{-impl-rel-ext}$ 
  assumes 4: cyc-checker-rec-code Gi = dRETURN x
  shows  $x \longleftrightarrow (\neg \text{acyclic } (g\text{-E } G \cap ((g\text{-E } G)^* \text{ `` } g\text{-V0 } G) \times UNIV))$ 
 $\langle$ proof $\rangle$ 

```

And, again, for the total correct version. Note that we generate a plain implementation, not inside a monad:

```

schematic-goal cyc-checker-implT:
  defines V  $\equiv Id :: ('v \times 'v::hashable)$  set
  assumes [unfolded V-def, autoref-rules]:
     $(Gi, G) \in \langle Rm, V \rangle g\text{-impl-rel-ext}$ 
  notes [unfolded V-def, autoref-tyrel] =
    TYRELI[where R= $\langle V \rangle$  dflt-ahs-rel]
    TYRELI[where R= $\langle V \times_r \langle V \rangle$  list-set-rel  $\rangle$  ras-rel]
  shows RETURN ( $?c::?'c$ )  $\leq \Downarrow ?R$  (cyc-checker-implT G)
 $\langle$ proof $\rangle$ 
concrete-definition cyc-checker-codeT uses cyc-checker-implT
export-code cyc-checker-codeT checking SML

```

```

theorem cyc-checker-codeT-correct:
  assumes 1: graph G finite (graph-defs.reachable G)
  assumes 2:  $(Gi, G) \in \langle Rm, Id \rangle g\text{-impl-rel-ext}$ 
  shows cyc-checker-codeT Gi  $\longleftrightarrow (\neg \text{acyclic } (g\text{-E } G \cap ((g\text{-E } G)^* \text{ `` } g\text{-V0 } G) \times UNIV))$ 
 $\langle$ 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*]: $s \langle \text{state.more} := \langle \text{ppath} = \text{foo} \rangle \rangle = s \langle \text{ppath} := \text{foo} \rangle$
 $\langle \text{proof} \rangle$

abbreviation *no-path* $\equiv \langle \text{ppath} = \text{None} \rangle$

abbreviation *a-path* *p v* $\equiv \langle \text{ppath} = \text{Some } (p,v) \rangle$

definition *fp0-params* :: (*'v* $\Rightarrow$ *bool*) $\Rightarrow$ *'v fp0-param*
where *fp0-params* *P* $\equiv \langle$
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* $\neq$ *None* $\rangle$

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* $\langle \text{proof} \rangle$

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

lemma [*simp*]:
ppath (empty-state (ppath = e)) = *e*
 $\langle \text{proof} \rangle$

```

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*: $it_dfs \leq SPEC\ (\lambda s. \text{case } ppath\ s \text{ of}$
 $None \Rightarrow \neg(\exists v0 \in V0. \exists v. (v0, v) \in E^* \wedge P\ v)$
 $| Some\ (p, v) \Rightarrow (\exists v0 \in V0. path\ E\ v0\ p\ v \wedge P\ v \wedge distinct\ (p@[v])))$
 $\langle proof \rangle$

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* $G\ P \equiv \lambda r. \text{case } r \text{ of}$

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

definition *find-path0-spec*

$:: ('v, -)\ graph_rec_scheme \Rightarrow ('v \Rightarrow bool) \Rightarrow 'v\ fp_result\ nres$

— Searches a path from the root nodes to some target node that satisfies a given predicate. If such a path is found, the path and the target node are returned

where

$find_path0_spec\ G\ P \equiv do\ \{$
 $ASSERT\ (fb_graph\ G);$
 $SPEC\ (find_path0_pred\ G\ P)$
 $\}$

definition *find-path0*

```

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

```

lemma *find-path0-correct*:
shows *find-path0 G P $\leq$ find-path0-spec G P*
<proof>

lemmas *find-path0-spec-rule[refine-vcg]* =
 ASSERT-le-defI[OF *find-path0-spec-def*]
 ASSERT-leof-defI[OF *find-path0-spec-def*]

2.2.2 Restricting the Graph

Extended interface, propagating set of already searched nodes (restriction)

definition *restr-invar*

— Invariant for a node restriction, i.e., a transition closed set of nodes known to not contain a target node that satisfies a predicate.

where

restr-invar E R P $\equiv$ E “ $R \subseteq R \wedge R \cap \text{Collect } P = \{\}$

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

lemma *restr-invar-imp-not-reachable*: *restr-invar E R P $\Longrightarrow$ E* “ $R \cap \text{Collect } P = \{\}$*
<proof>

type-synonym 'v fpr-result = 'v set + ('v list $\times$ 'v)

definition *find-path0-restr-pred G P R $\equiv$ λr .*

case r of

Inl R' $\Rightarrow$ R' = R $\cup$ (g-E G) “ g-V0 G $\wedge$ restr-invar (g-E G) R' P*

| Inr (vs,v) $\Rightarrow$ P v $\wedge$ ($\exists$ v0 $\in$ g-V0 G - R. path (rel-restrict (g-E G) R) v0 vs
v)

definition *find-path0-restr-spec*

— Find a path to a target node that satisfies a predicate, not considering nodes from the given node restriction. If no path is found, an extended restriction is returned, that contains the start nodes

where *find-path0-restr-spec G P R $\equiv$ do {*

ASSERT (fb-graph G $\wedge$ restr-invar (g-E G) R P);

SPEC (find-path0-restr-pred G P R)}

lemmas *find-path0-restr-spec-rule[refine-vcg]* =
 ASSERT-le-defI[OF *find-path0-restr-spec-def*]
 ASSERT-leof-defI[OF *find-path0-restr-spec-def*]

definition *find-path0-restr*

```

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

```

lemma *find-path0-restr-correct*:

shows *find-path0-restr* G P R $\leq$ *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)+ “ g-V0 G  $\wedge$  restr-invar (g-E G) R' P
  | Inr (vs,v)  $\Rightarrow$  P v  $\wedge$  vs  $\neq$  []  $\wedge$  ( $\exists$  v0  $\in$  g-V0 G. 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 $\equiv$ do {
 ASSERT (fb-graph G $\wedge$ restr-invar (g-E G) R P);
 SPEC (*find-path1-restr-pred* G P R)}

lemmas *find-path1-restr-spec-rule*[refine-vcg] =

ASSERT-le-defI[OF *find-path1-restr-spec-def*]
 ASSERT-leof-defI[OF *find-path1-restr-spec-def*]

definition *find-path1-restr*

