A Framework for Verifying Depth-First Search Algorithms

Peter Lammich and René Neumann

August 16, 2018
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 3
   1.1 General DFS with Hooks 3
      1.1.1 State and Parameterization 3
      1.1.2 DFS operations 4
      1.1.3 DFS Algorithm 9
      1.1.4 Invariants 10
      1.1.5 Basic Invariants 19
      1.1.6 Total Correctness 24
      1.1.7 Non-Failing Parameterization 26
   1.2 Basic Invariant Library 28
      1.2.1 Basic Timing Invariants 28
      1.2.2 Parenthesis Theorem 33
      1.2.3 Edge Types 36
      1.2.4 White Path Theorem 60
   1.3 Invariants for SCCs 62
   1.4 Generic DFS and Refinement 70
      1.4.1 Generic DFS Algorithm 70
      1.4.2 Refinement Between DFS Implementations 78
   1.5 Tail-Recursive Implementation 83
   1.6 Recursive DFS Implementation 90
   1.7 Simple Data Structures 101
      1.7.1 Stack, Pending Stack, and Visited Set 101
      1.7.2 Simple state without on-stack 109
      1.7.3 Simple state without stack and on-stack 110
   1.8 Restricting Nodes by Pre-Initializing Visited Set 112
   1.9 Basic DFS Framework 120

2 Examples 121
   2.1 Simple Cyclicity Checker 121
      2.1.1 Framework Instantiation 121
      2.1.2 Correctness Proof 123
      2.1.3 Implementation 126
      2.1.4 Synthesizing Executable Code 129
2.2 Finding a Path between Nodes ........................................... 131
  2.2.1 Including empty Path ........................................... 132
  2.2.2 Restricting the Graph ........................................... 136
  2.2.3 Path of Minimal Length One, with Restriction .............. 138
  2.2.4 Path of Minimal Length One, without Restriction .......... 141
  2.2.5 Implementation ............................................... 141
  2.2.6 Synthesis of Executable Code ................................. 144
  2.2.7 Conclusion .................................................. 147

2.3 Set of Reachable Nodes .................................................. 148
  2.3.1 Preliminaries .................................................. 148
  2.3.2 Framework Instantiation ...................................... 149
  2.3.3 Correctness ................................................... 149
  2.3.4 Synthesis of Executable Implementation ...................... 151
  2.3.5 Conclusions .................................................. 154

2.4 Find a Feedback Arc Set ................................................. 154
  2.4.1 Instantiation of the DFS-Framework ........................... 155
  2.4.2 Correctness Proof ............................................. 156
  2.4.3 Implementation ............................................... 157
  2.4.4 Synthesis of Executable Code ................................ 158
  2.4.5 Feedback Arc Set with Initialization ......................... 160
  2.4.6 Conclusion .................................................. 163

2.5 Nested DFS ............................................................. 163
  2.5.1 Auxiliary Lemmas .............................................. 163
  2.5.2 Instantiation of the Framework ................................ 163
  2.5.3 Correctness Proof ............................................. 166
  2.5.4 Interface ..................................................... 177
  2.5.5 Implementation ............................................... 178
  2.5.6 Synthesis of Executable Code ................................ 181
  2.5.7 Conclusion .................................................. 182

2.6 Invariants for Tarjan’s Algorithm .................................... 183

2.7 Tarjan’s Algorithm ..................................................... 193
  2.7.1 Preliminaries .................................................. 194
  2.7.2 Instantiation of the DFS-Framework ........................... 194
  2.7.3 Correctness Proof ............................................. 197
  2.7.4 Interface ..................................................... 223
Chapter 1

The DFS Framework

This chapter contains the basic DFS Framework

1.1 General DFS with Hooks

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

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

1.1.1 State and Parameterization

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

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

abbreviation NOOP s ≡ RETURN (state.more s)

Record holding the parameterization.

record ('v,'s,'es) gen-parameterization =
on-init :: 'es nres
on-new-root :: 'v ⇒ 's ⇒ 'es nres
on-discover :: 'v ⇒ 'v ⇒ 's ⇒ 'es nres
on-finish :: 'v ⇒ 's ⇒ 'es nres
on-back-edge :: 'v ⇒ 'v ⇒ 's ⇒ 'es nres
on-cross-edge :: 'v ⇒ 'v ⇒ 's ⇒ 'es nres
is-break :: 's ⇒ bool

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

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

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

definition \more init. dflt-parametrization more init ≡ ()
on-init = init,
on-new-root = λ- . RETURN o more,
on-discover = λ- . RETURN o more,
on-finish = λ- . RETURN o more,
on-back-edge = λ- . RETURN o more,
on-cross-edge = λ- . RETURN o more,
is-break = λ- . False

lemmas dflt-parametrization-simp[simp] =
gen-parameterization.simps[mk-record-simp, OF dflt-parametrization-def]

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

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

1.1.2 DFS operations

Node predicates

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

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

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

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

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

**definition** discover

\[ :: 'v \Rightarrow 'v \Rightarrow ('v,'es) state-scheme \Rightarrow ('v,'es) state-scheme \]

**where**

discover u v s ≡ let

\[ d = (\text{discovered } s)(v \mapsto \text{counter } s); c = \text{counter } s + 1; \]

\[ st = v\#stack s; \]

\[ p = \text{pending } s \cup \{v\} \times E''\{v\}; \]

\[ t = \text{insert } (u,v) (\text{tree-edges } s) \]

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

**lemma** discover-simps [simp]:

\[ \text{counter } (\text{discover } u v s) = \text{Suc } (\text{counter } s) \]

\[ \text{discovered } (\text{discover } u v s) = (\text{discovered } s)(v \mapsto \text{counter } s) \]

\[ \text{finished } (\text{discover } u v s) = \text{finished } s \]

\[ \text{stack } (\text{discover } u v s) = v\#stack s \]

\[ \text{pending } (\text{discover } u v s) = \text{pending } s \cup \{v\} \times E''\{v\} \]

\[ \text{tree-edges } (\text{discover } u v s) = \text{insert } (u,v) (\text{tree-edges } s) \]

\[ \text{cross-edges } (\text{discover } u v s) = \text{cross-edges } s \]

\[ \text{back-edges } (\text{discover } u v s) = \text{back-edges } s \]

\[ \text{state}.\text{more } (\text{discover } u v s) = \text{state}.\text{more } s \]

by (simp-all add: discover-def)

**definition** finish

\[ :: 'v \Rightarrow ('v,'es) state-scheme \Rightarrow ('v,'es) state-scheme \]

**where**

finish u s ≡ let

\[ f = (\text{finished } s)(u \mapsto \text{counter } s); c = \text{counter } s + 1; \]

\[ st = \text{tl } (\text{stack } s) \]

in s (\| finished := f, counter := c, stack := st|)

**lemma** finish-simps [simp]:

\[ \text{counter } (\text{finish } u s) = \text{Suc } (\text{counter } s) \]

\[ \text{discovered } (\text{finish } u s) = \text{discovered } s \]

\[ \text{finished } (\text{finish } u s) = (\text{finished } s)(u \mapsto \text{counter } s) \]

\[ \text{stack } (\text{finish } u s) = \text{tl } (\text{stack } s) \]

\[ \text{pending } (\text{finish } u s) = \text{pending } s \]

\[ \text{tree-edges } (\text{finish } u s) = \text{tree-edges } s \]

\[ \text{cross-edges } (\text{finish } u s) = \text{cross-edges } s \]

\[ \text{back-edges } (\text{finish } u s) = \text{back-edges } s \]

\[ \text{state}.\text{more } (\text{finish } u s) = \text{state}.\text{more } s \]

by (simp-all add: finish-def)

**definition** back-edge

\[ :: 'v \Rightarrow 'v \Rightarrow ('v,'es) state-scheme \Rightarrow ('v,'es) state-scheme \]

**where**

back-edge u v s ≡ let
\[ b = \text{insert}(u,v) \ (\text{back-edges} \ s) \]

in \( s[\text{back-edges} := b] \)

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

**Definition** cross-edge

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

**Where**

\[ \text{cross-edge} \ u \ v \ s \equiv \text{let} \ c = \text{insert}(u,v) \ (\text{cross-edges} \ s) \]

in \( s[\text{cross-edges} := c] \)

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

**Definition** new-root

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

**Where**

\[ \text{new-root} \ v0 \ s \equiv \text{let} \ c = \text{Suc} \ (\text{counter} \ s); \]

\[ d = (\text{discovered} \ s)(v0 \mapsto \text{counter} \ s); \]

\[ p = \{v0\} \times E'' \{v0\}; \]

\[ st = [v0] \]

in \( s[\text{counter} := c, \text{discovered} := d, \text{pending} := p, \text{stack} := st] \)

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

6
stack \( (\text{new-root } v0 \ s) = [v0] \)
pending \( (\text{new-root } v0 \ s) = (\{v0\} \times E' \{v0\}) \)
tree-edges \( (\text{new-root } v0 \ s) = \text{tree-edges } s \)
cross-edges \( (\text{new-root } v0 \ s) = \text{cross-edges } s \)
back-edges \( (\text{new-root } v0 \ s) = \text{back-edges } s \)
state.more \( (\text{new-root } v0 \ s) = \text{state.more } s \)
by \( (\text{simp-all add: new-root-def}) \)

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

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

Effects on Whole State

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

definition do-cross-edge
\:: \( 'v \Rightarrow 'v \Rightarrow ('v, 'es) \text{ state-scheme} \Rightarrow ('v, 'es) \text{ state-scheme } nres \)
where
do-cross-edge \( u \ v \ s \equiv \{ \)
\[ \text{let } s = \text{cross-edge } u \ v \ s; \]
\[ e \leftarrow \text{on-cross-edge param } u \ v \ s; \]
\[ \text{RETURN } (s[\text{state.more} := e]) \]
\}

definition do-back-edge
\:: \( 'v \Rightarrow 'v \Rightarrow ('v, 'es) \text{ state-scheme} \Rightarrow ('v, 'es) \text{ state-scheme } nres \)
where
do-back-edge \( u \ v \ s \equiv \{ \)
\[ \text{let } s = \text{back-edge } u \ v \ s; \]
\[ e \leftarrow \text{on-back-edge param } u v s; \]
\[ \text{RETURN } (s[\text{state}.more := e]) \]

**definition** do-known-edge
\[ :: 'v \Rightarrow 'v \Rightarrow ('v', 'es) \text{ state-scheme } \Rightarrow ('v', 'es) \text{ state-scheme } nres \]
**where**
do-known-edge u v s ≡
\[ \text{if } \text{is-finished } v s \text{ then } \]
do-cross-edge u v s
\[ \text{else } \]
do-back-edge u v s

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

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

**definition** get-new-root where
\[ \text{get-new-root } s \equiv \text{SPEC } (\lambda v. v \in V0 \land \text{is-discovered } v s) \]

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

**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 {
\[ e \leftarrow \text{on-init param}; \]
\[ \text{RETURN } (\text{empty-state } e) \]

1.1.3 DFS Algorithm

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

**definition** cond :: ('v, 'es) state-scheme ⇒ bool where

\[
\text{cond } s \leftarrow (V0 \subseteq \{ v. \text{is-discovered } v \ s \} \rightarrow \neg \text{is-empty-stack } s) \\
\wedge \neg \text{is-break param } s
\]

**lemma** cond-alt:

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

**apply** (rule ext)

**unfolding** cond-def is-discovered-def is-empty-stack-def

**by** auto

**definition** get-pending ::

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

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

**where** get-pending s ≡ do {

\[
\text{let } u = \text{hd } (\text{stack } s); \\
\text{let } Vs = \text{pending } s \ ' ' \{u\}; \\
\]

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

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

**where**

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


do-known-edge \( u \) \( \rightarrow \) \( v \) \( \rightarrow \) \( s \)
else
   do-discover \( u \) \( \rightarrow \) \( v \) \( \rightarrow \) \( s \)
}\
\
\textbf{definition} \( \text{it-dfs} \equiv \text{init} \Rightarrow \text{WHILE} \ \text{cond} \ \text{step} \)
\textbf{definition} \( \text{it-dfsT} \equiv \text{init} \Rightarrow \text{WHILET} \ \text{cond} \ \text{step} \)
end

\subsection{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 \textit{param-DFS} locale.

2. Next, we transfer the invariant to the \textit{DFS-invar}-locale.

\textbf{locale} \text{param-DFS} = 
\hspace{1em} \text{fb-graph} \ G + \ \text{param-DFS-defs} \ G \ \text{param}
\hspace{1em} \text{for} \ G :: (\mathit{\text{v}}, \mathit{\text{more}}) \ \text{graph-rec-scheme}
\hspace{1em} \text{and} \ \text{param} :: (\mathit{\text{v}}, \mathit{\text{es}}) \ \text{parameterization}
\hspace{1em} \begin{array}{c}
\text{begin} \\
\text{definition} \ \text{is-invar} :: (\mathit{\text{v}}, \mathit{\text{es}}) \ \text{state-scheme} \Rightarrow \text{bool} \Rightarrow \text{bool} \\
\hspace{1em} \\
\hspace{1em} \text{— Predicate that states that } I \text{ is an invariant.} \\
\hspace{1em} \text{where} \ is-invar \ I \equiv is-rwof-invar \ init \ cond \ step \ I \\
\end{array}
\hspace{1em} \begin{array}{c}
\text{end} \\
\end{array}

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

\textbf{locale} \text{DFS-invar} = 
\hspace{1em} \text{param-DFS} \ G \ \text{param}
\hspace{1em} \text{for} \ G :: (\mathit{\text{v}}, \mathit{\text{more}}) \ \text{graph-rec-scheme}
\hspace{1em} \text{and} \ \text{param} :: (\mathit{\text{v}}, \mathit{\text{es}}) \ \text{parameterization}
\hspace{1em} + 
\hspace{1em} \text{fixes} \ s :: (\mathit{\text{v}}, \mathit{\text{es}}) \ \text{state-scheme}
\hspace{1em} \text{assumes} \ \text{rwof} :: \text{rwof init cond step} \ s
\hspace{1em} \begin{array}{c}
\text{begin} \\
\text{lemma} \ \text{make-invar-thm}: \ \text{is-invar} \ I \implies I \ s \\
\end{array}

Lemma to transfer an invariant into this locale using \texttt{rwof-cons[OF - rwof, folded is-invar-def]}.

end

Establishing Invariants

context param-DFS

begin

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

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

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

lemma \texttt{DFS-invar-step: \[nofail it-dfs; DFS-invar G param s; cond s\]}
\implies step s \leq SPEC (DFS-invar G param)
— A step preserves the (best) invariant.
unfolding \texttt{it-dfs-def rwof-eq-DFS-invar[symmetric]}
by (rule rwof-step)

lemma \texttt{DFS-invar-step': \[nofail (step s); DFS-invar G param s; cond s\]}
\implies step s \leq SPEC (DFS-invar G param)
unfolding \texttt{it-dfs-def rwof-eq-DFS-invar[symmetric]}
by (rule rwof-step')

We define symbolic names for the preconditions of certain operations

definition \texttt{pre-is-break s \equiv DFS-invar G param s}

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

definition \texttt{pre-on-finish u s' \equiv \exists s.}
\texttt{DFS-invar G param s \land cond s \land}
\texttt{stack s \neq [] \land u = hd (stack s) \land pending s ' {u} = { } \land s' = finish u s}
Next, we define a set of rules to establish an invariant.