```

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

```

$$\begin{aligned} & \text{Inl } - \Rightarrow \text{RETURN } f0 \\ & | \text{Inr } (vs, v) \Rightarrow \text{RETURN } (\text{Inr } (v0 \# vs, v)) \\ & \}) (\text{Inl } R) \end{aligned}$$

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

$$\begin{aligned} & \text{case } s \text{ of} \\ & \quad \text{Inl } R \Rightarrow R = R0 \cup (g\text{-}E \ G)^+ \text{ `` } (g\text{-}V0 \ G - \text{it}) \wedge \text{restr-invar } (g\text{-}E \ G) \ R \ P \\ & \quad | \text{Inr } (vs, v) \Rightarrow P \ v \wedge vs \neq [] \wedge (\exists \ v0 \in g\text{-}V0 \ G - \text{it. } \text{path } (g\text{-}E \ G \cap \text{UNIV} \times \\ & \quad - R0) \ v0 \ vs \ v) \end{aligned}$$

lemma *find-path1-restr-correct*:

shows *find-path1-restr* $G P R \leq \text{find-path1-restr-spec } G P R$
 $\langle \text{proof} \rangle$

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

$$\begin{aligned} & \text{case } r \text{ of} \\ & \quad \text{None} \Rightarrow (g\text{-}E \ G)^+ \text{ `` } g\text{-}V0 \ G \cap \text{Collect } P = \{\} \\ & \quad | \text{Some } (vs, v) \Rightarrow P \ v \wedge vs \neq [] \wedge (\exists \ v0 \in g\text{-}V0 \ G. \text{path } (g\text{-}E \ G) \ v0 \ vs \ v) \end{aligned}$$

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* $G P \equiv \text{do } \{$
 $\text{ASSERT } (\text{fb-graph } G);$
 $\text{SPEC } (\text{find-path1-pred } G P) \}$

lemmas *find-path1-spec-rule*[*refine-vcg*] =

$\text{ASSERT-le-defI}[OF \ \text{find-path1-spec-def}]$
 $\text{ASSERT-leof-defI}[OF \ \text{find-path1-spec-def}]$

2.2.4 Path of Minimal Length One, without Restriction

definition *find-path1*

$$\begin{aligned} & :: ('v, 'more) \text{ graph-rec-scheme} \Rightarrow ('v \Rightarrow \text{bool}) \Rightarrow 'v \text{ fp-result } nres \\ & \text{where } \text{find-path1 } G P \equiv \text{do } \{ \\ & \quad r \leftarrow \text{find-path1-restr-spec } G P \ \{\}; \\ & \quad \text{case } r \text{ of} \\ & \quad \quad \text{Inl } - \Rightarrow \text{RETURN } \text{None} \\ & \quad | \text{Inr } vsv \Rightarrow \text{RETURN } (\text{Some } vsv) \\ & \} \end{aligned}$$

lemma *find-path1-correct*:

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

2.2.5 Implementation

record *'v fp0-state-impl* = *'v simple-state* +
 $\text{ppath} :: ('v \text{ list} \times 'v) \text{ option}$

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

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

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

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

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

lemma $fp0\text{-}ss\text{-}rel\text{-}ppath\text{-}cong[simp]$:
 $(s,s') \in \langle fp0\text{-}erel \rangle simple\text{-}state\text{-}rel \implies fp0\text{-}state\text{-}impl.ppath \ s = fp0\text{-}state.ppath \ s'$
 $\langle proof \rangle$

lemma $fp0i\text{-}cong[cong]$: $simple\text{-}state.more \ s = simple\text{-}state.more \ s'$
 $\implies fp0\text{-}state\text{-}impl.ppath \ s = fp0\text{-}state\text{-}impl.ppath \ s'$
 $\langle proof \rangle$

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

definition $fp0\text{-}params\text{-}impl$
 $:: - \Rightarrow ('v, 'v \text{ fp0}\text{-}state\text{-}impl, ('v, unit) \text{ fp0}\text{-}state\text{-}impl\text{-}ext) \text{ gen}\text{-}parameterization$
where $fp0\text{-}params\text{-}impl \ P \equiv \langle$
 $on\text{-}init = RETURN \ no\text{-}path\text{-}impl,$
 $on\text{-}new\text{-}root = \lambda v0 \ s.$
 $\text{ if } P \ v0 \text{ then } RETURN \ (a\text{-}path\text{-}impl \ [] \ v0) \text{ else } RETURN \ no\text{-}path\text{-}impl,$
 $on\text{-}discover = \lambda u \ v \ s.$
 $\text{ if } P \ v \text{ then } RETURN \ (a\text{-}path\text{-}impl \ (map \ fst \ (rev \ (tl \ (CAST \ (ss\text{-}stack \ s))))) \ v)$
 $\text{ else } RETURN \ no\text{-}path\text{-}impl,$
 $on\text{-}finish = \lambda u \ s. RETURN \ (simple\text{-}state.more \ s),$
 $on\text{-}back\text{-}edge = \lambda u \ v \ s. RETURN \ (simple\text{-}state.more \ s),$
 $on\text{-}cross\text{-}edge = \lambda u \ v \ s. RETURN \ (simple\text{-}state.more \ s),$
 $is\text{-}break = \lambda s. ppath \ s \neq None \ \rangle$

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

interpretation $fp0\text{-}impl$:
 $restricted\text{-}impl\text{-}defs \ fp0\text{-}params\text{-}impl \ P \ fp0\text{-}params \ P \ G \ R$
for $G \ P \ R \ \langle proof \rangle$

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

begin

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

sublocale *impl*: *restricted-impl* *G* *fp0-params* *P* *fp0-params-impl* *P*
fp0-erel *R*
 $\langle \text{proof} \rangle$
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*
 $\leq \Downarrow (\langle \text{Id}, \text{Id} \times_r \text{Id} \rangle \text{sum-rel})$
(*find-path0-restr* *G* *P* *R*)
 $\langle \text{proof} \rangle$

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*
 $\leq \Downarrow (\langle \text{Id} \times_r \text{Id} \rangle \text{option-rel})$ (*find-path0* *G* *P*)
 $\langle \text{proof} \rangle$

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* $\equiv$ {
($\langle \text{ppath-impl} = p_i, \dots = m_i \rangle, \langle \text{ppath} = p, \dots = m \rangle$) | $p_i \ m_i \ p \ m.$
 $(p_i, p) \in \langle \text{Id} \rangle \text{list-rel} \times_r \text{Id} \rangle \text{option-rel} \wedge (m_i, m) \in \text{erel}$ }

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

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

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

schematic-goal find-path0-restr-code:
fixes  $vis\text{-}rel :: ('v \times 'v) set \Rightarrow ('visi \times 'v) set \Rightarrow set$ 
notes [autoref-rel-intf] = REL-INTFI[of vis-rel i-set for I]
assumes [autoref-rules]: (op-vis-insert, insert)  $\in Id \rightarrow \langle Id \rangle vis\text{-}rel \rightarrow \langle Id \rangle vis\text{-}rel$ 
assumes [autoref-rules]: (op-vis-memb, (∈))  $\in Id \rightarrow \langle Id \rangle vis\text{-}rel \rightarrow bool\text{-}rel$ 
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-of (?c::?'c dres),
  find-path0-restr-impl
    G
    P
    ( $R :: {}_r \langle Id \rangle vis\text{-}rel$ ))  $\in ?R$ 
  <proof>

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  $R = \langle (Id :: ('a \times 'a :: hashable) set) \rangle dflt-ahs-rel$ ]
shows ( $nres-of \ ?c, find-path1 \ G \ P$ )
 $\in \langle \langle \langle V \rangle list-rel \times_r V \rangle option-rel \rangle nres-rel$ 
 $\langle proof \rangle$ 
concrete-definition find-path1-code uses find-path1-code

export-code find-path1-code checking SML

lemma find-path1-code-autoref-aux:
  assumes  $Vid: V = (Id :: 'a :: hashable \ rel)$ 
  shows ( $\lambda \ G \ P. nres-of (find-path1-code \ G \ P), find-path1-spec$ )
 $\in \langle Rm, V \rangle g-impl-rel-ext \rightarrow (V \rightarrow bool-rel) \rightarrow \langle \langle \langle V \rangle list-rel \times_r V \rangle option-rel \rangle nres-rel$ 
 $\langle proof \rangle$ 

lemmas find-path1-autoref[autoref-rules] = find-path1-code-autoref-aux[OF 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* $G \equiv do \{$
 $ASSERT \ (fb-graph \ G);$
 $s \leftarrow unit-dfs.it-dfsT \ TYPE('a) \ G;$
 $RETURN \ (dom \ (discovered \ s))$
 $\}$

2.3.3 Correctness

context *unit-DFS* **begin**

lemma *find-reachable-correct*: $find-reachable \ G \leq SPEC \ (\lambda r. \ r = reachable)$
 $\langle proof \rangle$

lemma *find-reachableT-correct*:
 $finite \ reachable \implies find-reachableT \ G \leq SPEC \ (\lambda r. \ r = reachable)$
 $\langle proof \rangle$

end

context *unit-DFS* **begin**

sublocale *simple-impl* $G \ unit-parametrization \ unit-parametrization \ unit-rel$
 $\langle proof \rangle$

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

interpretation *unit-simple-impl*:

simple-impl-defs $G \ unit-parametrization \ unit-parametrization$
for $G \ \langle proof \rangle$

term *unit-simple-impl.tailrec-impl* **term** *unit-simple-impl.rec-impl*

definition [*DFS-code-unfold*]: *find-reachable-impl* $G \equiv do \{$
 $ASSERT \ (fb-graph \ G);$
 $s \leftarrow unit-simple-impl.tailrec-impl \ TYPE('a) \ G;$
 $RETURN \ (simple-state.visited \ s)$
 $\}$

definition [*DFS-code-unfold*]: *find-reachable-implT* $G \equiv do \{$
 $ASSERT \ (fb-graph \ G);$
 $s \leftarrow unit-simple-impl.tailrec-implT \ TYPE('a) \ G;$
 $RETURN \ (simple-state.visited \ s)$
 $\}$

definition [*DFS-code-unfold*]: *find-reachable-rec-impl* $G \equiv do \{$
 $ASSERT \ (fb-graph \ G);$
 $s \leftarrow unit-simple-impl.rec-impl \ TYPE('a) \ G;$
 $RETURN \ (visited \ s)$
 $\}$

}

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 $V \equiv Id :: ('v \times 'v :: \text{hashable}) \text{ set}$
 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** $R = \langle V \rangle \text{dflt-ahs-rel}$]
 TYRELI[**where** $R = \langle V \times_r \langle V \rangle \text{list-set-rel} \rangle \text{ras-rel}$]
 shows $nres\text{-of } (?c :: ?'c \text{ dres}) \leq \Downarrow ?R$ (*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 $(r, (g\text{-}E\ G)^* \text{ “ } g\text{-}V0\ G) \in \langle Id \rangle \text{dflt-ahs-rel}$
 ⟨*proof*⟩

schematic-goal *find-reachable-implT*:
 fixes $V :: ('vi \times 'v) \text{ set}$
 assumes [*autoref-ga-rules*]: *is-bounded-hashcode* $V \text{ eq } bhc$
 assumes [*autoref-rules*]: $(eq, (=)) \in V \rightarrow V \rightarrow \text{bool-rel}$
 assumes [*autoref-ga-rules*]: *is-valid-def-hm-size* *TYPE* $('vi) \text{ sz}$
 assumes [*autoref-rules*]:
 (Gi, G) $\in \langle Rm, V \rangle g\text{-impl-rel-ext}$
 notes [*autoref-tyrel*] =
 TYRELI[**where** $R = \langle V \rangle \text{ahs-rel } bhc$]
 TYRELI[**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-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 × 'v) *set*
assumes *G*: *graph G*
assumes *FR*: *finite ((g-E G)* “ g-V0 G)*
assumes *BHC*: *is-bounded-hashcode V eq bhc*
assumes *EQ*: *(eq,(=)) ∈ V → V → bool-rel*
assumes *VDS*: *is-valid-def-hm-size TYPE ('vi) sz*
assumes *2*: *(Gi, G) ∈ ⟨Rm, V⟩g-impl-rel-ext*
shows *(find-reachable-codeT eq bhc sz Gi, (g-E G)* “ g-V0 G) ∈ (V)ahs-rel bhc*
⟨proof⟩

definition *all-unit-rel* :: *(unit × 'a) set* **where** *all-unit-rel* ≡ *UNIV*

lemma *all-unit-refine[simp]*:
(((),x) ∈ all-unit-rel ⟨proof⟩)

definition *unit-list-rel* :: *('c × 'a) set ⇒ (unit × 'a list) set*
where *[to-relAPP]*: *unit-list-rel R* ≡ *UNIV*

lemma *unit-list-rel-refine[simp]*: *(((),y) ∈ ⟨R⟩unit-list-rel*
⟨proof⟩

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

lemma *[autoref-rules]*:
(((),[]) ∈ ⟨R⟩unit-list-rel
(λ-. (()),tl) ∈ ⟨R⟩unit-list-rel → ⟨R⟩unit-list-rel
(λ-. (()),(#)) ∈ R → ⟨R⟩unit-list-rel → ⟨R⟩unit-list-rel
⟨proof⟩

schematic-goal *find-reachable-rec-impl*:
defines *V* ≡ *Id* :: *('v × 'v::hashable) set*
assumes *[unfolded V-def,autoref-rules]*:
(Gi, G) ∈ ⟨Rm, V⟩g-impl-rel-ext
notes *[unfolded V-def,autoref-tyrel]* =
TYRELI[where R=⟨V⟩dflt-ahs-rel]
shows *nres-of (?c::?'c dres) ≤_??R (find-reachable-rec-impl G)*
⟨proof⟩
concrete-definition *find-reachable-rec-code* **uses** *find-reachable-rec-impl*
prepare-code-thms *find-reachable-rec-code-def*
export-code *find-reachable-rec-code* **checking** *SML*

lemma *find-reachable-rec-code-correct*:
assumes 1: *fb-graph* G
assumes 2: $(Gi, G) \in \langle Rm, Id \rangle g\text{-impl-rel-ext}$
assumes 4: *find-reachable-rec-code* $Gi = dRETURN\ r$
shows $(r, (g\text{-}E\ G)^* \text{ `` } g\text{-}V0\ G) \in \langle Id \rangle dflt\text{-}ahs\text{-}rel$
 $\langle proof \rangle$

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

context begin interpretation *autoref-syn* $\langle proof \rangle$

lemma *autoref-op-reachable*[*autoref-rules*]:
fixes $V :: ('vi \times 'v)\ set$
assumes G : *SIDE-PRECOND* (*graph* G)
assumes FR : *SIDE-PRECOND* (*finite* $((g\text{-}E\ G)^* \text{ `` } g\text{-}V0\ G)$)
assumes BHC : *SIDE-GEN-ALGO* (*is-bounded-hashcode* $V\ eq\ bhc$)
assumes EQ : *GEN-OP* $eq\ (=)\ (V \rightarrow V \rightarrow bool\text{-}rel)$
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 (*find-reachable-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**
 $is\text{-}fas\ G\ EC \equiv \neg(\exists\ u \in (g\text{-}E\ G)^* \text{ `` } g\text{-}V0\ G. (u, u) \in (g\text{-}E\ G - EC)^+)$

lemma *is-fas-alt*:

is-fas G $EC = \text{acyclic } ((g-E \ G) \cap ((g-E \ G)^* \text{ `` } g-V0 \ G \times \text{ UNIV}) - EC))$
 $\langle \text{proof} \rangle$

2.4.1 Instantiation of the DFS-Framework

record $'v$ *fas-state* = $'v$ *state* +
fas :: $('v \times 'v)$ *set*

lemma *fas-more-cong*: $\text{state.more } s = \text{state.more } s' \implies \text{fas } s = \text{fas } s'$
 $\langle \text{proof} \rangle$

lemma [*simp*]: $s \langle \text{state.more} := \langle \text{fas} = \text{foo} \rangle \rangle = s \langle \text{fas} := \text{foo} \rangle$
 $\langle \text{proof} \rangle$

definition *fas-params* :: $('v, ('v, \text{unit}) \text{ fas-state-ext})$ *parameterization*
where *fas-params* $\equiv \text{dflt-parameterization state.more}$
 $(\text{RETURN } \langle \text{fas} = \{\} \rangle) \langle$
 $\text{on-back-edge} := \lambda u \ v \ s. \text{RETURN } \langle \text{fas} = \text{insert } (u, v) (\text{fas } s) \rangle$
 $\rangle$

lemmas *fas-params-simp*[*simp*] =
 $\text{gen-parameterization.simps}[\text{mk-record-simp}, \text{OF } \text{fas-params-def}[\text{simplified}]]$

interpretation *fas*: *param-DFS-defs* **where** $\text{param} = \text{fas-params}$ **for** G $\langle \text{proof} \rangle$

Find feedback arc set

definition *find-fas* $G \equiv \text{do } \{$
 $\text{ASSERT } (\text{graph } G);$
 $\text{ASSERT } (\text{finite } ((g-E \ G)^* \text{ `` } g-V0 \ G));$
 $s \leftarrow \text{fas.it-dfsT TYPE('a)} \ G;$
 $\text{RETURN } (\text{fas-state.fas } s)$
 $\}$

locale *fas* =
 $\text{param-DFS } G \text{ fas-params}$
for $G :: ('v, 'more)$ *graph-rec-scheme*
 $+$
assumes *finite-reachable*[*simp*, *intro!*]: $\text{finite } ((g-E \ G)^* \text{ `` } g-V0 \ G)$
begin

sublocale *DFS* $G \text{ fas-params}$
 $\langle \text{proof} \rangle$

end

lemma *fasI*:
assumes *graph* G
assumes *finite* $((g-E \ G)^* \text{ `` } g-V0 \ G)$
shows *fas* G

$\langle proof \rangle$

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* ($\lambda s. \text{fas-state.fas } s = \text{back-edges } s$)
 $\langle proof \rangle$

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*)
 $\langle proof \rangle$

end

lemma *find-fas-correct*:
assumes *graph* *G*
assumes *finite* $((g-E \ G)^* \text{ `` } g-V0 \ G)$
shows *find-fas* *G* $\leq \text{SPEC } (\text{is-fas } G)$
 $\langle proof \rangle$

2.4.3 Implementation

record *'v fas-state-impl* = *'v simple-state* +
fas :: (*'v* $\times$ *'v*) *set*

definition *fas-rel* $\equiv \{$
 $(\langle \text{fas-state-impl.fas} = f \rangle, \langle \text{fas-state.fas} = f \rangle) \mid f. \text{True} \}$
abbreviation *fas-rel* $\equiv \langle \text{fas-rel} \rangle \text{simple-state-rel}$

definition *fas-params-impl*
 $:: ('v, 'v \text{ fas-state-impl}, ('v, \text{unit}) \text{ fas-state-impl-ext}) \text{ gen-parameterization}$
where *fas-params-impl*
 $\equiv \text{dflt-parameterization simple-state.more } (\text{RETURN } \langle \text{fas} = \{\} \rangle) \langle$
 $\text{on-back-edge} := \lambda u \ v \ s. \text{RETURN } \langle \text{fas} = \text{insert } (u, v) (\text{fas } s) \rangle \rangle$
lemmas *fas-params-impl-simp* [*simp*, *DFS-code-unfold*] =
 $\text{gen-parameterization.simps}[\text{mk-record-simp}, \text{OF } \text{fas-params-impl-def}[\text{simplified}]]$

lemma *fas-impl*: $(si, s) \in \text{fas-rel}$
 $\implies \text{fas-state-impl.fas } si = \text{fas-state.fas } s$
 $\langle proof \rangle$

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

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* $G \text{ fas-params fas-params-impl fas-erel}$
 $\langle \text{proof} \rangle$

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

lemma *find-fas-impl-refine*: *find-fas-impl* $G \leq \Downarrow Id \ (\text{find-fas } G)$
 $\langle \text{proof} \rangle$

2.4.4 Synthesis of Executable Code

record $(\text{'si}, \text{'nsi}, \text{'fsi})\text{fas-state-impl}' = (\text{'si}, \text{'nsi})\text{simple-state-impl} +$
 $\text{fas-impl} :: \text{'fsi}$

definition [*to-relAPP*]: *fas-state-erel* *frel* *erel* $\equiv \{$
 $(\langle \text{fas-impl} = f_i, \dots = m_i \rangle, \langle \text{fas} = f, \dots = m \rangle) \mid f_i \ m_i \ f \ m.$
 $(f_i, f) \in \text{frel} \wedge (m_i, m) \in \text{erel} \}$

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

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  $V :: ('vi \times 'v) \text{ set}$ 
assumes [autoref-ga-rules]: is-bounded-hashcode  $V \text{ eq } bhc$ 
assumes [autoref-rules]:  $(eq, (=)) \in V \rightarrow V \rightarrow \text{bool-rel}$ 
assumes [autoref-ga-rules]: is-valid-def-hm-size  $TYPE ('vi) \text{ sz}$ 
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  $R = \langle V \times_r V \rangle ahs\text{-rel } (prod\text{-bhc } bhc \text{ } bhc)$ ]
  TYRELI[where  $R = \langle V \times_r \langle V \rangle list\text{-set-rel} \rangle ras\text{-rel}$ ]
shows RETURN  $(?c :: ?'c) \leq \Downarrow ?R (find\text{-fas-impl } G)$ 
 $\langle proof \rangle$ 
concrete-definition find-fas-code for  $eq \text{ } bhc \text{ } sz \text{ } Gi$  uses find-fas-impl
export-code find-fas-code checking SML