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

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

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

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

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

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

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

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

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

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

assumes discover: $\forall s s' u v.
[DFS-invar G param s; I s; cond s; \neg is-break param s;
stack s \neq []; (u, v) \in pending s; u = hd (stack s);
v \notin dom (discovered s);
\Rightarrow on-discover param u v s' \leq n
SPEC (\lambda x. DFS-invar G param (s'(\{\text{state.}more := x\}))
\rightarrow I (s'(\{\text{state.}more := x\})))]

shows is-invar I
unfolding is-invar-def

proof

show init $\leq_n SPEC I
unfolding init-def
by (refine-rcg refine-vcg) (simp add: init)

next

fix s
assume rwof init cond step s and IC: I s cond s
hence DI: DFS-invar G param s by unfold-locales
then interpret DFS-invar G param s .

from (cond s) have IB: $\neg is-break param s by (simp add: cond-def)

have B: step s $\leq_n SPEC (DFS-invar G param)
by rule (metis DFS-invar-step’ DI (cond s))

note rule-assms = DI IC IB

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

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

qed
lemma establish-invarI\,[@case-names\,init\,new-root\,finish\,cross-edge\,back-edge\,discover\]:
  --- Establish a DFS invariant (symbolic preconditions).
  assumes init: on-init param \(\leq_n\) SPEC (\(\lambda x. I\) (empty-state \(x\)))
  assumes new-root: \(\forall s' v0.\) pre-on-new-root \(v0\) \(s'\)
  \[\implies\] on-new-root param \(v0\) \(s'\) \(\leq_n\)
  SPEC (\(\lambda x. DFS\\text{-invar} G\) param \(s'\)[state.more := \(x\)])
  \[\implies\] I (\(s'\)[state.more := \(x\)])
  assumes finish: \(\forall s' u.\) pre-on-finish \(u\) \(s'\)
  \[\implies\] on-finish param \(u\) \(s'\) \(\leq_n\)
  SPEC (\(\lambda x. DFS\\text{-invar} G\) param \(s'\)[state.more := \(x\)])
  \[\implies\] I (\(s'\)[state.more := \(x\)])
  assumes cross-edge: \(\forall s' u v.\) pre-on-cross-edge \(u\) \(v\) \(s'\)
  \[\implies\] on-cross-edge param \(u\) \(v\) \(s'\) \(\leq_n\)
  SPEC (\(\lambda x. DFS\\text{-invar} G\) param \(s'\)[state.more := \(x\)])
  \[\implies\] I (\(s'\)[state.more := \(x\)])
  assumes back-edge: \(\forall s' u v.\) pre-on-back-edge \(u\) \(v\) \(s'\)
  \[\implies\] on-back-edge param \(u\) \(v\) \(s'\) \(\leq_n\)
  SPEC (\(\lambda x. DFS\\text{-invar} G\) param \(s'\)[state.more := \(x\)])
  \[\implies\] I (\(s'\)[state.more := \(x\)])
  assumes discover: \(\forall s' u v.\) pre-on-discover \(u\) \(v\) \(s'\)
  \[\implies\] on-discover param \(u\) \(v\) \(s'\) \(\leq_n\)
  SPEC (\(\lambda x. DFS\\text{-invar} G\) param \(s'\)[state.more := \(x\)])
  \[\implies\] I (\(s'\)[state.more := \(x\)])

shows is-invar \(I\)
apply (rule establish-invarI)
using assms
unfolding pre-on-defs
apply 
apply blast
apply (rprems,blast)+
done

lemma establish-invarI-ND\,[@case-names\,prereq\,init\,new-discover\,finish\,cross-edge\,back-edge\]:
  --- Establish a DFS invariant (new-root and discover cases are combined).
  assumes prereq: \(\lambda u v s.\) on-discover param \(u\) \(v\) \(s\) = on-new-root param \(v\) \(s\)
  assumes init: on-init param \(\leq_n\) SPEC (\(\lambda x. I\) (empty-state \(x\)))
  assumes new-discover: \(\forall s s' v.\)
  \[\text{DFS\\text{-invar} G\,param\,s;\,I\,s;\,cond\,s;\,\neg\,\text{is\,-{}break\,param\,s;}}\]
  \(v \notin\) dom (discovered \(s\));
  discovered \(s' = (\text{discovered\,s})\,\text{({v\rightarrow\text{counter\,s}})};\,\text{finished\,s' = finished\,s};\)
  counter \(s' = \text{Suc}\,\text{(\,counter\,s)};\,\text{stack\,s' = \,v\#stack\,s};\)
  back-edges \(s' = \text{back-edges\,s};\,\text{cross-edges\,s' = cross-edges\,s};\)
  tree-edges \(s' \supseteq \text{tree-edges\,s};\)
  state.more \(s' = \text{state.more\,s}\]
  \[\implies\] on-new-root param \(v\) \(s'\) \(\leq_n\)
  SPEC (\(\lambda x. DFS\\text{-invar} G\) param \(s'\)[state.more := \(x\)])
  \[\implies\] I (\(s'\)[state.more := \(x\)])
assumes finish: \(\forall s \ s' \ u.\)
\[\text{DFS-invar } G \ \text{param } s; \ I \ s; \ \text{cond } s; \ \neg \text{is-break param } s;\]
\(\\text{stack } s \neq []; \ u = \text{hd} (\text{stack } s);\)
\(\text{pending } s = \{ u \} = \{};\)
\(s' = \text{finish } u \ s\]
\(\implies \text{on-finish } \text{param } u \ s' \leq n\]
\(\text{SPEC } (\lambda x. \ \text{DFS-invar } G \ \text{param } (s'\{\text{state} := x\}))\]
\(\implies I (s'\{\text{state} := x\}))\]

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

assumes back-edge: \(\forall s \ s' \ u \ v.\)
\[\text{DFS-invar } G \ \text{param } s; \ I \ s; \ \text{cond } s; \ \neg \text{is-break param } s;\]
\(\text{stack } s \neq []; \ (u, v) \in \text{pending } s; \ u = \text{hd} (\text{stack } s);\)
\(v \in \text{dom} (\text{discovered } s); \ v \in \text{dom} (\text{finished } s);\)
\(s' = \text{back-edge } u \ v \ (s\{\text{pending} := \text{pending } s - \{(u, v)\})\]
\(\implies \text{on-back-edge } \text{param } u \ v \ s' \leq n\]
\(\text{SPEC } (\lambda x. \ \text{DFS-invar } G \ \text{param } (s'\{\text{state} := x\}))\]
\(\implies I (s'\{\text{state} := x\}))\]

shows is-invar I

proof (induct rule: establish-invarI)

case (new-root s) thus ?case by (auto intro!: new-discover)

next

case (discover s s' u v) hence

on-new-root param v s' \leq n
\(\text{SPEC } (\lambda x. \ \text{DFS-invar } G \ \text{param } (s'\{\text{state} := x\}))\]
\(\implies I (s'\{\text{state} := x\}))\]

by (auto intro!: new-discover)

with prereq show ?case by simp

qed fact+

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

— Establish a DFS invariant (cross and back edge cases are combined).

assumes prereq: \(\forall u \ v \ s. \ \text{on-back-edge } \text{param } u \ v \ s = \text{on-cross-edge } \text{param } u \ v \ s\)

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

assumes new-root: \(\forall s \ s' \ v0.\)
\[\text{DFS-invar } G \ \text{param } s; \ I \ s; \ \text{cond } s; \ \neg \text{is-break param } s;\]
\(\text{stack } s = []; \ v0 \in V0; \ v0 \notin \text{dom} (\text{discovered } s);\)
\(s' = \text{new-root } v0 \ s\]
\(\implies \text{on-new-root } \text{param } v0 \ s' \leq_n \text{SPEC } (\lambda x. \ \text{DFS-invar } G \ \text{param } (s'\{\text{state} := x\}))\]
\(\implies I (s'\{\text{state} := x\}))\]
assumes finish: \( \bigwedge s s' u \).
\[ \text{ DFS-invar } G \text{ param } s; I s; \text{ cond } s; \neg \text{ break } \text{ param } s; \]
\( \text{ stack } s \neq [] \); \( u = \text{ hd} (\text{ stack } s) \);
\( \text{ pending } s' \{ u \} = \{ \} \);
\( s' = \text{ finish } u s' \)
\( \implies \text{ on-finish } \text{ param } u s' \leq_n \)
\[ \text{ SPEC (} \lambda x. \text{ DFS-invar } G \text{ param } (s'[\text{ state.more } := x]) \)
\( \implies I (s'[\text{ state.more } := x]) \)
assumes cross-back-edge: \( \bigwedge s s' u v \).
\[ \text{ DFS-invar } G \text{ param } s; I s; \text{ cond } s; \neg \text{ break } \text{ param } s; \]
\( \text{ stack } s \neq [] \); \( (u, v) \in \text{ pending } s; u = \text{ hd} (\text{ stack } s) \);
\( v \in \text{ dom} (\text{ discovered } s) \);
\( \text{ discovered } s' = \text{ discovered } s; \text{ finished } s' = \text{ finished } s; \)
\( \text{ stack } s' = \text{ stack } s; \text{ tree-edges } s' = \text{ tree-edges } s; \text{ counter } s' = \text{ counter } s; \)
\( \text{ pending } s' = \text{ pending } s - \{(u,v)\}; \)
\( \text{ cross-edges } s' \cup \text{ back-edges } s' = \text{ cross-edges } s \cup \text{ back-edges } s \cup \{(u,v)\}; \)
\( \text{ state.more } s' = \text{ state.more } s \)
\( \implies \text{ on-cross-edge } \text{ param } u v s' \leq_n \)
\[ \text{ SPEC (} \lambda x. \text{ DFS-invar } G \text{ param } (s'[\text{ state.more } := x]) \)
\( \implies I (s'[\text{ state.more } := x]) \)
assumes discover: \( \bigwedge s s' u v \).
\[ \text{ DFS-invar } G \text{ param } s; I s; \text{ cond } s; \neg \text{ break } \text{ param } s; \]
\( \text{ stack } s \neq [] \); \( (u, v) \in \text{ pending } s; u = \text{ hd} (\text{ stack } s) \);
\( v \not\in \text{ dom} (\text{ discovered } s) \);
\( s' = \text{ discover } u v (s[\text{ pending } := \text{ pending } s - \{(u,v)\}]) \)
\( \implies \text{ on-discover } \text{ param } u v s' \leq_n \)
\[ \text{ SPEC (} \lambda x. \text{ DFS-invar } G \text{ param } (s'[\text{ state.more } := x]) \)
\( \implies I (s'[\text{ state.more } := x]) \)
shows is-invar I
proof (induct rule: establish-invarI)
next
proof case cross-edge thus ?case by (auto intro!: cross-back-edge)
next
proof case (back-edge s s' u v) hence
\[ \text{ on-cross-edge } \text{ param } u v s' \leq_n \]
\[ \text{ SPEC (} \lambda x. \text{ DFS-invar } G \text{ param } (s'[\text{ state.more } := x]) \)
\( \implies I (s'[\text{ state.more } := x]) \)
by (auto intro!: cross-back-edge)
with prereq intro!: cross-back-edge
qed fact+

lemma establish-invarI-ND-CB [case-names prereq-ND prereq-CB init new-discover finish cross-back-edge]:
— Establish a DFS invariant (new-root/discover and cross/back-edge cases are combined).
assumes prereq:
\( \bigwedge u v s. \text{ on-discover } \text{ param } u v s = \text{ on-new-root } \text{ param } v s \)
\( \bigwedge u v s. \text{ on-back-edge } \text{ param } u v s = \text{ on-cross-edge } \text{ param } u v s \)
assumes init: \( \text{ on-init } \text{ param } \leq_n \text{ SPEC (} \lambda x. I (\text{ empty-state } x) \)
assumes new-discover: \( s \ s' \ v. \)
\[ [\text{DFS-invar} G \ \text{param} s; I \ s; \ \text{cond} s; \ \neg \ \text{is-break} \ \text{param} s; \]
\( v \notin \text{dom} (\text{discovered} s); \)
\( \text{discovered} s' = (\text{discovered} s)(v \mapsto \text{counter} s); \)
\( \text{finished} s' = \text{finished} s; \)
\( \text{counter} s' = \text{Suc} (\text{counter} s); \)
\( \text{stack} s' = v \notin \text{stack} s; \)
\( \text{back-edges} s' = \text{back-edges} s; \)
\( \text{cross-edges} s' = \text{cross-edges} s; \)
\( \text{tree-edges} s' \supseteq \text{tree-edges} s; \)
\( \text{state}.\text{more} s' = \text{state}.\text{more} s \]
\( \implies \) on-new-root param \( v \ s' \leq n \)
\( \text{SPEC} (\lambda x. \ \text{DFS-invar} G \ \text{param} (s'(\text{state}.\text{more} := x))) \)
\( \implies I (s'(\text{state}.\text{more} := x))) \)
assumes finish: \( s \ s' \ u. \)
\[ [\text{DFS-invar} G \ \text{param} s; I \ s; \ \text{cond} s; \ \neg \ \text{is-break} \ \text{param} s; \]
\( \text{stack} s \neq []; \)
\( u = \text{hd} (\text{stack} s); \)
\( \text{pending} s \{ \{ u \} = \}; \)
\( s' = \text{finish} u s \]
\( \implies \) on-finish param \( u \ s' \leq n \)
\( \text{SPEC} (\lambda x. \ \text{DFS-invar} G \ \text{param} (s'(\text{state}.\text{more} := x))) \)
\( \implies I (s'(\text{state}.\text{more} := x))) \)
assumes cross-back-edge: \( s \ s' \ u. \)
\[ [\text{DFS-invar} G \ \text{param} s; I \ s; \ \text{cond} s; \ \neg \ \text{is-break} \ \text{param} s; \]
\( \text{stack} s \neq []; \)
\( (u, v) \in \text{pending} s; \)
\( u = \text{hd} (\text{stack} s); \)
\( v \in \text{dom} (\text{discovered} s); \)
\( \text{discovered} s' = \text{discovered} s; \)
\( \text{finished} s' = \text{finished} s; \)
\( \text{stack} s' = \text{stack} s; \)
\( \text{tree-edges} s' = \text{tree-edges} s; \)
\( \text{counter} s' = \text{counter} s; \)
\( \text{pending} s' = \text{pending} s - \{ \{ u, v \} \}; \)
\( \text{cross-edges} s' \cup \text{back-edges} s' = \text{cross-edges} s \cup \text{back-edges} s \cup \{ \{ u, v \} \}; \)
\( \text{state}.\text{more} s' = \text{state}.\text{more} s \]
\( \implies \) on-cross-edge param \( u \ v \ s' \leq n \)
\( \text{SPEC} (\lambda x. \ \text{DFS-invar} G \ \text{param} (s'(\text{state}.\text{more} := x))) \)
\( \implies I (s'(\text{state}.\text{more} := x))) \)
shows is-invar I
proof (induct rule: establish-invar-I-ND)
case cross-edge thus \( ?\text{case} \) by (auto intro!: cross-back-edge)
next
case (back-edge \( s \ s' \ u \ v) \) hence
on-cross-edge param \( u \ v \ s' \leq n \)
\( \text{SPEC} (\lambda x. \ \text{DFS-invar} G \ \text{param} (s'(\text{state}.\text{more} := x))) \)
\( \implies I (s'(\text{state}.\text{more} := x))) \)
by (auto intro!: cross-back-edge)
with prereq show \( ?\text{case} \) by simp
qed fact+

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

17
\[
[I \; s; \; \text{cond} \; s; \; \text{DFS-invar} \; G \; \text{param} \; s; \; \text{DFS-invar} \; G \; \text{param} \; s'];
stack \; s = []; \; v0 \notin \text{dom} \; (\text{discovered} \; s); \; v0 \in V0;
\]

\[
s' = \text{new-root} \; v0 \; s[\text{state}\cdot\text{more} := e]]
\]

\[
\implies I \; s'
\]

and \text{finish}: \bigwedge s \; s' \; u \; e.
\[
[I \; s; \; \text{cond} \; s; \; \text{DFS-invar} \; G \; \text{param} \; s; \; \text{DFS-invar} \; G \; \text{param} \; s'];
stack \; s \neq []; \; \text{pending} \; s'' \{u\} = \{\};
\]

\[
u = \text{hd} \; (\text{stack} \; s); \; s' = \text{finish} \; u \; s[\text{state}\cdot\text{more} := e]]
\]

\[
\implies I \; s'
\]

and \text{cross-edge}: \bigwedge s \; s' \; u \; v \; e.
\[
[I \; s; \; \text{cond} \; s; \; \text{DFS-invar} \; G \; \text{param} \; s; \; \text{DFS-invar} \; G \; \text{param} \; s'];
stack \; s \neq []; \; v \in \text{pending} \; s'' \{u\}; \; v \in \text{dom} \; (\text{discovered} \; s);
\]

\[
u = \text{hd} \; (\text{stack} \; s);
\]

\[
s' = (\text{cross-edge} \; u \; v \; (s[\text{pending} := \text{pending} \; s - \{(u,v)\}]); \text{state}\cdot\text{more} := e])
\]

\[
\implies I \; s'
\]

and \text{back-edge}: \bigwedge s \; s' \; u \; v \; e.
\[
[I \; s; \; \text{cond} \; s; \; \text{DFS-invar} \; G \; \text{param} \; s; \; \text{DFS-invar} \; G \; \text{param} \; s'];
stack \; s \neq []; \; v \in \text{pending} \; s'' \{u\}; \; v \notin \text{dom} \; (\text{discovered} \; s);
\]

\[
u = \text{hd} \; (\text{stack} \; s);
\]

\[
s' = (\text{back-edge} \; u \; v \; (s[\text{pending} := \text{pending} \; s - \{(u,v)\}]); \text{state}\cdot\text{more} := e])
\]

\[
\implies I \; s'
\]

and \text{discover}: \bigwedge s \; s' \; u \; v \; e.
\[
[I \; s; \; \text{cond} \; s; \; \text{DFS-invar} \; G \; \text{param} \; s; \; \text{DFS-invar} \; G \; \text{param} \; s'];
stack \; s \neq []; \; v \in \text{pending} \; s'' \{u\}; \; v \notin \text{dom} \; (\text{discovered} \; s);
\]

\[
u = \text{hd} \; (\text{stack} \; s);
\]

\[
s' = (\text{discover} \; u \; v \; (s[\text{pending} := \text{pending} \; s - \{(u,v)\}]); \text{state}\cdot\text{more} := e])
\]

\[
\implies I \; s'
\]

\text{shows is-invar} \; I
\text{apply} \; (\text{rule establish-invar} \; I)
\text{apply} \; (\text{blast intro: indep-invar-rules assms})+
done

\text{lemma is-invar} \; I \; [\text{case-names init new-root finish visited discover}]:
— Establish a DFS invariant not taking into account the parameterization, cross/back-edges combined.
\text{assumes init}': \bigwedge e. \; I \; (\text{empty-state} \; e)
\text{and new-root}': \bigwedge s \; s' \; v0 \; e.
\text{and finish}': \bigwedge s \; s' \; u \; e.

18
and visited: \( \bigwedge s \; s' \; u \; v \; e \; c \; b \).

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

and discover: \( \bigwedge s \; s' \; u \; v \; e \).

\[
[ 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'' \{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}\text{more} := e)[]]
\[\Rightarrow I \; s'\]

shows \text{is-invar} I

proof (induct rule: \text{is-invarI-full})

case (cross-edge \( s \; s' \; u \; v \; e \)) thus ?case
apply –

apply (rule visited \( [of \; s \; s' \; v \; u \; \text{insert} \; (u,v) \; (\text{cross-edges} \; s) \; \text{back-edges} \; s \; e] \))

apply clarsimp-all
done
next

case (back-edge \( s \; s' \; u \; v \; e \)) thus ?case
apply –

apply (rule visited \( [of \; s \; s' \; v \; u \; \text{cross-edges} \; s \; \text{insert} \; (u,v) \; (\text{back-edges} \; s) \; e] \))

apply clarsimp-all
done

qed fact+
end

1.1.5 Basic Invariants

We establish some basic invariants

context \text{param-DFS} begin

definition \text{basic-invar} \; s \equiv 
set \; (\text{stack} \; s) = \text{dom} \; (\text{discovered} \; s) - \text{dom} \; (\text{finished} \; s) \land 
distinct \; (\text{stack} \; s) \land 
(\text{stack} \; s \neq []) \rightarrow \text{last} \; (\text{stack} \; s) \in V0 \land 
\text{dom} \; (\text{finished} \; s) \subseteq \text{dom} \; (\text{discovered} \; s) \land
\text{Domain} \; (\text{pending} \; s) \subseteq \text{dom} \; (\text{discovered} \; s) - \text{dom} \; (\text{finished} \; s) \land
\text{pending} \; s \subseteq E

lemma i-basic-invar: \text{is-invar} \; \text{basic-invar}

unfolding \text{basic-invar-def}[\text{abs-def}]

19
apply (induction rule: is-invarI)
apply (clarsimp-all simp: neq-Nil-conv last-tl)
apply blast+
done
end

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

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

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

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

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

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

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

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

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

lemma finished-discovered:
\textbf{lemma} finished-no-pending:
\[ v \in \text{dom} \ (\text{finished } s) \implies \text{pending } s \ "\{v\} = \{\} \]
\textbf{using} basic-invar
\textbf{by} (auto simp add: basic-invar-def)

\textbf{lemma} discovered-eq-finished-un-stack:
\[ \text{dom} \ (\text{discovered } s) = \text{dom} \ (\text{finished } s) \cup \text{set} \ (\text{stack } s) \]
\textbf{using} stack-set-def finished-discovered \textbf{by} auto

\textbf{lemma} pending-on-stack:
\[(v,w) \in \text{pending } s \implies v \in \text{set} \ (\text{stack } s)\]
\textbf{using} basic-invar
\textbf{by} (auto simp add: basic-invar-def)

\textbf{lemma} empty-stack-imp-empty-pending:
\[\text{stack } s = [] \implies \text{pending } s = \{\} \]
\textbf{using} pending-on-stack \textbf{by} auto

end

\texttt{context \textit{param-DFS} \texttt{begin}}

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

\textbf{definition} discovered-closed \( s \equiv \)
\[ E'' \text{dom} \ (\text{finished } s) \subseteq \text{dom} \ (\text{discovered } s) \]
\[ \land \ (E \odot \text{pending } s) " \text{set} (\text{stack } s) \subseteq \text{dom} \ (\text{discovered } s) \]

\textbf{lemma} i-discovered-closed: \text{is-invar} discovered-closed
\textbf{proof} (induct rule: is-invarI)
\textbf{case} (finish \( s \ \ s') \text{\ hence} (E \odot \text{pending } s) " \text{set} (\text{stack } s') \subseteq \text{dom} \ (\text{discovered } s) \)
\textbf{by} (simp add: discovered-closed-def)
\textbf{moreover from} finish \textbf{have} \text{set} (\text{stack } s') \subseteq \text{set} (\text{stack } s) \textbf{by} (auto simp add: neq-Nil-conv cond-def)
ultimately have \((E − \text{pending } s')\) "' set \((\text{stack } s') \subseteq \text{dom } (\text{discovered } s')\)
using finish
by simp blast

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

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

end

context DFS-invar
begin

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

lemma stack-reachable: set \((\text{stack } s) \subseteq \text{reachable}\)
using stack-discovered discovered-reachable by blast

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

lemmas discovered-finite[simp, intro!] = i-discovered-finite [THEN make-invar-thm]

lemma finished-finite[simp, intro!]: finite \((\text{dom } (\text{finished } s))\)
using finished-discovered discovered-finite by (rule finite-subset)

lemma finished-closed:
\(E\)"'dom \((\text{finished } s) \subseteq \text{dom } (\text{discovered } s)\)
using discovered-closed [unfolded discovered-closed-def]
by auto

lemma finished-imp-succ-discovered:
v \in \text{dom } (\text{finished } s) \Rightarrow w \in \text{succ } v \Rightarrow w \in \text{dom } (\text{discovered } s)
using discovered-closed [unfolded discovered-closed-def]
by auto

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

lemma pending-finite[simp, intro!] finite (pending \(s))
proof –
have pending $s \subseteq (\Sigma u : \text{dom} \ (\text{discovered} \ s), E'' \{u\})$
  by (auto dest: pendingD)
also have finite ... 
  apply rule 
  apply (rule discovered-finite) 
  using discovered-reachable 
  by (blast intro: finitely-branching) 
finally (finite-subset) show ?thesis . 
qed

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

lemma nc-finished-eq-reachable:
  assumes NC: $\neg \text{cond} \ s \neg \text{is-break param} \ s$
  shows $\text{dom} \ (\text{finished} \ s) = \text{reachable}$
proof 
  from discovered-reachable show $\text{dom} \ (\text{finished} \ s) \subseteq \text{reachable}$
  by simp
next 
  from discovered-closed have $E''(\text{dom} \ (\text{finished} \ s)) \subseteq \text{dom} \ (\text{finished} \ s)$ 
  unfolding discovered-closed-def by auto 
  with SS show $\text{reachable} \subseteq \text{dom} \ (\text{finished} \ s)$
  by (simp, metis rtrancl-reachable-induct)
qed

lemma nc-V0-finished:
  assumes NC: $\neg \text{cond} \ s \neg \text{is-break param} \ s$
  shows $V0 \subseteq \text{dom} \ (\text{finished} \ s)$
using nc-finished-eq-reachable[OF NC] 
by blast

23
lemma nc-discovered-eq-finished:
  assumes NC: ¬ cond s ¬ is-break param s
  shows dom (discovered s) = dom (finished s)
using finished-discovered
using nc-finished-eq-reachable[OF NC] discovered-reachable
by blast

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

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

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

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

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

from A show ?thesis
unfolding rwof-eq-DFS-invar[symmetric] it-dfs-def
apply −
apply (drule (2) WHILE-nofail-imp-rwof-nofail)

1.1.6 Total Correctness

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

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

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

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

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

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

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

end

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

  show ?thesis
    unfolding it-dfsT-def it-dfs-def

end
apply (rule bind-mono)
apply simp
apply (subst WHLET-eq-WHILE-tproof[
  where I=DFS-invar G param
  and V=param-dfs-variant
])
apply auto []
apply (subst rwof-eq-DFS-invar[ symmetric])
using rwof-init[ OF True[ unfolded it-dfs-def]]
apply (fastforce dest: order-trans) []

apply (rule SPEC-rule-conj1)
apply (rule DFS-invar-step[ OF True], assumption+) []
apply (rule param-dfs-variant-step, (assumption|rule True)+) []

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

1.1.7 Non-Failing Parameterization

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

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

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

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

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

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

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

begin

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

lemma init-leaf-invar: init ≤ₜ SPEC (DFS-invar G param)
  unfolding rwof-eq-DFS-invar[symmetric]
  by (rule rwof-leaf-init)

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

lemma it-dfs-correct: it-dfs ≤ SPEC (λs. DFS-invar G param s ∧ ¬cond s)
  by (simp add: it-dfs-eq-spec)

lemma it-dfs-SPEC:
  assumes ∃ s. [DFS-invar G param s; ¬cond s] → P s
  shows it-dfs ≤ SPEC P
  using weaken-SPEC[OF it-dfs-correct]
  using assms
  by blast

lemma it-dfsT-correct:
  assumes finite reachable
  shows it-dfsT ≤ SPEC (λs. DFS-invar G param s ∧ ¬cond s)
  apply (subst it-dfsT-eq-it-dfs[OF assms])
  by (rule it-dfs-correct)

lemma it-dfsT-SPEC:
  assumes finite reachable
  assumes ∃ s. [DFS-invar G param s; ¬cond s] → P s
  shows it-dfsT ≤ SPEC P
  apply (subst it-dfsT-eq-it-dfs[OF assms(1)])
  using assms(2)
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 \( s \, v \) ≡ \( \text{the} (\text{discovered} \, s \, v) \)
abbreviation the-finished \( s \, v \) ≡ \( \text{the} (\text{finished} \, s \, v) \)

locale timing-syntax
begin

notation \( \text{the-discovered} (\delta) \)
notation \( \text{the-finished} (\varphi) \)

end

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

definition timing-common-inv \( s \) ≡
— \( \delta \, s \, v < \varphi \, s \, v \)
\( (\forall \, v \in \text{dom} \, (\text{finished} \, s)\, . \, \delta \, s \, v < \varphi \, s \, v) \)
— \( v \neq w \rightarrow \delta \, s \, v \neq \delta \, s \, w \land \varphi \, s \, v \neq \varphi \, s \, w \)
— Can’t use \( \text{card dom} = \text{card ran} \) as the maps may be infinite ...
\( \land (\forall \, v \in \text{dom} \, (\text{discovered} \, s)\, . \, \forall \, w \in \text{dom} \, (\text{discovered} \, s)\, . \, v \neq w \rightarrow \delta \, s \, v \neq \delta \, s \, w) \)
\( \land (\forall \, v \in \text{dom} \, (\text{finished} \, s)\, . \, \forall \, w \in \text{dom} \, (\text{finished} \, s)\, . \, v \neq w \rightarrow \varphi \, s \, v \neq \varphi \, s \, w) \)
— \( \delta \, s \, v < \text{counter} \land \varphi \, s \, v < \text{counter} \)
\( \land (\forall \, v \in \text{dom} \, (\text{discovered} \, s)\, . \, \delta \, s \, v < \text{counter} \, s) \)
\( \land (\forall \, v \in \text{dom} \, (\text{finished} \, s)\, . \, \varphi \, s \, v < \text{counter} \, s) \)
\( \land (\forall \, v \in \text{dom} \, (\text{finished} \, s)\, . \, \forall \, w \in \text{succ} \, v\, . \, \delta \, s \, w < \varphi \, s \, v) \)

lemma timing-common-inv:
is-invar timing-common-inv
proof (induction rule: is-invarI)
  case (finish \( s \, s' \)) then interpret DFS-invar where \( s=s \) by simp
from finish have NE: stack s ≠ [] by (simp add: cond-alt)

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

from discovered-closed have
(E - pending s) " {hd (stack s)} ⊆ dom (discovered s)
using hd-in-set[OF NE]
by (auto simp add: discovered-closed-def)

hence succ-hd: pending s " {hd (stack s)} = {}
⇒ succ (hd (stack s)) ⊆ dom (discovered s)
by blast

from finish show ?case
apply (simp add: timing-common-inv-def)
apply (intro conjI)
using * apply simp
using * apply simp
apply (metis less-irrefl)
apply (metis less-irrefl)
apply (metis less-SucI)
apply (metis less-SucI)
apply (blast dest!: succ-hd)
using * apply simp
done

next

case (discover s) then interpret DFS-invar where s=s by simp
from discover show ?case
apply (simp add: timing-common-inv-def)
apply (intro conjI)
using finished-discovered apply fastforce
apply (metis less-irrefl)
apply (metis less-irrefl)
apply (metis less-SucI)
apply (metis less-SucI)
using finished-imp-succ-discovered apply fastforce
done

next

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

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

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

lemma timing-less-counter:
  \( v \in \text{dom} \ (\text{discovered} \ s) \Rightarrow \delta \ s \ v < \text{counter} \ s \)
  \( v \in \text{dom} \ (\text{finished} \ s) \Rightarrow \varphi \ s \ v < \text{counter} \ s \)

using s-timing-common-inv
by (auto simp add: timing-common-inv-def)

lemma disc-lt-fin:
  \( v \in \text{dom} \ (\text{finished} \ s) \Rightarrow \delta \ s \ v < \varphi \ s \ v \)

using s-timing-common-inv
by (auto simp add: timing-common-inv-def)

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

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

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

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

lemma i-prev-stack-discover-all:
is-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)\)
proof (induct rule: is-invarI)
case (finish s) thus \(?case

30
by (cases stack s) auto

next

case (discover s s’ u v)

hence EQ[simp]: discovered s’ = (discovered s)(v ↦→ counter s)

stack s’ = v#stack s

by simp-all

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

from stack-discovered timing-less-counter have

\( \forall w. w \in \text{set} \; (\text{stack} \; s) \implies \delta \; s \; w < \text{counter} \; s \)

by blast

with v-ni have \( \forall w. w \in \text{set} \; (\text{stack} \; s) \implies \delta \; s’ \; w < \delta \; s’ \; v \) by auto

hence \( \forall w. w \in \text{set} \; (\text{drop} \; (\text{Suc} \; 0) \; (\text{stack} \; s’)) \implies \delta \; s’ \; w < \delta \; s’ \; (\text{stack} \; s’ ! 0) \)

by auto

moreover

from v-ni have

\( \forall n. [n < (\text{length} \; (\text{stack} \; s’)); n > 0] \implies \delta \; s’ \; (\text{stack} \; s’ ! n) = \delta \; s \; (\text{stack} \; s’ ! n) \)

by auto

with discover(1) v-ni

have \( \forall n. [n < (\text{length} \; (\text{stack} \; s’)) - 1; n > 0] \implies \forall w \in \text{set} \; (\text{drop} \; (\text{Suc} \; n) \; (\text{stack} \; s’)). \; \delta \; s’ \; (\text{stack} \; s’ ! n) > \delta \; s’ \; w \)

by (auto dest: in-set-dropD)

ultimately show ?case

by (metis drop-Suc-Cons length-drop length-pos-if-in-set length-tl list.sel(3) neq0-conv nth-Cons-0 EQ(2) zero-less-diff)

qed simp-all

end end

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

lemmas prev-stack-discover-all

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

lemmas prev-stack-discover:

\( [n < \text{length} \; (\text{stack} \; s); v \in \text{set} \; (\text{drop} \; (\text{Suc} \; n) \; (\text{stack} \; s))] \implies \delta \; s \; (\text{stack} \; s ! n) > \delta \; s \; v \)

by (metis prev-stack-discover-all)

lemma Suc-stack-discover:

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

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

proof –

from prev-stack-discover assumes have

\( \forall v. v \in \text{set} \; (\text{drop} \; (\text{Suc} \; n) \; (\text{stack} \; s)) \implies \delta \; s \; (\text{stack} \; s ! n) > \delta \; s \; v \)

by fastforce
moreover from \( n \) have \( \text{stack} \ s ! \ Suc \ n \in \set{\text{drop} \ (\Suc \ n) \ (\text{stack} \ s)} \)
using \( \text{in-set-conv-nth} \) by \( \text{fastforce} \)
ultimately show \( \text{thesis} \).
qed

lemma \( \text{tl-lt-stack-hd-discover} \):
assumes \( \text{notempty: stack} \ s \neq [] \)
and \( x \in \set{\text{tl} \ (\text{stack} \ s)} \)
shows \( \delta \ s \ x < \delta \ s \ (\text{hd} \ (\text{stack} \ s)) \)
proof –
from \( \text{notempty} \) obtain \( y \ ys \) where \( \text{stack} \ s = y \# ys \) by \( \text{metis list.exhaust} \)
with \( \text{assms} \) show \( \text{thesis} \)
using \( \text{prev-stack-discover} \)
by \( \text{(cases} \ ys) \) \( \text{force+} \)
qed

lemma \( \text{stack-nth-order} \):
assumes \( l: i < \text{length} \ (\text{stack} \ s) \ j < \text{length} \ (\text{stack} \ s) \)
shows \( \delta \ s \ (\text{stack} \ s ! i) < \delta \ s \ (\text{stack} \ s ! j) \rightleftharpoons i > j \ (\text{is} \ \delta \ s \ ?i < \delta \ s \ ?j \rightleftharpoons -) \)
proof
assume \( \delta: \delta \ s \ ?i < \delta \ s \ ?j \)

from \( l \) \( \text{stack-set-def} \) have
\( \text{disc:} \ ?i \in \text{dom} \ (\text{discovered} \ s) \ ?j \in \text{dom} \ (\text{discovered} \ s) \)
by \( \text{auto} \)
with \( \text{disc-unequal[OF disc]} \) \( \delta \) \( \text{have} \ i \neq j \) by \( \text{auto} \)

moreover
\{ 
assume \( i < j \)
with \( l \) have \( \text{stack} \ s ! \ j \in \set{\text{drop} \ (\Suc \ i) \ (\text{stack} \ s)} \)
using \( \text{in-set-drop-conv-nth[of stack} \ s ! \ Suc \ i \ \text{stack} \ s] \)
by \( \text{fastforce} \)
with \( \text{prev-stack-discover} \ l \) have \( \delta \ s \ (\text{stack} \ s ! j) < \delta \ s \ (\text{stack} \ s ! i) \)
by \( \text{simp} \)
with \( \delta \) have \( \text{False} \) by \( \text{simp} \)
\}
ultimately show \( i > j \) by \( \text{force} \)
next
assume \( i > j \)
with \( l \) have \( \text{stack} \ s ! \ i \in \set{\text{drop} \ (\Suc \ j) \ (\text{stack} \ s)} \)
using \( \text{in-set-drop-conv-nth[of stack} \ s ! \ Suc \ j \ \text{stack} \ s] \)
by \( \text{fastforce} \)
with \( \text{prev-stack-discover} \ l \) show \( \delta \ s \ ?i < \delta \ s \ ?j \) by \( \text{simp} \)
qed
end end
1.2.2 Paranthesis Theorem

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 \land v \in \text{dom} (\text{finished } s) \rightarrow \{ \phi s v < \delta s w \rightarrow \text{disjoint} \land (\phi s v > \delta s w \land w \in \text{dom} (\text{finished } s) \land \phi s w < \phi s v) \}

lemma i-parenthesis: is-invar parenthesis

proof (induct rule: \text{is-invarI})

case (finish s s')

hence $EQ[simp]$: discovered $s' = discovered s$

counter $s' = \text{Suc} (\text{counter } s)$

finished $s' = \text{finished } s(\text{hd (stack } s) \mapsto \text{counter } s)$

by \text{simp-all}

from finish interpret DFS-invar where $s=s$ by simp

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

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

note assms = dom $\delta f$ $EQ$

let $\text{DISJ} = \phi s' x < \delta s' y$

let $\text{IN} = \delta s' y < \phi s' x \land y \in \text{dom} (\text{finished } s') \land \phi s y < \phi s' x$

have $\text{DISJ} \lor \text{IN}$

proof (cases $x = \text{hd (stack } s)$)