```

thm *find-fas-code.refine*

lemma *find-fas-code-refine*[*refine*]:

```

fixes  $V :: ('vi \times 'v) \text{ set}$ 
assumes is-bounded-hashcode  $V \text{ eq } bhc$ 
assumes  $(eq, (=)) \in V \rightarrow V \rightarrow \text{bool-rel}$ 
assumes is-valid-def-hm-size  $TYPE ('vi) \text{ sz}$ 
assumes  $2: (Gi, G) \in \langle Rm, V \rangle g\text{-impl-rel-ext}$ 
shows RETURN  $(find\text{-fas-code } eq \text{ } bhc \text{ } sz \text{ } Gi) \leq \Downarrow (\langle V \times_r V \rangle ahs\text{-rel } (prod\text{-bhc } bhc \text{ } bhc)) (find\text{-fas } G)$ 
 $\langle proof \rangle$ 

```

context **begin interpretation** *autoref-syn* $\langle proof \rangle$

Declare this algorithm to Autoref:

theorem *find-fas-code-autoref*[*autoref-rules*]:

```

fixes  $V :: ('vi \times 'v) \text{ set}$  and  $bhc$ 
defines  $RR \equiv \langle \langle V \times_r V \rangle ahs\text{-rel } (prod\text{-bhc } bhc \text{ } bhc) \rangle nres\text{-rel}$ 
assumes BHC: SIDE-GEN-ALGO (is-bounded-hashcode  $V \text{ eq } bhc$ )
assumes EQ: GEN-OP  $eq (=) (V \rightarrow V \rightarrow \text{bool-rel})$ 
assumes VDS: SIDE-GEN-ALGO (is-valid-def-hm-size  $TYPE ('vi) \text{ sz}$ )
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)* “ g-V0 G));
  let nodes = (g-E G)* “ g-V0 G;
  fas  $\leftarrow$  find-fas (| g-V = g-V G, g-E = g-E G - FI, g-V0 = nodes |);
  RETURN (FI  $\cup$  fas)
}

```

The abstract idea: To find a feedback arc set that contains some set F2, we can find a feedback arc set for the graph with F2 removed, and then join with F2.

lemma *is-fas-join*: $is-fas\ G\ (F1 \cup F2) \longleftrightarrow$

```

is-fas (| g-V = g-V G, g-E = g-E G - F2, g-V0 = (g-E G)* “ g-V0 G |) F1
<proof>