\text{case } True note $x$-is-hd = this

\text{hence } $\phi x: \phi s' x = \text{counter } s$ by simp

\text{from } x-is-hd $\text{neg have } y$-not-hd: $y \neq \text{hd (stack } s)$ by simp

\text{have } $\delta s y < \phi s' x \land y \in \text{dom} (\text{finished } s) \land \phi s y < \phi s' x$

\text{proof (cases } y \in \text{set (stack } s)\text{)}

\text{— } y \text{ on stack is not possible: According to}

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

it is discovered after $x (= \text{hd (stack } s)$

\text{case } True \text{ with } y$-not-hd $\text{have } y \in \text{set (tl (stack } s))$

by (cases stack) simp-all

with $\text{tl-tl-stack-hd-discover[OF NE]}$ $\delta$ x-is-hd $\text{have } \delta s y < \delta s x$

by simp

with $\delta$ have $\text{False}$ by simp
thus \( \text{thesis} \).

next

\begin{cases} \text{case False} & \text{— y must be a successor of } x \ (= (\text{hd} \ (\text{stack} \ s))) \\
\text{from } \text{dom} \ \text{have } y \in \text{dom} \ (\text{discovered} \ s) \ \text{by simp} \\
\text{with } \text{False discovered-not-stack-imp-finished } \text{have } *: \\
y \in \text{dom} \ (\text{finished} \ s) \\
\text{by simp} \\
\text{moreover with } \text{timing-less-counter } \varphi x \ \text{have } \varphi \ s \ y < \varphi \ s' \ x \ \text{by simp} \\
\text{moreover with } * \ \text{disc-ll-fin } \varphi x \ \text{have } \delta \ s \ y < \varphi \ s' \ x \\
\text{by } (\text{metis less-trans}) \\
\text{ultimately show } \text{thesis by simp} \\
\text{qed} \\
\text{with } y\text{-not-hd show } \text{thesis by simp} \\
\end{cases}

next

\begin{cases} \text{case False} & \text{note } [\text{simp}] = \text{this} \\
\text{show } \text{thesis} \\
\text{proof } (\text{cases } y = \text{hd} \ (\text{stack} \ s)) \\
\text{case False} & \text{with } \text{finish assms show } \text{thesis} \\
\text{by } (\text{simp add: parenthesis-def}) \\
\end{cases}

next

\begin{cases} \text{case True} & \text{with } \text{stack-not-finished } \text{have } y \notin \text{dom} \ (\text{finished} \ s) \\
\text{using } \text{hd-in-set}[\text{OF NE}] \\
\text{by auto} \\
\text{with } \text{finish assms have } \varphi \ s \ x < \delta \ s \ y \\
\text{unfolding } \text{parenthesis-def} \\
\text{by auto} \\
\text{hence } \text{DISJ by simp} \\
\text{thus } \text{thesis \ ..} \\
\text{qed} \\
\text{qed} \\
\end{cases}

thus \( \text{case} \ \text{by } (\text{simp add: parenthesis-def}) \)

next

\begin{cases} \text{case } (\text{discover } s \ s' \ u \ v) \\
\text{hence } \text{EQ[simp]}: \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{by simp-all} \\
\end{cases}

from \text{discover interpret } \text{DFS-invar where } s=\text{s by simp} 

from \text{discover finished-discovered have} 

\begin{cases} \text{V'}: v \notin \text{dom} \ (\text{discovered} \ s) \ v \notin \text{dom} \ (\text{finished} \ s) \\
\text{by auto} \\
\end{cases}

\{ 
\text{fix } x \ y 
\text{assume } \text{dom} : x \in \text{dom} \ (\text{discovered} \ s') \ y \in \text{dom} \ (\text{discovered} \ s') \\
\text{and } \delta: \delta \ s' x < \delta \ s' y \\
\text{and } f: x \in \text{dom} \ (\text{finished} \ s') 
\}
let \(?\text{DISJ}\) = \(\varphi \ s' \ x < \delta \ s' \ y\)
let \(?\text{IN}\) = \(\delta \ s' \ y < \varphi \ s' \ x \wedge y \in \text{dom} \ (\text{finished} \ s') \wedge \varphi \ s' \ y < \varphi \ s' \ x\)

from \(\text{dom} \ V' \ f\) have \(x: x \in \text{dom} \ (\text{discovered} \ s) x \neq v\) by auto

have \(?\text{DISJ} \lor ?\text{IN}\)
proof (cases \(y = v\))
  case True hence \(\delta \ s' \ y = \text{counter} \ s\) by simp
  moreover from \(\text{timing-less-counter} \ x \ f\) have \(\varphi \ s' \ x < \text{counter} \ s\) by auto
  ultimately have \(?\text{DISJ}\) by simp
  thus \(?\text{thesis}..\)
next
  case False with \(\text{dom}\) have \(y \in \text{dom} \ (\text{discovered} \ s)\) by simp
  with \(\text{discover} \ False\ \delta \ f \ x\) show \(?\text{thesis}\) by (simp add: parenthesis-def)
qed

thus \(?\text{case by} \ (\text{simp add: parenthesis-def})\)
next
  case \(\text{new-root} \ s \ s' \ v0\)
then interpret \(\text{DFS-invar}\) where \(s=s\) by simp

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

context \(\text{DFS-invar}\) begin context begin interpretation \(\text{timing-syntax}\).

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 \(\varphi \ s \ v < \delta \ s \ w\) — disjoint
  \(\lor \ (\varphi \ s \ v > \delta \ s \ w \wedge w \in \text{dom} \ (\text{finished} \ s) \wedge \varphi \ s \ w < \varphi \ s \ v)\)
using assms
using i-parenthesis[THEN make-invar-thm]
using finished-discovered
unfolding parenthesis-def
by blast

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

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 $\varphi s v < \delta s w$
using parenthesis assms
by force

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 $w \in \text{dom} (\text{finished } s) \land \varphi s w < \varphi s v$
using finished-succ-fin finished-imp-succ-discovered parenthesis-contained
using assms
by metis
end end

1.2.3 Edge Types

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

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

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

lemma edgesE-CB:
assumes $x \in \text{edges } s$
and $x \in \text{tree-edges } s \implies P$
and $x \in \text{cross-edges } s \cup \text{back-edges } s \implies P$
shows $P$
using assms by auto

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

lemma i-edges-basic:
is-invar edges-basic
unfolding edges-basic-def[abs-def]
proof (induct rule: is-invarI-full)
case (back-edge $s$)
then interpret DFS-invar where $s=s$ by simp
from back-edge show ?case by (auto dest: pendingD)
next
  case (cross-edge s)
  then interpret DFS-invar where $s = s$ by simp
from cross-edge show ?case by (auto dest: pendingD)
next
  case (discover s)
  then interpret DFS-invar where $s = s$ by simp
from discover show ?case
  apply (simp add: Field-def Range-def Domain-def)
  apply (drule pendingD)
  apply simp
  by (blast)
next
  case (new-root s)
  thus ?case by (simp add: Field-def) blast
qed auto

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

lemma i-edges-covered:
  is-invar $(\lambda s. (E \cap \text{dom} (\text{discovered } s) \times \text{UNIV}) - \text{pending } s = \text{edges } s)$
proof (induction rule: is-invarI-full)
  case (new-root $s' v0$)
  interpret DFS-invar $G$ param $s$ by fact
  from new-root empty-stack-imp-empty-pending
  have [simp]: pending $s = \{\}$ by simp
  from $v0 \notin \text{dom} (\text{discovered } s)$:
  have [simp]: $E \cap \text{insert } v0 (\text{dom} (\text{discovered } s)) \times \text{UNIV} - \{v0\} \times \text{succ } v0$
  
  $= E \cap \text{dom} (\text{discovered } s) \times \text{UNIV}$ by auto

from new-root show ?case by simp
next
  case (cross-edge $s s' u v$)
  interpret DFS-invar $G$ param $s$ by fact
  from cross-edge stack-discovered have $u \in \text{dom} (\text{discovered } s)$
  by (cases stack $s$) auto
  with cross-edge(2-) pending-ssE have
  $E \cap \text{dom} (\text{discovered } s) \times \text{UNIV} - (\text{pending } s - \{(\text{hd} (\text{stack } s), v)\})$
  $= \text{insert } (\text{hd} (\text{stack } s), v) (E \cap \text{dom} (\text{discovered } s) \times \text{UNIV} - \text{pending } s)$
  by auto
  thus ?case using cross-edge by simp
next
  case (back-edge $s s' u v$)

37
interpret DFS-invar G param s by fact

from back-edge stack-discovered have \( u \in \text{dom} \) (discovered s)
by (cases stack s) auto

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

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

from discover stack-discovered have \( u \in \text{dom} \) (discovered s)
by (cases stack s) auto

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

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

context DFS-invar begin

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

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

lemma nc-edges-covered:
assumes \( \neg \text{cond} \) s \( \neg\text{is-break} \) param s
shows E \( \cap \) reachable \( \times \) UNIV = \( \text{edges} \) s
proof -
from assms have [simp]: \( \text{stack} \) s = []
unfolding cond-def by (auto simp: pred-defs)

hence [simp]: \( \text{pending} \) s = {} by (rule empty-stack-imp-empty-pending)

from edges-covered nc-discovered-eq-reachable[OF assms]
show ?thesis by simp
lemma tree-edges-ssE: tree-edges $s \subseteq E$ and
        tree-edges-not-pending: tree-edges $s \subseteq -\text{pending }s$ and
        tree-edge-is-succ: $(v,w) \in \text{tree-edges }s \Rightarrow w \in \text{succ }v$ and
        tree-edges-discovered: Field(\text{tree-edges }s) \subseteq \text{dom }\text{(discovered }s)$ and

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

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

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

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

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

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

lemma edge-imp-discovered:
        $(v,w) \in \text{edges }s \Rightarrow v \in \text{dom }\text{(discovered }s)$
(v, w) ∈ edges s → w ∈ dom (discovered s)
using tree-edge-imp-discovered cross-edge-imp-discovered back-edge-imp-discovered
by blast+

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

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

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

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

end

Properties of the DFS Tree

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

lemma tree-edge-disc-lt-fin:
(v, w) ∈ tree-edges s → v ∈ dom (finished s) → δ s w < ϕ s v
by (metis finished-succ-fin tree-edge-is-succ)

lemma back-edge-disc-lt-fin:
(v, w) ∈ back-edges s → v ∈ dom (finished s) → δ s w < ϕ s v
by (metis finished-succ-fin back-edge-is-succ)

lemma cross-edge-disc-lt-fin:
(v, w) ∈ cross-edges s → v ∈ dom (finished s) → δ s w < ϕ s v
by (metis finished-succ-fin cross-edge-is-succ)
end

context param-DFS begin

lemma i-stack-is-tree-path:
is-invar (λs. stack s ≠ [] → (∃v0 ∈ V0. path (tree-edges s) v0 (rev (tl (stack s))) (hd (stack s))))
proof (induct rule: is-invarI)
case (discover s s' u v)
hence EQ[simp]: stack s' = v ≠ stack s
  tree-edges s' = insert (hd (stack s), v) (tree-edges s)
  by simp-all
from discover have NE[simp]: stack s ≠ [] by simp

from discover obtain v0 where
\[v_0 \in V_0\]

\[
\text{path } (\text{tree-edges } s) \ v_0 \ (\text{rev } (\text{tl } (stack \ s))) \ (hd \ (stack \ s)) \]

by blast

with \text{path-mono}[OF - this(2)] EQ have

\[
\text{path } (\text{tree-edges } s') \ v_0 \ (\text{rev } (\text{tl } (stack \ s))) \ (hd \ (stack \ s))
\]

by blast

with \((v_0 \in V_0)\) show ?case

by (cases stack \ s) \ (auto simp: path-simps)

next

\begin{itemize}
\item case \((\text{finish } s \ s')\)
\item hence EQ[simp]: \text{stack } s' = \text{tl } (stack \ s)
  \item tree-edges \ s' = \text{tree-edges } \ s
  by simp-all
\end{itemize}

from \text{finish} obtain \(v_0\) where

\[
\text{path } (\text{tree-edges } s) \ v_0 \ (\text{rev } (\text{tl } (stack \ s))) \ (hd \ (stack \ s))
\]

by blast

hence \(P: \text{path } (\text{tree-edges } s') \ v_0 \ (\text{rev } (\text{stack } s')) \ (hd \ (stack \ s))\) by simp

show ?case

proof

assume \(A: \text{stack } s' \neq []\)

with \(P\) have \((hd \ (stack \ s'), \ hd \ (stack \ s)) \in \text{tree-edges } \ s'\)

by (auto simp: neg-nil-conv path-simps)

moreover from \(P \ A\) have

\[
\text{path } (\text{tree-edges } s') \ v_0 \ (\text{rev } (\text{tl } (stack \ s'))) \ @ \ (hd \ (stack \ s')) \ (hd \ (stack \ s))
\]

by (simp)

moreover note \((v_0 \in V_0)\)

ultimately show \(\exists v_0 \in V_0. \text{path } (\text{tree-edges } s') \ v_0 \ (\text{rev } (\text{tl } (stack \ s'))) \ (hd \ (stack \ s'))\)

by (auto simp add: path-append-conv)

qed

qed simp-all

end

context DFS-invar begin

lemmas stack-is-tree-path =

\text{i-stack-is-tree-path}[THEN make-invar-thm, rule-format]

lemma stack-is-path:

\[
stack \ s \neq [] \implies \exists \ v_0 \in V_0. \text{path } E \ v_0 \ (\text{rev } (\text{tl } (stack \ s))) \ (hd \ (stack \ s))
\]

using \text{stack-is-tree-path path-mono}[OF \text{tree-edges-ssE}]

by blast

lemma hd-succ-stack-is-path:

assumes \(ne: \text{stack } s \neq []\)
and \( \text{succ}: v \in \text{succ} (\text{hd} (\text{stack} s)) \)

shows \( \exists v_0 \in V_0. \text{path} E v_0 \) (rev (stack s)) v

proof −

from stack-is-path[OF ne] \( \text{succ} \) obtain \( v_0 \) where

\( v_0 \in V_0 \)

path E v_0 (rev (tl (stack s))) @ [hd (stack s)]) v

by (auto simp add: path-append-cone)

thus ?thesis using ne

by (cases stack s) auto

qed

lemma tl-stack-hd-tree-path:

assumes \( \text{stack} s \neq [] \)

and \( v \in \text{set} (\text{tl} (\text{stack} s)) \)

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

proof −

from stack-is-tree-path assms obtain \( v_0 \) where

path (tree-edges s) v_0 (rev (tl (stack s))) (hd (stack s))

by auto

from assms path-member-reach-end[OF this] show ?thesis by simp

qed

class param-DFS begin

definition tree-discovered-inv s ≡

\[
\begin{align*}
& (\text{tree-edges} s = \{\} \longrightarrow \text{dom (discovered} s) \subseteq V_0 \land (\text{stack} s = \[] \land (\exists v_0 \in V_0. \text{stack} s = [v_0]))) \\
& \land (\text{tree-edges} s \neq \{\} \longrightarrow (\text{tree-edges} s)^+ \land \text{V}_0 \cup \text{V}_0 = \text{dom (discovered} s)) \cup \text{V}_0)
\end{align*}
\]

lemma i-tree-discovered-inv:

is-invar tree-discovered-inv

proof (induct rule: is-invarI)

case (discover s s' u v)

hence EQ[simp]: stack s' = v ≠ stack s

tree-edges s' = insert (hd (stack s), v) (\text{tree-edges} s)

discovered s' = (\text{discovered} s)(v \mapsto \text{counter} s)

by simp-all

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

from discover have NE[simp]: stack s ≠ [] by simp

note TDI = (tree-discovered-inv s)[unfolded tree-discovered-inv-def]

have tree-edges s' = \{\} \longrightarrow \text{dom (discovered} s') \subseteq V_0 \land (\text{stack} s' = \[] \lor (\exists v_0 \in V_0. \text{stack} s' = [v_0]))

by simp — tree-edges s' ≠ \{\}

end
moreover 

\begin{verbatim}
fix x
assume A: x ∈ (tree-edges s')⁺ "\(V0 \cup V0 x \notin V0\)
then obtain y where y: (y,x) ∈ (tree-edges s')⁺ y ∈ V0 by auto
\end{verbatim}

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

show ?thesis
proof (cases x = v)
case True thus ?thesis by simp
next
case False with y have (y,x) ∈ (tree-edges s')⁺
proof (induct rule: trancl-induct)
case (step a b) hence (a,b) ∈ tree-edges s by simp
  with tree-edge-imp-discovered have a ∈ dom (discovered s) by simp
  with discover have a ≠ v by blast
  with step show ?case by auto
qed simp
with y ∈ V0; have x ∈ (tree-edges s')⁺ "\(V0\) by auto
with t-ne TDI show ?thesis by auto
qed
qed

\end{verbatim}
with TDI t-ne \( x \not\in V0 \) have \( x \in (\text{tree-edges } s)^+ \) \( V0 \) by auto
with trancl-sub-insert-trancl show \( \text{thesis} \) by simp blast
next
case True
from t-ne TDI have \( \text{dom } (\text{discovered } s) \cup V0 = (\text{tree-edges } s)^+ \) \( V0 \cup V0 \) by simp
moreover from stack-is-tree-path[OF NE] obtain \( v0 \) where \( v0 \in V0 \) and
(\( v0, \text{hd } (\text{stack } s) \) \( \in (\text{tree-edges } s)^+ \) by (blast intro: path-is-rtrancl)
with EQ have \( (v0, \text{hd } (\text{stack } s)) \in (\text{tree-edges } s')^{\ast} \) by (auto intro: rtrancl-mono-mp)
ultimately show \( \text{thesis} \) using \( \langle v0 \in V0 \rangle \) True by (auto elim: rtrancl-into-trancl1)
qed
qed
\}
with t-d have \( (\text{tree-edges } s')^+ \) \( V0 \cup V0 = \text{dom } (\text{discovered } s') \cup V0 \) by blast
ultimately show \( \text{?case} \) by (simp add: tree-discovered-inv-def)
qed (auto simp add: tree-discovered-inv-def)