```

lemma *graphI-init*:

```

assumes graph G
shows graph (| g-V = g-V G, g-E = g-E G - FI, g-V0 = (g-E G)* “ g-V0 G |)
<proof>

```

lemma *find-fas-init-correct*:

```

assumes [simp, intro!]: graph G
assumes [simp, intro!]: finite ((g-E G)* “ g-V0 G)
shows find-fas-init G FI  $\leq$  SPEC ( $\lambda fas.$  is-fas G fas  $\wedge$  FI  $\subseteq$  fas)
<proof>

```

lemma *gen-cast-set[autoref-rules-raw]*:

```

assumes PRIO-TAG-GEN-ALGO
assumes INS: GEN-OP ins Set.insert (Rk  $\rightarrow$  (Rk)Rs2  $\rightarrow$  (Rk)Rs2)
assumes EM: GEN-OP emp {} ((Rk)Rs2)
assumes IT: SIDE-GEN-ALGO (is-set-to-list Rk Rs1 tsl)
shows ( $\lambda s.$  gen-union ( $\lambda x.$  foldli (tsl x)) ins s emp, CAST)
   $\in$  ((Rk)Rs1)  $\rightarrow$  ((Rk)Rs2)
<proof>

```

lemma *gen-cast-fun-set-rel[autoref-rules-raw]*:

```

assumes INS: GEN-OP mem ( $\in$ ) (Rk  $\rightarrow$  (Rk)Rs  $\rightarrow$  bool-rel)
shows ( $\lambda s\ x.$  mem x s, CAST)  $\in$  ((Rk)Rs)  $\rightarrow$  ((Rk)fun-set-rel)
<proof>

```

lemma *find-fas-init-impl-aux-unfolds*:

```

Let (E* “V0) = Let (CAST (E* “V0))

```

$(\lambda S. \text{RETURN } (FI \cup S)) = (\lambda S. \text{RETURN } (FI \cup \text{CAST } S))$
 $\langle \text{proof} \rangle$

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

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* $\langle \text{proof} \rangle$

The following theorem declares our implementation to Autoref:

theorem *find-fas-init-code-autoref*[*autoref-rules*]:
fixes $V :: ('vi \times 'v) \text{ set}$ **and** *bhc*
defines $RR \equiv \langle V \times_r V \rangle fun\text{-set-rel}$
assumes *SIDE-GEN-ALGO* (*is-bounded-hashcode* $V \text{ eq } bhc$)
assumes *GEN-OP* *eq* $(=) (V \rightarrow V \rightarrow \text{bool-rel})$
assumes *SIDE-GEN-ALGO* (*is-valid-def-hm-size* $TYPE ('vi) \text{ sz}$)
shows $(\lambda Gi \ FIi. \text{RETURN } (\text{find-fas-init-code } eq \ bhc \ sz \ Gi \ FIi), \text{find-fas-init})$
 $\in \langle Rm, V \rangle g\text{-impl-rel-ext} \rightarrow RR \rightarrow \langle RR \rangle nres\text{-rel}$
 $\langle \text{proof} \rangle$
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:  $E \models F \subseteq F \cup S$ 
  assumes NR:  $E^* \models U \cap S = \{\}$ 
  assumes SS:  $U \subseteq F$ 
  shows  $E^* \models U \subseteq F$ 
  — 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  $\times$  'v list) option
  red :: 'v set

type-synonym 'v blue-dfs-param = ('v, ('v,unit) blue-dfs-state-ext) parameteriza-
tion

lemma lasso-more-cong[cong]: state.more s = state.more s'  $\implies$  lasso s = lasso s'
  <proof>
lemma red-more-cong[cong]: state.more s = state.more s'  $\implies$  red s = red s'
  <proof>

lemma [simp]: s ( $\lfloor$  state.more := ( $\lfloor$  lasso = foo, red = bar  $\rfloor$ )  $\rfloor$ ) = s ( $\lfloor$  lasso := foo,
red := bar  $\rfloor$ )
  <proof>

```

```

abbreviation dropWhileNot v  $\equiv$  dropWhile (( $\neq$ ) v)
abbreviation takeWhileNot v  $\equiv$  takeWhile (( $\neq$ ) v)

```

```

locale BlueDFS-defs = graph-defs G
  for G :: ('v, 'more) graph-rec-scheme +
  fixes accpt :: 'v  $\Rightarrow$  bool
begin

```

```

definition blue s  $\equiv$  dom (finished s) - red s
definition cyan s  $\equiv$  set (stack s)
definition white s  $\equiv$   $-$  dom (discovered s)

```

abbreviation $red\text{-}dfs\ R\ ss\ x \equiv find\text{-}path1\text{-}restr\text{-}spec\ (G\ \langle\ g\text{-}V0 := \{x\}\ \rangle)\ ss\ R$

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

$mk\text{-}blue\text{-}witness\ s\ redS \equiv case\ redS\ of$
 $\quad Inl\ R' \Rightarrow \langle\ lasso = None, red = (R' \setminus \{\beta\})\ \rangle$
 $\quad | Inr\ (vs, v) \Rightarrow let\ rs = rev\ (stack\ s)\ in$
 $\quad \quad \langle\ lasso = Some\ (rs, vs@dropWhileNot\ v\ rs), red = red\ s\rangle$

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

$if\ accpt\ u\ then$

$\quad let\ rs = rev\ (tl\ (stack\ s));$

$\quad ur = rs;$

$\quad ul = u\#dropWhileNot\ v\ rs$

$\quad in\ RETURN\ \langle\ lasso = Some\ (ur, ul), red = red\ s\rangle$

$else\ if\ accpt\ v\ then$

$\quad let\ rs = rev\ (stack\ s);$

$\quad vr = takeWhileNot\ v\ rs;$

$\quad vl = dropWhileNot\ v\ rs$

$\quad in\ RETURN\ \langle\ lasso = Some\ (vr, vl), red = red\ s\rangle$

$else\ NOOP\ s$

$| - \Rightarrow NOOP\ s$

definition $blue\text{-}dfs\text{-}params :: 'v\ blue\text{-}dfs\text{-}param$

where $blue\text{-}dfs\text{-}params = \langle\$

$\quad on\text{-}init = RETURN\ \langle\ lasso = None, red = \{\}\ \rangle,$

$\quad on\text{-}new\text{-}root = \lambda v0\ s. NOOP\ s,$

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

$\quad on\text{-}finish = \lambda u\ s. if\ accpt\ u\ then\ run\text{-}red\text{-}dfs\ u\ s\ else\ NOOP\ s,$

$\quad on\text{-}back\text{-}edge = se\text{-}back\text{-}edge,$

$\quad on\text{-}cross\text{-}edge = \lambda u\ v\ s. NOOP\ s,$

$\quad is\text{-}break = \lambda s. lasso\ s \neq None\ \rangle$

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  $\Rightarrow$  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  $\longleftrightarrow$  BlueDFS-invar G accept s
    <proof>

end

```

2.5.3 Correctness Proof

context BlueDFS **begin**

definition blue-basic-invar s $\equiv$
 case lasso s of
 None $\Rightarrow$ restr-invar E (red s) ($\lambda x. x \in \text{set } (\text{stack } s)$)
 $\wedge$ red s $\subseteq$ dom (finished s)
 | Some l $\Rightarrow$ True

lemma (in BlueDFS-invar) red-DFS-precond-aux:
assumes BI: blue-basic-invar s
assumes [simp]: lasso s = None
assumes SNE: stack s $\neq$ []
shows

$fb\text{-}graph\ (G\ \|\ g\text{-}V0 := \{hd\ (stack\ s)\}\ \|\)$
and $fb\text{-}graph\ (G\ \|\ g\text{-}E := E \cap UNIV \times -\ red\ s,\ g\text{-}V0 := \{hd\ (stack\ s)\}\ \|\)$
and $restr\text{-}invar\ E\ (red\ s)\ (\lambda x.\ x \in set\ (stack\ s))$
 $\langle proof \rangle$

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

lemma *blue-basic-invar*: *is-invar* *blue-basic-invar*
 $\langle proof \rangle$

lemmas (in *BlueDFS-invar*) *s-blue-basic-invar*
 $= blue\text{-}basic\text{-}invar[THEN\ make\text{-}invar\text{-}thm]$

lemmas (in *BlueDFS-invar*) *red-DFS-precond*
 $= red\text{-}DFS\text{-}precond\text{-}aux[OF\ s\text{-}blue\text{-}basic\text{-}invar]$

sublocale *DFS* *G* *blue-dfs-params*
 $\langle proof \rangle$

end

context *BlueDFS-invar*
begin

context **assumes** [*simp*]: *lasso* *s* = *None*
begin

lemma *red-closed*:
 $E\ \text{“}\ red\ s \subseteq red\ s$
 $\langle proof \rangle$

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

lemma *red-finished*: $red\ s \subseteq dom\ (finished\ s)$
 $\langle proof \rangle$

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

lemma *colors-disjoint*:

$white\ s \cap (blue\ s \cup cyan\ s \cup red\ s) = \{\}$
 $blue\ s \cap (white\ s \cup cyan\ s \cup red\ s) = \{\}$
 $cyan\ s \cap (white\ s \cup blue\ s \cup red\ s) = \{\}$
 $red\ s \cap (white\ s \cup blue\ s \cup cyan\ s) = \{\}$
 $\langle proof \rangle$

end

lemma (**in** *BlueDFS*) *i-no-accept-cycle-in-finish*:

$is_invar\ (\lambda s. \text{lasso}\ s = None \longrightarrow (\forall x. \text{accept}\ x \wedge x \in \text{dom}\ (\text{finished}\ s) \longrightarrow (x, x) \notin E^+))$
 $\langle proof \rangle$

lemma *no-accept-cycle-in-finish*:

$\llbracket \text{lasso}\ s = None; \text{accept}\ v; v \in \text{dom}\ (\text{finished}\ s) \rrbracket \implies (v, v) \notin E^+$
 $\langle proof \rangle$

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*: $\text{lasso-inv}\ s$
and *ne*: $\text{stack}\ s \neq []$
and *R*: $\text{lasso}\ s = None$
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 $\text{se-back-edge}\ (\text{hd}\ (\text{stack}\ s))\ v\ s'$
 $\leq \text{SPEC}\ (\lambda e. \text{DFS-invar}\ G\ \text{blue-dfs-params}\ (s' \setminus \{ \text{state.more} := e \})) \longrightarrow$
 $\text{lasso-inv}\ (s' \setminus \{ \text{state.more} := e \})$
 $\langle proof \rangle$

lemma *lasso-inv*:

$is_invar\ \text{lasso-inv}$
 $\langle proof \rangle$

end

context *BlueDFS-invar*

begin

lemmas $s\text{-lasso-inv} = \text{lasso-inv}[THEN \text{make-invar-thm}]$

lemma

assumes $\text{lasso } s = \text{Some } (pr, pl)$
shows $\text{loop-nonempty}: pl \neq []$
and $\text{accpt-loop}: \text{accpt } (hd \ pl)$
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 $\text{case } \text{lasso } s \text{ of}$
 $\text{None} \Rightarrow \neg(\exists v0 \in V0. \exists v. (v0, v) \in E^* \wedge \text{accpt } v \wedge (v, v) \in E^+)$
 $| \text{Some } (pr, pl) \Rightarrow (\exists v0 \in V0. \exists v.$
 $\text{path } E \ v0 \ pr \ v \wedge \text{accpt } v \wedge pl \neq [] \wedge \text{path } E \ v \ pl \ v)$
 $\langle \text{proof} \rangle$

end

2.5.4 Interface

interpretation $\text{BlueDFS-defs for } G \text{ accpt } \langle \text{proof} \rangle$

definition $\text{nested-dfs-spec } G \text{ accpt} \equiv \lambda r. \text{case } r \text{ of}$

$\text{None} \Rightarrow \neg(\exists v0 \in g\text{-}V0 \ G. \exists v. (v0, v) \in (g\text{-}E \ G)^* \wedge \text{accpt } v \wedge (v, v) \in (g\text{-}E \ G)^+)$
 $| \text{Some } (pr, pl) \Rightarrow (\exists v0 \in g\text{-}V0 \ G. \exists v.$
 $\text{path } (g\text{-}E \ G) \ v0 \ pr \ v \wedge \text{accpt } v \wedge pl \neq [] \wedge \text{path } (g\text{-}E \ G) \ v \ pl \ v)$

definition $\text{nested-dfs } G \text{ accpt} \equiv \text{do } \{$

$\text{ASSERT } (\text{fb-graph } G);$
 $s \leftarrow \text{it-dfs } TYPE('a) \ G \ \text{accpt};$
 $\text{RETURN } (\text{lasso } s)$
 $\}$

theorem $\text{nested-dfs-correct}$:

assumes $\text{fb-graph } G$
shows $\text{nested-dfs } G \text{ accpt} \leq \text{SPEC } (\text{nested-dfs-spec } G \text{ accpt})$
 $\langle \text{proof} \rangle$

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 $\text{bdfs-erel} \equiv \{(\langle \text{lasso-impl}=li, \text{red-impl}=ri \rangle, \langle \text{lasso}=l, \text{red}=r \rangle) \mid$
 $li \ ri \ l \ r. li=l \wedge ri=r\}$

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

$\text{Inl } R' \Rightarrow \langle \text{lasso-impl} = \text{None}, \text{red-impl} = (R' \text{ ~~red-impl~~}) \rangle$

$\mid \text{Inr } (vs, v) \Rightarrow \text{let}$

$rs = \text{rev } (\text{map fst } (\text{CAST } (\text{ss-stack } s)))$

$\text{in } \langle$

$\text{lasso-impl} = \text{Some } (rs, \text{vs@dropWhileNot } v \text{ rs}),$

$\text{red-impl} = \text{red-impl } s \rangle$

lemma *mk-blue-witness-impl[refine]*:

$\llbracket (si, s) \in \text{bdfs-rel}; (ri, r) \in \langle \text{Id}, \langle \text{Id} \rangle \text{list-rel} \times_r \text{Id} \rangle \text{sum-rel} \rrbracket$

$\implies (\text{mk-blue-witness-impl } si \text{ } ri, \text{mk-blue-witness } s \text{ } r) \in \text{bdfs-erel}$

$\langle \text{proof} \rangle$

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

$\text{run-red-dfs-impl } G \text{ } u \text{ } s \equiv \text{case lasso-impl } s \text{ of None} \Rightarrow \text{do } \{$

$\text{redS} \leftarrow \text{red-dfs TYPE('more)} \text{ } G \text{ } (\text{red-impl } s) \text{ } (\lambda x. x = u \vee x \in \text{cyan-impl}$

$s) \text{ } u;$

$\text{RETURN } (\text{mk-blue-witness-impl } s \text{ redS})$

$\}$

$\mid - \Rightarrow \text{RETURN } (\text{simple-state.more } s)$

lemma *run-red-dfs-impl[refine]*: $\llbracket (Gi, G) \in \text{Id}; (ui, u) \in \text{Id}; (si, s) \in \text{bdfs-rel} \rrbracket$

$\implies \text{run-red-dfs-impl } Gi \text{ } ui \text{ } si \leq \text{bdfs-erel } (\text{run-red-dfs TYPE('a)} \text{ } G \text{ } u \text{ } s)$