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

lemma (in DFS-invar) discovered-iff-tree-path:
\( v \not\in V0 \implies v \in \text{dom } (\text{discovered } s) \iff (\exists v0 \in V0. (v0, v) \in (\text{tree-edges } s)^+) \)
using tree-discovered-inv
by (auto simp add: tree-discovered-inv-def)

lemma i-tree-one-predecessor:
is-invar \( (\lambda s. \forall (v, v') \in \text{tree-edges } s. \forall y. y \neq v \implies (y, v') \not\in \text{tree-edges } s) \)
proof (induct rule: is-invarI)
case (discover s s' u v)
hence EQ[simp]: \( \text{tree-edges } s' = \text{insert } (\text{hd } (\text{stack } s), v) \) \( (\text{tree-edges } s) \) by simp
from discover interpret DFS-invar where \( s = s \) by simp
from discover have NE[simp]: \( s \neq \[] \) by (simp add: cond-alt)

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

from discover stack-discovered have \( v:\text{-hd: } \text{hd } (\text{stack } s) \neq v \)
using hd-in-set[OF NE] by blast
from discover tree-edges-discovered have \( v:\text{-notin-tree: } \forall (x, x') \in \text{tree-edges } s. x \neq v \land x' \neq v \)

44
by (blast intro!: Field-not-elem)

have \((y,w') \notin \text{tree-edges } s'\)
proof (cases \(w = \text{hd } (\text{stack } s)\))
  case True
  have \((y,v) \notin \text{tree-edges } s'\)
  proof (rule notI)
    assume \((y,v) \in \text{tree-edges } s'\)
    with True \((y \neq w)\) have \((y,v) \in \text{tree-edges } s\) by simp
    with v-notin-tree show \(\text{False}\) by auto
  qed

  with True \((y \neq w)\) v-hd show \(?\text{thesis}\)
  apply (cases \(w = v\))
  apply simp
  using discover apply simp apply blast
  done

next
  case False with v-notin-tree \((y \neq w)\) v-hd
  show \(?\text{thesis}\)
  apply (cases \(w' = v\))
  apply simp apply blast
  using discover apply simp apply blast
  done

  qed

} thus \(?\text{case}\) by blast

qed simp-all

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

lemma \((\text{in DFS-invar})\) \(\text{tree-eq-rule}\):
\[ ((v,w) \in \text{tree-edges } s; (u,w) \in \text{tree-edges } s) \implies v = u \]
using tree-one-predecessor
by blast

context begin interpretation \textit{timing-syntax}.

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

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

from discover tree-edges-discovered have
$v\text{-notin-tree}: \forall (x, x') \in \text{tree-edges } s. x \neq v \land x' \neq v$
by (blast intro: Field-not-elem)

from discover stack-discovered have
$v\text{-hd}: \text{hd (stack } s) \neq v$
using $\text{hd-in-set}[\text{OF } \text{NE}]$
by blast

{ 
fix $a \hspace{1em} b$
assume $T; (a, b) \in \text{tree-edges } s'$
have $\delta \ s' a \prec \delta \ s' b$
proof (cases $b = v$)
case True with $T$ v\text{-notin-tree} have [simp]: $a = \text{hd (stack } s)$ by auto
with stack-discovered have $a \in \text{dom (discovered } s)$
by (metis hd-in-set NE set-mp)
with v\text{-hd True timing-less-counter show }?\text{thesis by simp}
next
case False with v\text{-notin-tree }T$ have ($a, b) \in \text{tree-edges } s a \neq v$ by auto
with discover have $\delta \ s a < \delta \ s b$ by auto
with False $\langle a \neq v\rangle$ show $\text{thesis by simp}$
qed
} thus $\text{?case by blast}

next
case $\langle \text{new-root } s \hspace{1em} s' \hspace{1em} v0\rangle$ then interpret DFS-invar where $s\:=\:s$ by simp
from new-root have $\text{tree-edges } s' = \text{tree-edges } s$ by simp
moreover from tree-edge-imp-discovered new-root have $\forall (v, v') \in \text{tree-edges } s. v \neq v0 \land v' \neq v0$ by blast
ultimately show $\text{?case using } \text{new-root by auto}$
qed simp-all
end end

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

lemma tree-edge-disc:
$(v, w) \in \text{tree-edges } s \implies \delta \ s v < \delta \ s w$
using $\text{i-tree-edge-disc[THEN make-invar-thm]}$
by blast

lemma tree-path-disc:
$(v, w) \in (\text{tree-edges } s)^+ \implies \delta \ s v < \delta \ s w$
by (auto elim!: trancl-induct dest: tree-edge-disc)

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

\textbf{lemma} tree-acyclic:
\begin{itemize}
\item \texttt{acyclic (tree-edges s)}
\item \texttt{by (metis acyclicI no-loop-in-tree)}
\end{itemize}

\textbf{lemma} no-self-loop-in-tree:
\begin{itemize}
\item \texttt{(v,v) \notin tree-edges s}
\item \texttt{using tree-edge-disc by auto}
\end{itemize}

\textbf{lemma} tree-edge-unequal:
\begin{itemize}
\item \texttt{(v,w) \in tree-edges s = \Rightarrow v \neq w}
\item \texttt{by (metis no-self-loop-in-tree)}
\end{itemize}

\textbf{lemma} tree-path-unequal:
\begin{itemize}
\item \texttt{(v,w) \in (tree-edges s)+ = \Rightarrow v \neq w}
\item \texttt{by (metis no-loop-in-tree)}
\end{itemize}

\textbf{lemma} tree-subpath’:
\begin{itemize}
\item assumes \texttt{x: (x,v) \in (tree-edges s)+}
\item and \texttt{y: (y,v) \in (tree-edges s)+}
\item and \texttt{x \neq y}
\item shows \texttt{(x,y) \in (tree-edges s)+ \lor (y,x) \in (tree-edges s)+}
\end{itemize}

\textbf{proof —}
\begin{itemize}
\item from \texttt{x obtain px where px: path (tree-edges s) x px v and px \neq []}
\item using \texttt{trancl-is-path by metis}
\item from \texttt{y obtain py where py: path (tree-edges s) y py v and py \neq []}
\item using \texttt{trancl-is-path by metis}
\item from \texttt{\langle px \neq [] \rangle \langle py \neq [] \rangle px py}
\item show ?thesis
\end{itemize}

\textbf{proof (induction arbitrary: v rule: rev-nonempty-induct2’)}
\begin{itemize}
\item case \texttt{(single)} hence \texttt{(x,v) \in tree-edges s (y,v) \in tree-edges s}
\item by \texttt{(simp-all add: path-simps)}
\item with \texttt{tree-eq-rule have x=y by simp}
\item with \texttt{(x\neq y) show ?case by contradiction}
\end{itemize}

\textbf{next}
\begin{itemize}
\item case \texttt{(snoc a as) hence (y,v) \in tree-edges s by (simp add: path-simps)}
\item moreover from \texttt{snoc have path (tree-edges s) x as a (a,v) \in tree-edges s}
\item by \texttt{(simp-all add: path-simps)}
\item ultimately have \texttt{path (tree-edges s) x as y}
\item using \texttt{tree-eq-rule}
\item by \texttt{auto}
\item with \texttt{path-is-trancl (as \neq []): show ?case by metis}
\end{itemize}

\textbf{next}
\begin{itemize}
\item case \texttt{(snocr - a as) hence (x,v) \in tree-edges s by (simp add: path-simps)}
\item moreover from \texttt{snocr have path (tree-edges s) y as a (a,v) \in tree-edges s}
\item by \texttt{(simp-all add: path-simps)}
\item ultimately have \texttt{path (tree-edges s) y as x}
\end{itemize}
using tree-eq-rule by auto
with path-is-trancl (as ≠ []) show ?case by metis
next
case (snoclr a as b bs)
hence
path (tree-edges s) x as a (a,v) ∈ tree-edges s
path (tree-edges s) y bs b (b,v) ∈ tree-edges s
by (simp-all add: path-simps)
moreover hence a=b using tree-eq-rule by simp
ultimately show ?thesis using snoclr.IH by metis
qed

lemma tree-subpath:
assumes (x,v) ∈ (tree-edges s)⁺
and (y,v) ∈ (tree-edges s)⁺
and δ: δ s x < δ s y
shows (x,y) ∈ (tree-edges s)⁺
proof –
from δ have x ≠ y by auto
with assms tree-subpath’ have (x,y) ∈ (tree-edges s)⁺ ∨ (y,x) ∈ (tree-edges s)⁺ by simp
moreover from δ tree-path-disc have (y,x) /∈ (tree-edges s)⁺ by force
ultimately show ?thesis by simp
qed

lemma on-stack-is-tree-path:
assumes x: x ∈ set (stack s)
and y: y ∈ set (stack s)
and δ: δ s x < δ s y
shows (x,y) ∈ (tree-edges s)⁺
proof –
from x obtain i where i: stack s ! i = x i < length (stack s)
by (metis in-set-conv-nth)
from y obtain j where j: stack s ! j = y j < length (stack s)
by (metis in-set-conv-nth)
with i δ stack-nth-order have j < i by force
from x have ne[simp]: stack s ≠ [] by auto
from j<i have x ∈ set (tl (stack s))
using nth-mem nth-tl[OF ne, of i - 1] i
by auto
with il-stack-hd-tree-path have
x-path: (x, hd (stack s)) ∈ (tree-edges s)⁺
by simp

48
then show thesis
proof (cases j=0)
case True with j have hd (stack s) = y by (metis hd-conv-nth ne)
with x-path show thesis by simp
next
case False hence y ∈ set (tl (stack s))
  using nth-mem nth-tl[OF ne, of j - 1] j
  by auto
with tl-stack-hd-tree-path have (y, hd (stack s)) ∈ (tree-edges s)⁺
  by simp
with x-path δ show thesis
  using tree-subpath
  by metis
qed

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

lemma tree-edge-impl-parenthesis:
  assumes \( \delta \) \((v,w) \in \text{tree-edges} \ s \)
  and \( f : v \in \text{dom} (\text{finished} \ s) \)
  shows \( w \in \text{dom} (\text{finished} \ s) \)
  \( \land \delta s v < \delta s w \)
  \( \land \varphi s w < \varphi s v \)
proof -
  from tree-edge-disc-lt-fin assms have \( \delta s w < \varphi s v \) by simp
  with f tree-edge-imp-discovered[\( OF \ t \)] tree-edge-disc[\( OF \ t \)]
  show thesis
    using parenthesis-contained
    by metis
qed
lemma tree-path-impl-parenthesis:
assumes \((v, w) \in (\text{tree-edges } s)^+\)
and \(v \in \text{dom } (\text{finished } s)\)
shows \(w \in \text{dom } (\text{finished } s)\)
\(\land \delta s v < \delta s w\)
\(\land \varphi s w < \varphi s v\)
using assms
by (auto elim!: trancl-induct dest: tree-edge-impl-parenthesis)

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

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

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

lemma paren-imp-tree-reach:
is-invar paren-imp-tree-reach
unfolding paren-imp-tree-reach-def[abs-def]
proof (induct rule: is-invarI)
case (discover s s' u v)
  hence EQ[simp]: tree-edges s' = insert (hd (stack s), v) (tree-edges s)
  finished s' = finished s
  discovered s' = (discovered s)(v \mapsto \text{counter } s)
  by simp-all
from discover interpret DFS-invar where s=s by simp
from discover have NE[simp]: stack s \neq [] by (simp add: cond-alt)
show \(?case\)
proof (intro ballI impI)
fix a b
assume F: a ∈ dom (discovered s') b ∈ dom (finished s')
  and D: δ s' a < δ s' b ∧ (a ∉ dom (finished s') ∨ ϕ s' a > ϕ s' b)

from F finished-discovered discover have b ≠ v by auto
show (a,b) ∈ (tree-edges s')+
proof (cases a = v)
case True with D have counter s < δ s b by simp
also from F have b ∈ dom (discovered s)
  using finished-discovered by auto
finally have False .
thus ?thesis ..
next
case False with F D discover have (a,b) ∈ (tree-edges s)⁺ by simp
thus ?thesis by (auto intro: trancl-mono-mp)
qed

next
case (finish s s' u)
hence EQ[simp]: tree-edges s' = tree-edges s
  finished s' = (finished s)(hd (stack s) ↦→ counter s)
  discovered s' = discovered s
  stack s' = tl (stack s)
by simp-all

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

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

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

next

\textbf{case} False \textbf{note} b-not-hd = \textit{this}

\textbf{show} \( ?\text{thesis} \)

\textbf{proof} (cases \( a = \text{hd} \) (stack \( s \)))

\textbf{case} False \textbf{with} b-not-hd \( F \) \textbf{paren finish} \textbf{show} \( ?\text{thesis} \) by simp

next

\textbf{case} True \textbf{with} paren b-not-hd \( F \) \textbf{have}

\( a \in \text{dom} \) (discovered \( s \)) \( b \in \text{dom} \) (finished \( s \)) \( \delta \ s \ a < \delta \ s \ b \)

by simp-all

moreover \textbf{from} True stack-not-finished \textbf{have} \( a \notin \text{dom} \) (finished \( s \))

by simp

ultimately \textbf{show} \( ?\text{thesis} \) by (simp add: finish)

\textbf{qed}

\textbf{qed}

\textbf{qed}

\textbf{next}

\textbf{case} (new-root \( s \) \( s' \) \( v_0 \)) \textbf{then interpret} DFS-invar \textbf{where} \( s=s \) by simp

\textbf{from} new-root finished-discovered \textbf{have} \( v_0 \notin \text{dom} \) (finished \( s \)) by auto

moreover \textbf{note} timing-less-counter finished-discovered

ultimately \textbf{show} \( ?\text{case} \) using new-root by clarsimp force

\textbf{qed simp-all}

end

end

c\textbf{ontext} DFS-invar \textbf{begin context} \textbf{begin interpretation} timing-syntax .

\textbf{lemmas} s-paren-imp-tree-reach =

\textit{paren-imp-tree-reach}[THEN make-invar-thm]

\textbf{lemma} parenthesis-impl-tree-path-not-finished:

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

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

\textbf{and} \( \delta \ s \ v < \delta \ s \ w \)

\textbf{and} \( v \notin \text{dom} \) (finished \( s \))

\textbf{shows} \( (v,w) \in (\text{tree-edges} \ s)^\exists \)

\textbf{using} s-paren-imp-tree-reach \textbf{assms}

\textbf{by} (auto simp add: paren-imp-tree-reach-def)

\textbf{lemma} parenthesis-impl-tree-path:

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

\textbf{and} \( \delta \ s \ v < \delta \ s \ w \) \( \varphi \ s \ v > \varphi \ s \ w \)

\textbf{shows} \( (v,w) \in (\text{tree-edges} \ s)^\exists \)

\textbf{proof} =

\textbf{from} \textbf{assms}(1) \textbf{have} \( v \in \text{dom} \) (discovered \( s \))

\textbf{using} finished-discovered \textbf{by} blast

\textbf{with} \textbf{assms} \textbf{show} \( ?\text{thesis} \)

\textbf{using} s-paren-imp-tree-reach \textbf{assms}

\textbf{by} (auto simp add: paren-imp-tree-reach-def)

\textbf{qed}
lemma tree-path-iff-parenthesis:
assumes \( v \in \text{dom} (\text{finished } s) \) \( w \in \text{dom} (\text{finished } s) \)
shows \((v,w) \in (\text{tree-edges } s)^+ \iff \delta s v < \delta s w \land \varphi s v > \varphi s w \)
using assms
by (metis parenthesis-impl-tree-path tree-path-impl-parenthesis)

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

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 \((v,w) \in (\text{tree-edges } s)^+ \)
using no-pending-succ-impl-path-in-tree finished-no-pending finished-discovered
using assms
by blast
end

Properties of Cross Edges

context param-DFS begin context

lemma i-cross-edges-finished: is-invar \((\lambda s. \forall (u,v) \in \text{cross-edges } s).\)
\( v \in \text{dom} \ (\text{finished } s) \land (u \in \text{dom} \ (\text{finished } s) \rightarrow \varphi \ s \ v < \varphi \ s \ u) \)

**proof** (induction rule: is-invarI-full)

- **case** (finish \( s \ s' \ u \ e \))
  - **interpret** DFS-invar \( G \) param \( s \) by fact
  - from finish stack-not-finished have \( u \not\in \text{dom} \ (\text{finished } s) \) by auto
  - with finish show ?case by (auto intro: timing-less-counter)

- **next**
  - **case** (cross-edge \( s \ s' \ u \ v \ e \))
  - **interpret** DFS-invar \( G \) param \( s \) by fact
  - from cross-edge stack-not-finished have \( u \not\in \text{dom} \ (\text{finished } s) \) by auto
  - with cross-edge show ?case by (auto intro: timing-less-counter)

**qed simp-all**

end end

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

**lemmas** cross-edges-finished

\( = \ i\)-cross-edges-finished TH.en make-invar-thm\]

**lemma** cross-edges-target-finished:

\( (u,v)\in\text{cross-edges } s \implies v \in \text{dom} \ (\text{finished } s) \)

**using** cross-edges-finished by auto

**lemma** cross-edges-finished-decr:

\( [(u,v)\in\text{cross-edges } s; u\in\text{dom} \ (\text{finished } s)] \implies \varphi \ s \ v < \varphi \ s \ u \)

**using** cross-edges-finished by auto

**lemma** cross-edge-unequal:

- **assumes** cross: \((v,w)\in\text{cross-edges } s\)
- **shows** \( v \neq w \)

**proof**

- from cross-edges-target-finished(OF cross) have \( w\)-fin: \( v \in \text{dom} \ (\text{finished } s) \).

**show** ?thesis

**proof** (cases \( v \in \text{dom} \ (\text{finished } s) \))

- **case** True with cross-edges-finished-decr(OF cross)
  - **show** ?thesis by force

- **next**
  - **case** False with \( w\)-fin show ?thesis by force

**qed**

**qed**

end end

**Properties of Back Edges**

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

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

proof (induct rule: is-invarI-full)

\text{case (back-edge } s s' u v) \text{ then interpret } DFS-invar \text{ where } s=s \text{ by simp}

from back-edge have st: \(v \in \text{set (stack } s\) u \in \text{set (stack } s\)

using stack-set-def

by auto

have \((v,u) \in (\text{tree-edges } s)^+ \lor u = v\)

proof (rule disjCI)

assume \(u \neq v\)

with \(st \text{ back-edge have } v \in \text{set (tl (stack } s\) by (metis not-hd-in-tl)\}

with \(tl\text{-lt-stack-hd-discover } st \text{ back-edge have } \delta s v < \delta s u \text{ by simp}\)

with \(on-stack-is-tree-path } st \text{ show } (v,u) \in (\text{tree-edges } s)^+ \text{ by simp}\)

qed

with back-edge show ?case by auto

next

\text{case discover thus ?case using trancl-sub-insert-trancl by force}\n
qed simp-all

end end

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

lemma back-edge-impl-tree-path:

\([(v,w) \in \text{back-edges } s; v \neq w] \implies (w,v) \in (\text{tree-edges } s)^+\]

using i-back-edge-impl-tree-path[THEN make-invar-thm]

by blast

lemma back-edge-disc:

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

shows \(\delta s w \leq \delta s v\)

proof cases

assume \(v \neq w\)

with \(assms \text{ back-edge-impl-tree-path have } (w,v) \in (\text{tree-edges } s)^+ \text{ by simp}\)

with \(tree-path-disc } show \?thesis by force\)

qed simp

lemma back-edges-tree-disjoint:

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

using \(\text{back-edge-disc tree-edge-disc}\)

by force

lemma back-edges-tree-pathes-disjoint:

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

using \(\text{back-edge-disc tree-path-disc}\)

by force

lemma back-edge-finished:

55
assumes \((v, w) \in \text{back-edges } s\)
and \(w \in \text{dom } (\text{finished } s)\)
shows \(v \in \text{dom } (\text{finished } s) \land \varphi s v \leq \varphi s w\)
proof (cases \(v = w\))
  case True with \(\text{assms}\) show \(\text{?thesis by simp}\)
next
case False with \(\text{back-edge-impl-tree-path } \text{assms}\) have \((w, v) \in (\text{tree-edges } s)^+\)
  by simp
    with \(\text{tree-path-impl-parenthesis } \text{assms}\) show \(\text{?thesis by fastforce}\)
qed

end end

context \(\text{param-DFS}\) begin context
begin interpretation \(\text{timing-syntax}\).

lemma \(\text{i-disc-imp-back-edge-or-pending}:\)
  is-invar \((\lambda s. \forall (v, w) \in E.\)
  \(v \in \text{dom } (\text{discovered } s) \land w \in \text{dom } (\text{discovered } s)\)
  \(\land \delta s v \geq \delta s w\)
  \(\land (w \in \text{dom } (\text{finished } s) \Rightarrow v \in \text{dom } (\text{finished } s) \land \varphi s w \geq \varphi s v)\)
  \(\Rightarrow (v, w) \in \text{back-edges } s \lor (v, w) \in \text{pending } s)\)
proof (induct rule: is-invarI-full)
  case (cross-edge \(s s' u v\)) then interpret \(\text{DFS-invar}\) where \(s = s\) by simp
  from cross-edge stack-not-finished[of \(u\)] have \(u \notin \text{dom } (\text{finished } s)\)
    using hd-in-set
    by (auto simp add: cond-alt)
  with cross-edge show \(\text{?case by auto}\)
next
case (finish \(s s' u v\)) then interpret \(\text{DFS-invar}\) where \(s = s\) by simp
from finish have
  \(\text{IH: } \forall v, w. [w \in \text{succ } v; v \in \text{dom } (\text{discovered } s); w \in \text{dom } (\text{discovered } s);\)
  \(\delta s w \leq \delta s v;\)
  \((w \in \text{dom } (\text{finished } s) \Rightarrow v \in \text{dom } (\text{finished } s) \land \varphi s w \leq \varphi s v)\]
  \(\Rightarrow (v, w) \in \text{back-edges } s \lor (v, w) \in \text{pending } s\)
by blast
from finish have \(\text{ne[simp]}: \text{stack } s \neq []\)
  and \(\text{p[simp]}: \text{pending } s = {}\)
by (simp-all)
from hd-in-set[OF ne] have disc: \(\text{hd } (\text{stack } s) \in \text{dom } (\text{discovered } s)\)
  and not-fin: \(\text{hd } (\text{stack } s) \notin \text{dom } (\text{finished } s)\)
using stack-discovered stack-not-finished
by blast+

\{ fix \(w\)
assume \(w: w \in \text{succ } (\text{hd } (\text{stack } s)) w \neq \text{hd } (\text{stack } s) w \in \text{dom } (\text{discovered } s)\)

56
and $f$: $w \in \text{dom}(\text{finished } s) \rightarrow \text{counter } s \leq \varphi s w$
and $\delta$: $\delta s w \leq \delta s (\text{hd (stack } s))$

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

moreover

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

ultimately show ?case
  using finish
  by clarsimp auto
next
  case $(\text{discover } s s' u v)$ then interpret DFS-invar where $s=s$ by simp

  from discover show ?case
    using timing-less-counter
    by clarsimp fastforce
next
  case $(\text{new-root } s s' v0)$ then interpret DFS-invar where $s=s$ by simp

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

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

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

lemma finished-imp-back-edge:
[w \in \text{succ } v; v \in \text{dom (finished } s); w \in \text{dom (finished } s); \delta s w \leq \delta s v; \varphi s v \leq \varphi s w]

\[
(v, w) \in \text{back-edges } s
\]

using disc-imp-back-edge-or-pending finished-discovered finished-no-pending
by fast

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

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

context DFS-invar begin

context begin interpretation timing-syntax.

lemma tree-cross-acyclic:
acyclic \((\text{tree-edges } s \cup \text{cross-edges } s)\) (is acyclic ?E)
proof (rule ccontr)
{
fix \(u, v\)
assume \(\ast\): \(u \in \text{dom } (\text{finished } s)\) and \((u, v) \in ?E^+\)
from this(2) have \(\varphi s v \prec \varphi s u \land v \in \text{dom } (\text{finished } s)\)
proof induct
\hspace{1em}case base thus \(\mathcal{G}\) case
by (metis Un-iff cross-edges-finished-decr cross-edges-target-finished
\text{tree-edge-impl-parenthesis})
next
\hspace{1em}case (step \(v, w\))
hence \(\varphi s w \prec \varphi s v \land w \in \text{dom } (\text{finished } s)\)
by (metis Un-iff cross-edges-finished-decr cross-edges-target-finished
\text{tree-edge-impl-parenthesis})
with step show \(\mathcal{G}\) case by auto
qed
\}

note aux = this

assume \(\neg\)acyclic ?E
then obtain \(u\) where path: \((u, u) \in ?E^+\) by (auto simp add: acyclic-def)
show False
proof cases
  assume u ∈ dom (finished s)
  with aux path show False by blast
next
  assume *: u ∉ dom (finished s)
  moreover
  from no-loop-in-tree have (u,u) ∉ (tree-edges s)+ .
  with trancl-union-outside[OF path] obtain x y where (u,x) ∈ ?E+ (x,y) ∈ cross-edges s (y,u) ∈ ?E+ by auto
  with cross-edges-target-finished have y ∈ dom (finished s) by simp
  moreover with * ⟨(y,u) ∈ ?E+⟩ have (y,u) ∈ ?E+ by (auto simp add: rtrancl-eq-or-trancl)
  ultimately show False by (metis aux)
qed
qed
end

lemma cycle-contains-back-edge:
  assumes cycle: (u,u) ∈ (edges s)+
  shows ∃ v w. (u,v) ∈ (edges s)* ∧ (v,w) ∈ back-edges s ∧ (w,u) ∈ (edges s)*
proof −
  from tree-cross-acyclic have (u,u) ∉ (tree-edges s ∪ cross-edges s)+ by (simp add: acyclic-def)
qed

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

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

59
qed auto
thus ?thesis by (auto simp add: acyclic-def)
qed

lemma back-edge-closes-reachable-cycle:
back-edges s ≠ {} ⇒ ¬ acyclic (E ∩ reachable × UNIV)
by (metis back-edge-closes-cycle edges-ss-reachable-edges cyclic-subset)

lemma cycle-iff-back-edges:
acyclic (edges s) ←→ back-edges s = {}
by (metis back-edge-closes-cycle cycle-needs-back-edge)

1.2.4 White Path Theorem
context DFS begin
context begin interpretation timing-syntax.
definition white-path where
white-path s x y ≡ x ≠ y → (∃ p. path E x p y ∧ (δ s x < δ s y ∧ (∀ v ∈ set (tl p), δ s x < δ s v)))
lemma white-path:
it-dfs ≤ SPEC (λs. ∀ x ∈ reachable. ∀ y ∈ reachable. ¬ is-break param s →
white-path s x y ↔ (x,y) ∈ (tree-edges s)*)
proof (rule it-dfs-SPEC, intro ballI impI)
fix s x y
assume DI: DFS-invar G param s
and C: ¬ cond s ¬ is-break param s
and reach: x ∈ reachable y ∈ reachable

from DI interpret DFS-invar where s=s.

note fin-eq-reach = nc-finished-eq-reachable[OF C]
show white-path s x y ↔ (x,y) ∈ (tree-edges s)*
proof (cases x=y)
case True thus ?thesis by (simp add: white-path-def)
next
case False

show ?thesis
proof
  assume (x,y) ∈ (tree-edges s)*
  with (x≠y) have T: (x,y) ∈ (tree-edges s)* by (metis rtranclD)
  then obtain p where P: path (tree-edges s) x p y by (metis trancl-is-path)
  with tree-edges-ssE have path E x p y using path-mono[where E=tree-edges s]
by simp
moreover
from P have \( \delta s x < \delta s y \land (\forall v \in \text{set} (\text{tl} p). \delta s x < \delta s v) \)
using \((x \not= y)\)
proof (induct rule: path-tl-induct)
case (single u) thus ?case by (fact tree-edge-disc)
next
case (step u v) note \( \delta s x < \delta s w \)
also from step have \( \delta s u < \delta s v \) by (metis tree-edge-disc)
finally show ?case .
qed
ultimately show \( \text{white-path} s x y \)
by (auto simp add: \((x \not= y)\) white-path-def)
next
assume \( \text{white-path} s x y \)
with \((x \not= y)\) obtain \( p \) where
\( P: \text{path} E x p y \) and
\( \text{white}: \delta s x < \delta s y \land (\forall v \in \text{set} (\text{tl} p). \delta s x < \delta s v) \)
unfolding white-path-def
by blast
hence \( p \neq [] \) by auto
thus \((x,y) \in (\text{tree-edges} s)^*\) using \( P \) white reach(2)
proof (induction \( p \) arbitrary: \( y \) rule: rev-nonempty-induct)
case single hence \( y \in \text{succ} x \) by (simp add: path-cons-conv)
with reach single show ?case
using fin-eq-reach finished-succ-impl-path-in-tree[of \( x y \)]
by simp
next
case (snoc \( u us \)) hence \( \text{path} E x us u \) by (simp add: path-append-conv)
moreover hence \((x,u) \in E^*\) by (simp add: path-is-rtrancl)
with reach have \( \text{ureach}: u \in \text{reachable} \)
by (metis rtrancl-image-advance-rtrancl)
moreover from snoc have \( \delta s x < \delta s u \) \((\forall v \in \text{set} (\text{tl} us). \delta s x < \delta s v) \)
by simp-all
ultimately have \( x-u: (x,u) \in (\text{tree-edges} s)^* \) by (metis snoc.IH)

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

from \((y \in \text{succ} w)\) ureach fin-eq-reach have \( \delta s y < \varphi s u \)
using finished-succ-fin by simp
also from \( \delta s x < \delta s w \) have \( x \not= u \) by auto
with \( x-u \) have \((x,u) \in (\text{tree-edges} s)^*\) by (metis rtrancl-eq-or-trancl)
with fin-eq-reach reach have \( \varphi s u < \varphi s x \)
using tree-path-impl-parenthesis
by simp
finally have \( \varphi s y < \varphi s x \)

61
using reach fin-eq-reach y-f-d snoc
using parenthesis-contained
by blast
hence \((x, y) \in (\text{tree-edges } s)^+\)
using reach fin-eq-reach y-f-d snoc
using parenthesis-impl-tree-path
by blast
thus ?case by auto
qed
qed
qed
qed
end end

1.3 Invariants for SCCs

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

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

context param-DFS-defs begin
abbreviation \( scc\text{-root} \equiv scc\text{-root}' \ E \)
lemmas \( scc\text{-root-def} = scc\text{-root}'\text{-def} \)

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

definition \( scc\text{-roots } s = \{v. \ \exists \text{ scc. } scc\text{-root } s \ v \ scc\} \)
end

context DFS-invar begin
lemma scc-root-is-discovered:
  scc-root s v scc \Rightarrow v \in \text{dom} (\text{discovered} s)
by (simp add: scc-root-def)

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

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

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

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

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

lemma scc-root-unique-root:
  assumes scc1: scc-root s v scc
  and scc2: scc-root s v' scc
  shows v = v'
proof (rule ccontr)
  assume v \neq v'
  from scc1 have v \in scc v \in \text{dom} (\text{discovered} s)
    by (simp-all add: scc-root-def)
with $scc-root-scc-tree-trancl[\text{OF scc2}] \langle v \neq v' \rangle$ have $(v',v) \in (\text{tree-edges } s)^+$ by simp
also from $scc2$ have $v' \in scc \; v' \in \text{dom (discovered s)}$
by (simp-all add: $scc-root-def$)
with $scc-root-scc-tree-trancl[\text{OF scc1}] \langle v \neq v' \rangle$ have $(v,v') \in (\text{tree-edges } s)^+$
by simp
finally show False using no-loop-in-tree by contradiction
qed

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

lemma $scc-root-finished-impl-scc-finished$:
assumes $v \in \text{dom (finished s)}$\; and $scc-root \; s \; v \; scc$
shows $scc \subseteq \text{dom (finished s)}$
proof
fix $x$
assume $x \in scc$
let $?E = \text{Restr } E \; scc$
from assms have $\text{is-scc } E \; scc \; v \in scc$ by (simp-all add: $scc-root-def$)
hence $(v,x) \in (\text{Restr } E \; scc)^*$ using $x \in scc$
by (simp add: $\text{is-scc-connected'}$)
with rtrancl-is-path obtain $p$ where $\text{path } ?E \; v \; p \; x$ by metis
thus $x \in \text{dom (finished s)}$
proof (induction $p$ arbitrary: $x$ rule: rev-induct)
case Nil hence $v = x$ by simp
with assms show ?case by simp
next
case $(\text{snoc } y \; ys)$ hence $\text{path } ?E \; v \; ys \; y \; (y,x) \in ?E$
by (simp-all add: path-append-conv)
with snoc III have $y \in \text{dom (finished s)}$ by simp
moreover from $(y,x) \in ?E$ have $(y,x) \in E \; x \in scc$ by auto
ultimately have $x \in \text{dom (discovered s)}$
using finished-impl-succ-discovered
by blast
with $(x \in scc)$ show ?case
using assms $scc-root-scc-tree-trancl$ tree-path-impl-parenthesis
by blast
qed
context begin interpretation timing-syntax.

lemma scc-root-disc-le:
  assumes scc-root s v scc
  and \( x \in \text{sec} \) \( x \in \text{dom} (\text{discovered} \ s) \)
  shows \( \delta s v \leq \delta s x \)

proof (cases \( x = v \))
  case False with assms scc-root-scc-tree-trancl tree-path-disc have
  \( \delta s v < \delta s x \)
  by blast
  thus ?thesis by simp
qed simp

lemma scc-root-fin-ge:
  assumes scc-root s v scc
  and \( v \in \text{dom} (\text{finished} \ s) \)
  and \( x \in \text{sec} \)
  shows \( \varphi s v \geq \varphi s x \)

proof (cases \( x = v \))
  case False from assms scc-root-finished-impl-scc-finished have
  \( x \in \text{dom} (\text{finished} \ s) \) by auto
  hence \( x \in \text{dom} (\text{discovered} \ s) \) using finished-discovered by auto
  with assms False have \( (v, x) \in (\text{tree-edges} \ s)^+ \)
  using scc-root-scc-tree-trancl by simp
  with tree-path-impl-parenthesis assms False show ?thesis by force
qed simp

lemma scc-root-is-Min-disc:
  assumes scc-root s v scc
  shows \( \text{Min} (\delta s \ \text{scc} \cap \text{dom} (\text{discovered} \ s))) = \delta s v \) (is \( ?S = - \))

proof (rule Min-eqI)
  from discovered-finite show finite \( ?S \) by auto
  from scc-root-disc-le[\{OF assms\}] show \( \forall y. y \in ?S \implies \delta s v \leq y \) by force
from assms have \( v \in \text{sec} \) \( v \in \text{dom} (\text{discovered} \ s) \)
  by (simp-all add: scc-root-def)
  thus \( \delta s v \in ?S \) by auto
qed

lemma Min-disc-is-scc-root:
  assumes \( v \in \text{sec} \) \( v \in \text{dom} (\text{discovered} \ s) \)
  and \( \text{is-scc} E \ \text{scc} \)
  and min: \( \delta s v = \text{Min} (\delta s \ \text{scc} \cap \text{dom} (\text{discovered} \ s)) \)
  shows scc-root s v scc

proof (cases x = v)
  fix y
assume \( A: y \in \text{scc} \) \( y \in \text{dom} (\text{discovered } s) \) \( y \neq v \)

with \( \min \) have \( \delta s v \leq \delta s y \) by \( \text{auto} \)

with \( \text{assms disc-unequal } A \) have \( \delta s v < \delta s y \) by \( \text{fastforce} \)