$\langle \text{proof} \rangle$

definition *se-back-edge-impl* $\text{accpt } u \text{ } v \text{ } s \equiv \text{case lasso-impl } s \text{ of}$

$\text{None} \Rightarrow$

$\text{if accpt } u \text{ then}$

$\text{let } rs = \text{rev } (\text{map fst } (\text{tl } (\text{CAST } (\text{ss-stack } s))));$

$ur = rs;$

$ul = u \# \text{dropWhileNot } v \text{ rs}$

$\text{in RETURN } \langle \text{lasso-impl} = \text{Some } (ur, ul), \text{red-impl} = \text{red-impl } s \rangle$

$\text{else if accpt } v \text{ then}$

$\text{let } rs = \text{rev } (\text{map fst } (\text{CAST } (\text{ss-stack } s)));$

$vr = \text{takeWhileNot } v \text{ rs};$

$vl = \text{dropWhileNot } v \text{ rs}$

$\text{in RETURN } \langle \text{lasso-impl} = \text{Some } (vr, vl), \text{red-impl} = \text{red-impl } s \rangle$

else RETURN (simple-state.more s)
 | - $\Rightarrow$ RETURN (simple-state.more s)

lemma *se-back-edge-impl*[refine]: $\llbracket (accpti, accpt) \in Id; (ui, u) \in Id; (vi, v) \in Id; (si, s) \in bdfs\text{-}rel$
 $\rrbracket$
 $\Rightarrow se\text{-}back\text{-}edge\text{-}impl\ accpt\ ui\ vi\ si \leq \Downarrow bdfs\text{-}erel\ (se\text{-}back\text{-}edge\ accpt\ u\ v\ s)$
 $\langle proof \rangle$

lemma *NOOP-impl*: $(si, s) \in bdfs\text{-}rel$
 $\Rightarrow RETURN\ (simple\text{-}state.more\ si) \leq \Downarrow bdfs\text{-}erel\ (NOOP\ s)$
 $\langle proof \rangle$

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* $\equiv \llbracket$
on-init = RETURN ($\llbracket$ lasso-impl = None, red-impl = $\{\}$ $\rrbracket$),
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 = $\lambda s.\ lasso\text{-}impl\ s \neq None\ \rrbracket$

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* $\langle proof \rangle$

context *BlueDFS* **begin**

sublocale *impl*: *simple-impl* *G* *blue-dfs-params* *bdfs-params-impl* *G* *accpt* *bdfs-erel*
 $\langle proof \rangle$

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 $nested-dfs-impl\ Gi\ accpti \leq \Downarrow (\langle \langle Id \rangle list-rel \times_r \langle Id \rangle list-rel \rangle option-rel)$
 $(nested-dfs\ G\ accpt)$
 $\langle proof \rangle$

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*] =
 $TYRELI[\textbf{where } R = \langle V \rangle dflt-ahs-rel]$
 $TYRELI[\textbf{where } R = \langle V \rangle ras-rel]$
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 $(\lambda G \text{ accpt. } nres\text{-of } (nested\text{-dfs-code } G \text{ accpt}), nested\text{-dfs}) \in$
 $\langle Rm, V \rangle g\text{-impl-rel-ext} \rightarrow (V \rightarrow bool\text{-rel}) \rightarrow$
 $\langle \langle \langle V \rangle list\text{-rel} \times_r \langle V \rangle list\text{-rel} \rangle option\text{-rel} \rangle nres\text{-rel}$
 $\langle proof \rangle$

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

$lowlink\text{-path } s \ v \ p \ w \equiv path \ E \ v \ p \ w \wedge p \neq []$
 $\wedge (last \ p, w) \in cross\text{-edges } s \cup back\text{-edges } s$
 $\wedge (length \ p > 1 \longrightarrow$
 $\quad p!1 \in dom \ (finished \ s)$
 $\wedge (\forall k < length \ p - 1. (p!k, p!Suc \ k) \in tree\text{-edges } s))$

definition

$lowlink\text{-set } s \ v \equiv \{w \in dom \ (discovered \ s).$
 $\quad v = w$
 $\vee (v, w) \in E^+ \wedge (w, v) \in E^+$
 $\wedge (\exists p. lowlink\text{-path } s \ v \ p \ w)\}$

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

abbreviation *LowLink* **where**

$LowLink \ s \ v \equiv Min \ (\delta \ s \ ' lowlink\text{-set } s \ v)$

end

end

context *DFS-invar* **begin**

lemma *lowlink-setI*:

assumes *lowlink-path* *s v p w*
and $w \in \text{dom } (\text{discovered } s)$
and $(v, w) \in E^* \ (w, v) \in E^*$
shows $w \in \text{lowlink-set } s \ v$

<proof>

lemma *lowlink-set-discovered*:

lowlink-set *s v* $\subseteq \text{dom } (\text{discovered } s)$
<proof>

lemma *lowlink-set-finite*[*simp, intro!*]:

finite (*lowlink-set* *s v*)
<proof>

lemma *lowlink-set-not-empty*:

assumes $v \in \text{dom } (\text{discovered } s)$
shows *lowlink-set* *s v* $\neq \{\}$
<proof>

lemma *lowlink-path-single*:

assumes $(v, w) \in \text{cross-edges } s \cup \text{back-edges } s$
shows *lowlink-path* *s v [v] w*
<proof>

lemma *lowlink-path-Cons*:

assumes *lowlink-path* *s v (x#xs) w*
and $xs \neq []$
shows $\exists u. \text{lowlink-path } s \ u \ xs \ w$
<proof>

lemma *lowlink-path-in-tree*:

assumes *p*: *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*: *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*: *lowlink-path* *s v p w*
and *tree-edges*: $(u, v) \in (\text{tree-edges } s)^+$
and *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  $\equiv$  dropWhile (( $\neq$ ) u) tjs
  defines tw  $\equiv$  takeWhile (( $\neq$ ) u) tjs
  assumes u  $\in$  set tjs
  shows set tjs = set (tl dw)  $\cup$  insert u (set tw)
<proof>

```

2.7.2 Instantiation of the DFS-Framework

```

record 'v tarjan-state = 'v state +
  sccs :: 'v set set
  lowlink :: 'v  $\rightarrow$  nat
  tj-stack :: 'v list

```