} note \( \text{scc-disc} = \text{this} \)

{ fix \( x \)
assume \( A: x \in \text{scc} \cap \text{dom} (\text{discovered } s) \)

have \( x \in (\text{tree-edges } s)^* \) \{ \( v \) \}

proof (cases \( v = x \))
  case False with \( \text{A scc-disc} \) have \( \delta: \delta s v < \delta s x \) by \( \text{simp} \)

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

proof (cases \( v \in \text{dom} (\text{finished } s) \))
  case False with \( \text{stack-set-def } \text{assms} \) have \( v\text{-stack}: v \in \text{set} (\text{stack } s) \) by \( \text{auto} \)
  show \( ?\text{thesis} \)
  proof (cases \( x \in \text{dom} (\text{finished } s) \))
    case True
    with parenthesis-impl-tree-path-not-finished \( [v \ x] \) \( \text{assms} \) \( \delta \) \( \text{False} \)
    show \( ?\text{thesis} \) by \( \text{auto} \)
  next
    case False with \( \text{stack-set-def} \) have \( x \in \text{set} (\text{stack } s) \) by \( \text{auto} \)
    show \( v\text{-stack} \)
    proof
      with \( \text{v-stack} \)
      show \( ?\text{thesis} \) using \( \text{on-stack-is-tree-path} \)
        by \( \text{simp} \)
    qed

next
  case True note \( v\text{-fin} = \text{this} \)

let \( ?E = \text{Restr } E \text{ scc} \)

{ fix \( y \)
assume \( (v, y) \in ?E \) and \( v \neq y \)

hence \( *: y \in \text{succ } v \ y \in \text{scc} \) by \( \text{auto} \)

with \( \text{finished-imp-succ-discovered v-fin} \) \( \text{have} \)

\( y \in \text{dom} (\text{discovered } s) \) by \( \text{simp} \)

with \( \text{scc-disc } (v \neq y) \) \( * \) \( \text{have} \) \( \delta s v < \delta s y \) by \( \text{simp} \)

with \( \text{* } \) \( \text{finished-succ-impl-path-in-tree v-fin} \) \( \text{have} \ (v,y) \in (\text{tree-edges } s)^+ \)

by \( \text{simp} \)

} note \( \text{trancl-base} = \text{this} \)

from \( A \) have \( x \in \text{scc} \) by \( \text{simp} \)

with \( \text{assms} \) \( \text{have} \ (v,x) \in ?E^* \)

by \( \text{(simp add: is-scc-connected)} \)

with \( (v \neq x) \) \( \text{have} \ (v,x) \in ?E^* \) by \( \text{(metis rtrancl-eq-or-rtrancl)} \)

thus \( ?\text{thesis} \) using \( (v \neq x) \)


proof (induction)
  case (base y) with trancl-base show ?case .
next
  case (step y z)
  show ?case
  proof (cases v = y)
  case True with step trancl-base show ?thesis by simp
  next
  case False with step have \((v, y) \in (\text{tree-edges } s)^+\) by simp
  with \text{tree-path-impl-parenthesis}[OF - v-fin] have
  \(y\)-fin: \(y \in \text{dom} (\text{finished } s)\)
  and \(y\)-t: \(\delta s v < \delta s y \varphi s y < \varphi s v\)
  by auto
  with \text{finished-discovered have} \(y\)-disc: \(y \in \text{dom} (\text{discovered } s)\)
  by auto
  from step have \(*: z \in \text{succ } y z \in \text{see } by \text{auto}\)
  with \text{finished-imp-succ-discovered} \(y\)-fin have
  \(z\)-disc: \(z \in \text{dom} (\text{discovered } s) \text{ by simp}\)
  with \(* (v\neq z) \text{ have } \delta z: \delta s v < \delta s z \text{ by } (\text{simp add: } \text{scc-disc})\)
  from \(* \text{ edges-covered finished-no-pending}[OF / y \in \text{dom} (\text{finished } s)]\]
  \(y\)-disc have \((y, z) \in \text{edges } s \text{ by auto}\)
  thus ?thesis
  proof safe
    assume \((y, z) \in \text{tree-edges } s \text{ with } (v, y) \in (\text{tree-edges } s)^+\) show
    ?thesis ..
  next
    assume CE: \((y, z) \in \text{cross-edges } s\)
    with \text{cross-edges-finished-decr} \(y\)-fin \(y\)-t have \(\varphi s z < \varphi s v\)
    by force
    moreover note \(\delta z\)
    moreover from CE \text{cross-edges-target-finished have}
    \(z \in \text{dom} (\text{finished } s) \text{ by simp}\)
    ultimately show ?thesis
    using \text{parenthesis-impl-tree-path}[OF \(v\)-fin] by metis
  next
    assume BE: \((y, z) \in \text{back-edges } s\)
    with \text{back-edge-disc-lt-fin} \(y\)-fin \(y\)-t have
    \(\delta s z < \varphi s v \text{ by force}\)
    moreover note \(\delta z\)
    moreover note \(z\)-disc
    ultimately have \(z \in \text{dom} (\text{finished } s) \varphi s z < \varphi s v\)
    using \text{parenthesis-contained}[OF \(v\)-fin] by \text{simp-all}
    with \(\delta z\) show ?thesis
    using \text{parenthesis-impl-tree-path}[OF \(v\)-fin] by metis
  qed
  qed
qed
qed
thus ?thesis by auto
qed simp
}

hence \( \text{scc} \cap \text{dom} (\text{discovered } s) \subseteq (\text{tree-edges } s)^* \) " \{v\} by blast

with assms show ?thesis by (auto intro: scc-root!)
qed

lemma scc-root-iff-Min-disc:
assumes is-scc E scc r \in scc r \in \text{dom} (\text{discovered } s)
shows scc-root s r scc \iff Min (\delta s ' (\text{scc} \cap \text{dom} (\text{discovered } s))) = \delta s r (\text{is ?L} \iff ?R)
proof
next
  assume ?R with Min-disc-is-scc-root assms show ?L by simp
qed

lemma scc-root-exists:
assumes is-scc E scc
and scc: scc \cap \text{dom} (\text{discovered } s) \not= \{\}
shows \exists r. \text{scc-root } s r scc
proof
  let ?S = scc \cap \text{dom} (\text{discovered } s)
  from discovered-finite have finite (\delta s?qS) by auto
moreover from scc have \delta s ' ?S \not= \{\} by auto
  moreover have \forall (x::nat) f A. x \notin f ' A \lor (\exists y. x = f y \land y \in A) by blast
— autogenerated by sledgehammer
ultimately have \exists x \in ?S. \delta x = Min (\delta s ' ?S) by (metis Min-in)
with Min-disc-is-scc-root (is-scc E scc) show ?thesis by auto
qed

lemma scc-root-of-node-exists:
assumes v \in \text{dom} (\text{discovered } s)
shows \exists r. \text{scc-root } s r (\text{scc-of } E v)
proof
  have is-scc E \text{(scc-of } E v) by simp
moreover have v \in scc-of E v by simp
with assms have scc-of E v \cap \text{dom} (\text{discovered } s) \not= \{\} by blast
ultimately show ?thesis using scc-root-exists by metis
qed

lemma scc-root-transfer':
assumes discovered s = discovered s' tree-edges s = tree-edges s'
shows scc-root s r scc \iff scc-root s' r scc
unfolding scc-root-def
lemma scc-root-transfer:
assumes inv: DFS-invar G param s'
assumes r-d: r ∈ dom (discovered s)
assumes d: dom (discovered s) ⊆ dom (discovered s')
∀ x ∈ dom (discovered s), δ s x = δ s' x
∀ x ∈ dom (discovered s') − dom (discovered s), δ s' x ≥ counter s
and t: tree-edges s ⊆ tree-edges s'
shows scc-root s r scc ←→ scc-root s' r scc
proof
interpret s': DFS-invar where s = s' by fact
let ?sd = scc ∩ dom (discovered s)
let ?sd' = scc ∩ dom (discovered s')
let ?sdd = scc ∩ (dom (discovered s') − dom (discovered s))
{ assume r-s: r ∈ scc is-scc E scc
  with r-d have ne: δ s' ?sd ≠ {} by blast
  from discovered-finite have fin: finite (δ s' ?sd) by simp
  from timing-less-counter d have ∃ x ∈ δ s' ?sd ⇒ x < counter s by auto
  hence Min: Min (δ s' ?sd) < counter s
  using Min-less-iff by blast
  from d have Min (δ s' ?sd) = Min (δ s' ?sd') by (auto simp: image-def)
  also from d have ?sd' = ?sd ∪ ?sdd by auto
  hence s': δ s' ?sd' = δ s' ?sd ∪ δ s' ?sdd by auto
  hence Min (δ s' ?sd') = Min (δ s' ?sd')
  proof (cases ?sdd = {})
    case False
    from d have ∃ x ∈ δ s' ?sdd ⇒ x ≥ counter s by auto
    moreover from False have ne': δ s' ?sdd ≠ {} by blast
    moreover from s'.discovered-finite have fin': finite (δ s' ?sdd) by blast
    ultimately have Min (δ s' ?sdd) ≥ counter s
    using Min-ge-iff by metis
    with Min Min-Un[OF fin ne fin'] have thesis by simp
  qed simp
  finally have Min (δ s' ?sd) = Min (δ s' ?sd')
} note aux = this
show ?thesis
proof
  assume r: scc-root s r scc
  from r-d d have δ s' r = δ s r by simp
  also from r scc-root-is-Min-disc have δ s r = Min (δ s' ?sd) by simp
  also from r aux have Min (δ s' ?sd) = Min (δ s' ?sd') by (simp add: scc-root-def)
finally show scc-root s' r scc
  using r-d d r[unfolded scc-root-def]
  by (blast intro!: s'.Min-disc-is-scc-root)

next
  assume r': scc-root s' r scc
  from r-d d have δ s r = δ s' r by simp
  also from r' s'.scc-root-is-Min-disc have δ s' r = Min (δ s' ' ?sd') by simp
  also from r' aux have Min (δ s' ' ?sd') = Min (δ s ' ?sd) by (simp add: scc-root-def)

finally show scc-root s r scc
  using r-d d r[unfolded scc-root-def]
  by (blast intro!: Min-disc-is-scc-root)
qed
qed

dend end
dend

1.4 Generic DFS and Refinement

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

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

named-theorems DFS-code-unfold ⟨DFS framework: Unfolding theorems to pre-
pare term for automatic refinement⟩:

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

1.4.1 Generic DFS Algorithm

record ('v,'s) gen-dfs-struct =
  gds-init :: 's nres
  gds-is-break :: 's ⇒ bool
  gds-is-empty-stack :: 's ⇒ bool
  gds-new-root :: 'v ⇒ 's ⇒ 's nres
  gds-get-pending :: 's ⇒ ('v × 'v option × 's) nres
  gds-finish :: 'v ⇒ 's ⇒ 's nres
  gds-is-discovered :: 'v ⇒ 's ⇒ bool
  gds-is-finished :: 'v ⇒ 's ⇒ bool
  gds-back-edge :: 'v ⇒ 'v ⇒ 's ⇒ 's nres
locale gen-dfs-defs =
 fixes gds :: ('v,'s) gen-dfs-struct
 fixes V0 :: 'v set
 begin

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

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

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

 definition gen-dfsT ≡
  gds-init gds ≫ WHILE gen-cond gen-step

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

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

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

 definition pre-get-pending s ≡
  gen-rwof s ∧ ¬gds-is-empty-stack gds s ∧ ¬gds-is-break gds s
definition post-get-pending u Vs s0 s ≡ pre-get-pending s0 ∧ inres (gds-get-pending gds s0) (u, Vs, s)

definition pre-finish u s0 s ≡ post-get-pending u None s0 s

definition pre-cross-edge u v s0 s ≡ post-get-pending u (Some v) s0 s ∧ gds-is-discovered gds v s ∧ gds-is-finished gds v s

definition pre-back-edge u v s0 s ≡ post-get-pending u (Some v) s0 s ∧ gds-is-discovered gds v s ∧ ¬gds-is-finished gds v s

definition pre-discover u v s0 s ≡ post-get-pending u (Some v) s0 s ∧ ¬gds-is-discovered gds v s

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

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

definition gen-dfs-assert ≡
gds-init gds ≫ WHILE gen-cond gen-step-assert

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

72
abbreviation \[\text{gen-rwof-assert} \equiv \text{rwof} (\text{gds-init gds}) \text{gen-cond gen-step-assert}\]

lemma \[\text{gen-step-eq-assert}; \begin{array}{} \text{gen-cond} s; \text{gen-rwof} s \end{array} \implies \text{gen-step} s = \text{gen-step-assert} s\]
apply (rule antisym)
subgoal
apply (unfold \text{gen-step-def}[abs-def] \text{gen-step-assert-def}[abs-def]) []
apply (rule \text{refine-IdD})
apply \text{refine-rcg}
apply \text{refine-dref-type}
by simp-all

subgoal
apply (simp (no-asn) only: \text{gen-step-def}[abs-def] \text{gen-step-assert-def}[abs-def]) []
apply (unfold \text{GHOST-elim-Let}) []
apply (rule \text{refine-IdD})
apply (\text{refine-rcg bind-refine}')
apply \text{refine-dref-type}
by (auto simp: \text{pre-defs gen-cond-def})
done

lemma \[\text{gen-dfs-eq-assert}; \text{gen-dfs} = \text{gen-dfs-assert}\]
unfolding \text{gen-dfs-def gen-dfs-assert-def}
apply (rule antisym)
subgoal
apply (\text{unfold \text{GHOST-elim-Let}}) []
apply (rule \text{refine-IdD})
by (\text{refine-rcg, refine-dref-type, simp-all}) []

subgoal
apply (\text{subst (2) WHILE-eq-I-rwof})
apply (\text{refine-IdD})
apply (\text{refine-rcg, simp-all})
apply (simp (no-asn) only: \text{gen-step-def}[abs-def] \text{gen-step-assert-def}[abs-def]) []
apply (unfold \text{GHOST-elim-Let}) []
apply (rule \text{refine-IdD})
apply (\text{refine-rcg bind-refine}')
apply \text{refine-dref-type}
by (auto simp: \text{pre-defs gen-cond-def})
done

lemma \[\text{gen-dfsT-eq-assert}; \text{gen-dfsT} = \text{gen-dfsT-assert}\]
unfolding gen-dfsT-def gen-dfsT-assert-def
apply (rule antisym)

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

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

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

done

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

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

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

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

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

apply (rule refine-IdD)

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

done

lemma gen-dfs-le-gen-dfsT: gen-dfs ≤ gen-dfsT

unfolding gen-dfs-def gen-dfsT-def

apply (rule bind-mono)

apply simp

unfolding WHILET-def WHILE-def

apply (rule WHILEI-le-WHILEIT)

done

end

locale gen-dfs = gen-dfs-defs gds V0

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

and V0 :: 'v set

record ('v,'s,'es) gen-basic-dfs-struct =
  gbs-init :: 'es ⇒ '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)

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

definition \text{do-new-root} \ v0 \equiv \text{do-action} \ (\text{gbs-new-root} \ gbs \ v0) \ (\text{on-new-root} \ \text{param} \ v0)

definition \text{do-finish} \ u \equiv \text{do-action} \ (\text{gbs-finish} \ gbs \ u) \ (\text{on-finish} \ \text{param} \ u)

definition \text{do-back-edge} \ u \ v \equiv \text{do-action} \ (\text{gbs-back-edge} \ gbs \ u \ v) \ (\text{on-back-edge} \ \text{param} \ u \ v)

definition \text{do-cross-edge} \ u \ v \equiv \text{do-action} \ (\text{gbs-cross-edge} \ gbs \ u \ v) \ (\text{on-cross-edge} \ \text{param} \ u \ v)

definition \text{do-discover} \ u \ v \equiv \text{do-action} \ (\text{gbs-discover} \ gbs \ u \ v) \ (\text{on-discover} \ \text{param} \ u \ v)

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

definition \text{gds} \equiv \{} \\
\text{gds-init} = \text{do-init}, \\
\text{gds-is-break} = \text{is-break param}, \\
\text{gds-is-empty-stack} = \text{gbs-is-empty-stack} \ gbs, \\
\text{gds-new-root} = \text{do-new-root}, \\
\text{gds-get-pending} = \text{gbs-get-pending} \ gbs, \\
\text{gds-finish} = \text{do-finish}, \\
\text{gds-is-discovered} = \text{gbs-is-discovered} \ gbs, \\
\text{gds-is-finished} = \text{gbs-is-finished} \ gbs, \\
\text{gds-back-edge} = \text{do-back-edge}, \\
\text{gds-cross-edge} = \text{do-cross-edge}, \\
\text{gds-discover} = \text{do-discover} \\
\}

\text{lemmas} \ \text{gds-simps}[\text{simp,DFS-code-unfold}] = \text{gen-dfs-struct.simps}[\text{ink-record-simp}, \text{OF} \ \text{gds-def}]

\text{sublocale} \ \text{gen-dfs-defs} \ \text{gds} \ V0 .
\end

\text{locale} \ \text{gen-param-dfs} = \text{gen-param-dfs-defs} \ \text{gbs} \ \text{param} \ \text{upd-ext} \ V0 \\
\text{for} \ \text{gbs} :: \ ('v,'s,'es) \ \text{gen-basic-dfs-struct} \\
\text{and} \ \text{param} :: \ ('v,'s,'es) \ \text{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.