```

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

```

```

abbreviation the-lowlink s v  $\equiv$  the (lowlink s v)

```

```

context timing-syntax
begin
  notation the-lowlink ( $\lrcorner\zeta\rangle$ )
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  $\Rightarrow$  'v tarjan-state  $\Rightarrow$  ('v,unit) tarjan-state-ext nres
  where
    tarjan-disc v s = RETURN ( $\emptyset$  sccs = sccs s,

```

$lowlink = (lowlink\ s)(v \mapsto \delta\ s\ v),$
 $tj_stack = v \# tj_stack\ s \parallel$

definition $tj_stack_pop :: 'v\ list \Rightarrow 'v \Rightarrow ('v\ list \times 'v\ set)\ nres$ **where**
 $tj_stack_pop\ tjs\ u = RETURN\ (tl\ (dropWhile\ ((\neq)\ u)\ tjs),\ insert\ u\ (set\ (takeWhile\ ((\neq)\ u)\ tjs)))$

lemma $tj_stack_pop_set$:
 $tj_stack_pop\ tjs\ u \leq SPEC\ (\lambda(tjs', scc).\ u \in set\ tjs \longrightarrow set\ tjs = set\ tjs' \cup scc \wedge u \in scc)$
 $\langle proof \rangle$

lemmas $tj_stack_pop_set_leof_rule = weaken_SPEC[OF\ tj_stack_pop_set,\ THEN\ leof_lift]$

definition $tarjan_fin :: 'v \Rightarrow 'v\ tarjan_state \Rightarrow ('v, unit)\ tarjan_state_ext\ nres$ **where**

$tarjan_fin\ v\ s = do\ \{$
 $\quad let\ ll = (if\ stack\ s = []\ then\ lowlink\ s$
 $\quad\quad else\ let\ u = hd\ (stack\ s)\ in$
 $\quad\quad\quad (lowlink\ s)(u \mapsto min\ (\zeta\ s\ u)\ (\zeta\ s\ v)));$
 $\quad let\ s' = s \parallel lowlink := ll \parallel;$

 $\quad ASSERT\ (v \in set\ (tj_stack\ s));$
 $\quad ASSERT\ (distinct\ (tj_stack\ s));$
 $\quad if\ \zeta\ s\ v = \delta\ s\ v\ then\ do\ \{$
 $\quad\quad ASSERT\ (scc_root'\ E\ s\ v\ (scc_of\ E\ v));$
 $\quad\quad (tjs, scc) \leftarrow tj_stack_pop\ (tj_stack\ s)\ v;$
 $\quad\quad RETURN\ (state.more\ (s' \parallel tj_stack := tjs,\ sccs := insert\ scc\ (sccs\ s) \parallel))$
 $\quad\} else\ do\ \{$
 $\quad\quad ASSERT\ (\neg\ scc_root'\ E\ s\ v\ (scc_of\ E\ v));$
 $\quad\quad RETURN\ (state.more\ s')$
 $\quad\}$
 $\quad\}$

definition $tarjan_back :: 'v \Rightarrow 'v \Rightarrow 'v\ tarjan_state \Rightarrow ('v, unit)\ tarjan_state_ext\ nres$ **where**

$tarjan_back\ u\ v\ s = ($
 $\quad if\ \delta\ s\ v < \delta\ s\ u \wedge v \in set\ (tj_stack\ s)\ then$
 $\quad\quad let\ ul' = min\ (\zeta\ s\ u)\ (\delta\ s\ v)$
 $\quad\quad in\ RETURN\ (state.more\ (s \parallel lowlink := (lowlink\ s)(u \mapsto ul') \parallel))$
 $\quad else\ NOOP\ s)$

end

definition $tarjan_params :: 'v\ tarjan_param$ **where**

$tarjan_params = \parallel$
 $\quad on_init = RETURN\ \parallel sccs = \{\},\ lowlink = Map.empty,\ tj_stack = [] \parallel,$
 $\quad on_new_root = tarjan_disc,$
 $\quad on_discover = \lambda u.\ tarjan_disc,$
 $\quad on_finish = tarjan_fin,$

$on-back-edge = tarjan-back,$
 $on-cross-edge = tarjan-back,$
 $is-break = \lambda s. False \mid$

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 $[simp]:$

$sccs\ (empty-state\ (\mid sccs = s, lowlink = l, tj-stack = t \mid)) = s$
 $lowlink\ (empty-state\ (\mid sccs = s, lowlink = l, tj-stack = t \mid)) = l$
 $tj-stack\ (empty-state\ (\mid sccs = s, lowlink = l, tj-stack = t \mid)) = t$
 $\langle proof \rangle$

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 $[simp]:$

$s \mid state.more := (\mid sccs = sc, lowlink = l, tj-stack = t \mid)$
 $= s \mid sccs := sc, lowlink := l, tj-stack := t \mid$
 $\langle proof \rangle$

end

locale $Tarjan-invar = Tarjan +$
 $DFS-invar\ \mathbf{where}\ param = tarjan-params$

context $Tarjan-def\ \mathbf{begin}$

lemma $Tarjan-invar-eq[simp]:$

$DFS-invar\ G\ tarjan-params\ s \longleftrightarrow Tarjan-invar\ G\ s\ (\mathbf{is}\ ?D \longleftrightarrow ?T)$

$\langle proof \rangle$
end

2.7.3 Correctness Proof

context Tarjan begin

lemma *i-tj-stack-discovered*:

is-invar ($\lambda s. \text{set } (tj\text{-stack } s) \subseteq \text{dom } (discovered\ s)$)

$\langle proof \rangle$

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

lemma *i-tj-stack-distinct*:

is-invar ($\lambda s. \text{distinct } (tj\text{-stack } s)$)

$\langle proof \rangle$

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

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

lemma *i-tj-stack-incr-disc*:

is-invar ($\lambda s. \forall k < \text{length } (tj\text{-stack } s). \forall j < k. \delta\ s\ (tj\text{-stack } s\ !\ j) > \delta\ s\ (tj\text{-stack } s\ !\ k)$)

$\langle proof \rangle$

end end

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

lemma *tj-stack-incr-disc*:

assumes $k < \text{length } (tj\text{-stack } s)$

and $j < k$

shows $\delta\ s\ (tj\text{-stack } s\ !\ j) > \delta\ s\ (tj\text{-stack } s\ !\ k)$

$\langle proof \rangle$

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

$\langle proof \rangle$

end end

context Tarjan begin context begin interpretation timing-syntax $\langle proof \rangle$

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

is-invar ($\lambda s. \bigcup (\text{sccs } s) \subseteq \text{dom } (finished\ s) \wedge \text{set } (stack\ s) \subseteq \text{set } (tj\text{-stack } s)$)

$\langle proof \rangle$

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

$is-invar \ (\lambda s. \ set \ (tj-stack \ s) \subseteq \ set \ (stack \ s) \cup \ dom \ (finished \ s))$
 $\langle proof \rangle$

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

context *Tarjan-invar* **begin**
lemmas *finished-ss-sccs-tj-stack* =
 $i-finished-ss-sccs-tj-stack[THEN \ make-invar-thm]$

lemmas *tj-stack-ss-stack-finished* =
 $i-tj-stack-ss-stack-finished[THEN \ make-invar-thm]$

lemma *sccs-finished*:
 $\bigcup (sccs \ s) \subseteq \ dom \ (finished \ s)$
 $\langle proof \rangle$

lemma *stack-ss-tj-stack*:
 $set \ (stack \ s) \subseteq \ set \ (tj-stack \ s)$
 $\langle proof \rangle$

lemma *hd-stack-in-tj-stack*:
 $stack \ s \neq [] \implies hd \ (stack \ s) \in \ set \ (tj-stack \ s)$
 $\langle proof \rangle$
end

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

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

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

lemma *tj-stack-reach-stack*:
assumes $u \in \ set \ (tj-stack \ s)$
shows $\exists v \in \ set \ (stack \ s). \ (u, v) \in E^* \wedge \delta \ s \ v \leq \delta \ s \ u$
 $\langle proof \rangle$

```

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:
     $is\text{-invar } (\lambda s. \forall scc \in sccs\ s. is\text{-scc } E\ scc)$ 
     $\langle proof \rangle$ 
  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:
     $is\text{-invar } (\lambda s. \forall x \in \text{dom } (discovered\ s). \zeta\ s\ x = LowLink\ s\ x)$ 
     $\langle proof \rangle$ 
  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  $\zeta\ s\ v = \delta\ s\ v \longleftrightarrow scc\text{-root } s\ v\ (scc\text{-of } E\ v)$ 
     $\langle proof \rangle$ 

  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')
   $\langle proof \rangle$ 

  sublocale DFS G tarjan-params
     $\langle proof \rangle$ 
end

interpretation tarjan: Tarjan-def for G  $\langle proof \rangle$ 

```

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 ( $\lambda sccs. (\forall scc \in sccs. is\_scc (g\text{-}E\ G\ scc)$ 
     $\wedge \bigcup sccs = tarjan.reachable\ TYPE('a)\ G)$ )}

lemma tarjan-correct:
  tarjan G  $\leq$  tarjan-spec G
   $\langle proof \rangle$ 

end

```