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

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

  apply (simp
    cong: if-cong option.case-cong)
  done

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

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

77
lemma gen-dfsT-simp[simp]: gen-dfs.gen-dfsT = it-dfsT
unfolding it-dfsT-def gen-dfs.gen-dfsT-def
apply (simp)
done
end

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

1.4.2 Refinement Between DFS Implementations

locale gen-dfs-refine-defs =
c: gen-dfs-defs gdsi V0i + a: gen-dfs-defs gds V0
for gdsi V0i gds V0
locale gen-dfs-refine =
c: gen-dfs gdsi V0i + a: gen-dfs gds V0 +
gen-dfs-refine-defs gdsi V0i gds V0
fixes V S
assumes BIJV[relator-props]: bijective V
assumes V0-param[param]: (V0i, V0)∈⟨V⟩set-rel
assumes is-discovered-param[param]:
(gds-is-discovered gdsi, gds-is-discovered gds)∈V→S→bool-rel
assumes is-finished-param[param]:
(gds-is-finished gdsi, gds-is-finished gds)∈V→S→bool-rel
assumes is-empty-stack-param[param]:
(gds-is-empty-stack gdsi, gds-is-empty-stack gds)∈S→bool-rel
assumes is-break-param[param]:
(gds-is-break gdsi, gds-is-break gds)∈S→bool-rel
assumes init-refine[refine]:
gds-init gdsi ≤⇓S (gds-init gds)
assumes new-root-refine[refine]:
[a.pre-new-root v0 s; (v0i,v0)∈V; (si,s)∈S]
⇒ gds-new-root gdsi v0i si ≤⇓S (gds-new-root gds v0 s)
assumes get-pending-refine[refine]:
[a.pre-get-pending s; (si,s)∈S]
⇒ gds-get-pending gdsi si ≤⇓(V×r⟨V⟩option-rel×rS) (gds-get-pending
gds s)
assumes finish-refine[refine]:
[a.pre-finish v s0 s; (vi,v)∈V; (si,s)∈S]
⇒ gds-finish gdsi vi si ≤⇓S (gds-finish gds v s)
assumes cross-edge-refine[refine]:
[a.pre-cross-edge u v s0 s; (ui,u)∈V; (vi,v)∈V; (si,s)∈S]
⇒ gds-cross-edge gdsi ui vi si ≤⇓S (gds-cross-edge gds u v s)
assumes back-edge-refine[refine]:
[a.pre-back-edge u v s0 s; (ui,u)∈V; (vi,v)∈V; (si,s)∈S]
⇒ gds-back-edge gdsi ui vi si ≤⇓S (gds-back-edge gds u v s)
assumes discover-refine[refine]:
\[
\begin{align*}
\& \text{pre-discover } u \ v \ s0 \ s; \ (u,i) \in V; \ (v,i) \in V; \ (s,i) \in S \\
\Rightarrow \& gds-discover \ gdsi \ u \ v \ s0 \ s \leq \Downarrow S \ (gds-discover \ gds \ u \ v \ s)
\end{align*}
\]

begin

term gds-is-discovered gdsi

lemma select-v0-refine[refine]:
assumes s-param: \((s,i,s) \in S\)
shows SPEC (\(\lambda v0. \ v0 \in V \land \neg gds-is-discovered \ gdsi \ v0 \ s\)) \[\leq \Downarrow V \ (SPEC (\lambda v0. \ v0 \in V \land \neg gds-is-discovered \ gds \ v0 \ s))\]
apply (rule RES-refine)
apply (simp add: Bex-def[symmetric], elim conjE)
apply (erule set-refD1[OF V0-param], elim bexE)
apply (is-discovered-param[param-fo, OF - s-param]
apply auto

lemma gen-rwof-refine:
assumes NF: nofail (a.gen-dfs)
assumes RW: c.gen-rwof s
obtains \(s'\) where \((s,s') \in S\) and a.gen-rwof \(s'\)
proof
- from NF have NFa: nofail (a.gen-dfs-assert)
  unfolding a.gen-dfs-eq-assert
  apply (rule rwof-refine[OF RW NFa]
    unfolding a.gen-dfs-assert-def)
  apply (rule leofI, rule init-refine)
  unfolding c.gen-cond-def a.gen-cond-def
  apply (rule IdD)
  apply (simp only: subset-Collect-conv)
  apply parametricity
  unfolding c.gen-step-def a.gen-step-assert-def GHOST-elim-Let
  apply (rule leofI)
  apply (refine-reg IdD)
  apply simp-all
  apply ((rule IdD, parametricity) | (auto) []+)
  done
thus \?thesis
unfolding a.gen-rwof-eq-assert[OF NF, symmetric]
by (blast intro: that)
qed

lemma gen-step-refine[refine]: (si,s) ∈ S ⇒ c.gen-step si ≤⇓ S (a.gen-step-assert s)
  unfolding c.gen-step-def a.gen-step-assert-def GHOST-elim-Let
  apply (refine-rcg IdD)
  apply simp-all
  apply ((rule IdD, parametricity) | (auto [])) +
  done

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

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

end

locale gbs-refinement =
c: gen-param-dfs gbsi parami upd-exti V0i +
a: gen-param-dfs gbs param upd-ext V0
for gbsi parami upd-exti V0i gbs param upd-ext V0 +
fixes V S ES
assumes BJV: bijective V
assumes V0-param[Param]: (V0i,V0) ∈ (V) set-rel
assumes is-discovered-param[Param]:
  (gbs-is-discovered gbsi,gbs-is-discovered gbs) ∈ V → S → bool-rel
assumes is-finished-param[Param]:
  (gbs-is-finished gbsi,gbs-is-finished gbs) ∈ V → S → 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 \(\text{is-break-param}[\text{param}]:\)
\((\text{is-break parami}, \text{is-break param}) \in S \rightarrow \text{bool-rel}\)

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

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

assumes \(\text{gbs-get-pending-refine}[\text{refine}]:\)
\([a.\text{pre-get-pending} \ s; (si, s) \in S]\)
\(\implies \text{gbs-get-pending} \ gbsi \ si \leq \downarrow (V \times_r (V \text{option-rel} \times_r S)) \ (\text{gbs-get-pending} \ gbs s)\)

assumes \(\text{gbs-finish-refine}[\text{refine}]:\)
\([a.\text{pre-finish} \ v0 \ s; (vi, v) \in V; (si, s) \in S]\)
\(\implies \text{gbs-finish} \ gbsi \ vi \ si \leq \downarrow S \ (\text{gbs-finish} \ gbs v0 \ s)\)

assumes \(\text{gbs-cross-edge-refine}[\text{refine}]:\)
\([a.\text{pre-cross-edge} \ u \ v \ s0 \ s; (ui, u) \in V; (vi, v) \in V; (si, s) \in S]\)
\(\implies \text{gbs-cross-edge} \ gbsi \ ui \ vi \ si \leq \downarrow S \ (\text{gbs-cross-edge} \ gbs u \ v \ s)\)

assumes \(\text{gbs-back-edge-refine}[\text{refine}]:\)
\([a.\text{pre-back-edge} \ u \ v \ s0 \ s; (ui, u) \in V; (vi, v) \in V; (si, s) \in S]\)
\(\implies \text{gbs-back-edge} \ gbsi \ ui \ vi \ si \leq \downarrow S \ (\text{gbs-back-edge} \ gbs u \ v \ s)\)

assumes \(\text{gbs-discover-refine}[\text{refine}]:\)
\([a.\text{pre-discover} \ u \ v \ s0 \ s; (ui, u) \in V; (vi, v) \in V; (si, s) \in S]\)
\(\implies \text{gbs-discover} \ gbsi \ ui \ vi \ si \leq \downarrow S \ (\text{gbs-discover} \ gbs u \ v \ s)\)

locale \(\text{param-refinement} =\)
\(c: \text{gen-param-dfs} \ gbsi \ \text{parami} \ \text{upd-exti} \ V0i +\)
\(a: \text{gen-param-dfs} \ gbs \ \text{param} \ \text{upd-ext} \ V0 +\)
for \(\text{gbsi} \ \text{parami} \ \text{upd-exti} \ V0i \ \text{gbs} \ \text{param} \ \text{upd-ext} \ V0 +\)

fixes \(V \ S ES\)

assumes \(\text{upd-ext-param}[\text{param}]: (\text{upd-exti}, \text{upd-ext}) \in (ES \rightarrow ES) \rightarrow S \rightarrow S\)

assumes \(\text{on-init-refine}[\text{refine}]: \ \text{on-init} \ \text{parami} \leq \downarrow ES \ (\text{on-init} \ \text{param})\)

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

assumes \(\text{on-new-root-refine}[\text{refine}]:\)
\([a.\text{pre-new-root} \ v0 \ s; (v0i, v0) \in V; (si, s) \in S; (si', s') \in S; \text{nf-inres} (\text{gbs-new-root} \ gbs v0 \ s) \ s']\)

81
\[ \Rightarrow \text{on-new-root\ param}\ v0\ \text{i' } \leq \downarrow ES \text{ (on-new-root\ param\ v0\ s')} \]

**assumes** on-finish-refine[refine]:
\[ a. \text{pre-finish}\ v\ s0\ s; (v_i, v) \in V; (s_i, s) \in S; (s_i', s') \in S; \\
\text{nf-inres}\ (gbs-finish\ gbs\ v\ s)\ s' \] 
\[ \Rightarrow \text{on-finish\ param}\ vi\ si' \leq \downarrow ES \text{ (on-finish\ param\ v\ s')} \]

**assumes** on-cross-edge-refine[refine]:
\[ a. \text{pre-cross-edge}\ u\ v\ s0\ s; (u_i, u) \in V; (v_i, v) \in V; (s_i, s) \in S; \\
(s_i', s') \in S; \text{nf-inres}\ (gbs-cross-edge\ gbs\ u\ v\ s)\ s' \] 
\[ \Rightarrow \text{on-cross-edge\ param}\ ui\ vi\ si' \leq \downarrow ES \text{ (on-cross-edge\ param\ u\ v\ s')} \]

**assumes** on-back-edge-refine[refine]:
\[ a. \text{pre-back-edge}\ u\ v\ s0\ s; (u_i, u) \in V; (v_i, v) \in V; (s_i, s) \in S; \\
(s_i', s') \in S; \text{nf-inres}\ (gbs-back-edge\ gbs\ u\ v\ s)\ s' \] 
\[ \Rightarrow \text{on-back-edge\ param}\ ui\ vi\ si' \leq \downarrow ES \text{ (on-back-edge\ param\ u\ v\ s')} \]

**assumes** on-discover-refine[refine]:
\[ a. \text{pre-discover}\ u\ v\ s0\ s; (u_i, u) \in V; (v_i, v) \in V; (s_i, s) \in S; \\
(s_i', s') \in S; \text{nf-inres}\ (gbs-discover\ gbs\ u\ v\ s)\ s' \] 
\[ \Rightarrow \text{on-discover\ param}\ ui\ vi\ si' \leq \downarrow ES \text{ (on-discover\ param\ u\ v\ s')} \]

**locale** gen-param-dfs-refine-defs =
\[ c: \text{gen-param-dfs-defs}\ gbsi\ parami\ upd-exti\ V0i + \\
a: \text{gen-param-dfs-defs}\ gbs\ param\ upd-ext\ V0 \]
\[ \text{for}\ gbsi\ parami\ upd-exti\ V0i\ gbs\ param\ upd-ext\ V0 \]
\[ \text{begin} \]
\[ \text{sublocale}\ gen-dfs-refine\ c: gds\ V0i\ a: gds\ V0 . \]
\[ \text{end} \]

**locale** gen-param-dfs-refine =
\[ \text{gbs-refinement\ where}\ V=V\ \text{and}\ S=S\ \text{and}\ ES=ES + \\
\text{param-refinement\ where}\ V=V\ \text{and}\ S=S\ \text{and}\ ES=ES + \\
\text{gen-param-dfs-refine-defs} \]
\[ \text{for}\ V: (vixv)\ \text{set}\ \text{and}\ S: (sixs)\ \text{set}\ \text{and}\ ES: (esi'es)\ \text{set} \]
\[ \text{begin} \]
\[ \text{sublocale}\ gen-dfs-refine\ c: gds\ V0i\ a: gds\ V0\ V\ S \]
\[ \text{apply}\ unfold-locales \]
\[ \text{apply}\ (\text{simp-all}\ add:\ BJV\ V0-param\ a.do-action-defs\ c.do-action-defs) \]
\[ \text{apply}\ (\text{parametricity}+) [4] \]
\[ \text{apply}\ refine-rcg \]
\[ \text{apply}\ (\text{refine-rcg}\ bind-refine-abs',\ \text{assumption}+,\ \text{parametricity}) [] \]
\[ \text{apply}\ refine-rcg \]
\[ \text{apply}\ (\text{refine-rcg}\ bind-refine-abs',\ \text{assumption}+,\ \text{parametricity}) [] \]
\[ \text{apply}\ refine-rcg \]
\[ \text{apply}\ (\text{refine-rcg}\ bind-refine-abs',\ \text{assumption}+,\ \text{parametricity}) [] \]
\[ \text{apply}\ refine-rcg \]
\[ \text{apply}\ (\text{refine-rcg}\ bind-refine-abs',\ \text{assumption}+,\ \text{parametricity}) [] \]

82
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
define tailrec-implT where [DFS-code-unfold]:
tailrec-implT ≡ do {
  s ← gds-init gds;

FOREACHci
  (λit s.
define tailrec-implT where [DFS-code-unfold]:
tailrec-implT ≡ do {
  s ← gds-init gds;

FOREACHci
  (λit s.
defi
WHILEIT

(λs. gen-rwof s ∧ insert v0 (gen-discovered s) ⊆ gen-discovered s)
(λs. ¬gds-is-break gds s ∧ ¬gds-is-empty-stack gds s)
tr-impl-while-body s

s

| } s
| }

| } s
| }

| } s
| }

definition tailrec-impl where [DFS-code-unfold]:
tailrec-impl ≡ do {
  s ← gds-init gds;
  FOREACH ci
  (λit s.
    gen-rwof s
    ∧ (¬gds-is-break gds s ⟷ gds-is-empty-stack gds s )
    ∧ V0 − it ⊆ gen-discovered s)
  V0
  (Not o gds-is-break gds)
  (λv0 s. do {
    let — ghost: s0 = s;
    if gds-is-discovered gds v0 s then
      RETURN s
    else do {
      s ← gds-new-root gds v0 s;
      WHILEI
      (λs. gen-rwof s ∧ insert v0 (gen-discovered s) ⊆ gen-discovered s)
      (λs. ¬gds-is-break gds s ∧ ¬gds-is-empty-stack gds s)
      (λs. do {
        (u, Vs, s) ← gds-get-pending gds s;
        case Vs of
         None ⇒ gds-finish gds u s
        | Some v ⇒ do {
          if gds-is-discovered gds v s then do {
            if gds-is-finished gds v s then
              gds-cross-edge gds u v s
            else
              gds-back-edge gds u v s
          } else
          gds-discover gds u v s
        }
        )
      )
      s
    }
  }
| }

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:
  \[\begin{array}{l}
  \text{pre-new-root } v0 s \\
  \implies gds-new-root gds v0 s \leq_n SPEC (\lambda s'. \\
  \hspace{1em} \text{insert } v0 (\text{gen-discovered } s) \subseteq \text{gen-discovered } s')
  \end{array}\]

assumes get-pending-incr:
  \[\begin{array}{l}
  \text{pre-get-pending } s \\
  \implies gds-get-pending gds s \leq_n SPEC (\lambda (s'). \\
  \hspace{1em} \text{gen-discovered } s \subseteq \text{gen-discovered } s')
  \end{array}\]

assumes finish-incr: \[\begin{array}{l}
  \text{pre-finish } u s0 s \\
  \implies gds-finish gds u s \leq_n SPEC (\lambda s'. \\
  \hspace{1em} \text{gen-discovered } s \subseteq \text{gen-discovered } s')
  \end{array}\]

assumes cross-edge-incr:
  \[\begin{array}{l}
  \text{pre-cross-edge } u v s0 s \\
  \implies gds-cross-edge gds u v s \leq_n SPEC (\lambda s'. \\
  \hspace{1em} \text{gen-discovered } s \subseteq \text{gen-discovered } s')
  \end{array}\]

assumes back-edge-incr:
  \[\begin{array}{l}
  \text{pre-back-edge } u v s0 s \\
  \implies gds-back-edge gds u v s \leq_n SPEC (\lambda s'. \\
  \hspace{1em} \text{gen-discovered } s \subseteq \text{gen-discovered } s')
  \end{array}\]

assumes discover-incr:
  \[\begin{array}{l}
  \text{pre-discover } u v s0 s \\
  \implies gds-discover gds u v s \leq_n SPEC (\lambda s'. \\
  \hspace{1em} \text{gen-discovered } s \subseteq \text{gen-discovered } s')
  \end{array}\]

begin

context
assumes nofail:
  nofail (gds-init gds \Rightarrow WHILE gen-cond gen-step)
begin

lemma gds-init-refine: gds-init gds
  \leq SPEC (\lambda s. \text{gen-rwof } s \land \text{gds-is-empty-stack } gds s)
  apply (rule SPEC-rule-conj-leofI1)
  apply (rule rwof-init[OF nofail])
  apply (rule init-empty-stack)
  done

lemma gds-new-root-refine:
  assumes PNR: pre-new-root v0 s
  shows gds-new-root gds v0 s
  \leq SPEC (\lambda s'. \text{gen-rwof } s' \\
  \hspace{1em} \land \text{insert } v0 (\text{gen-discovered } s) \subseteq \text{gen-discovered } s')
  apply (rule SPEC-rule-conj-leofI1)
  apply (rule order-trans[OF - rwof-step[OF nofail]])

end
using PNR apply (unfold gen-step-def gen-cond-def pre-new-root-def) [3]
apply (simp add: pw-le-iff refine-pw-simps, blast)
apply simp
apply blast

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

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

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

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

from rwof-step[OF nofail RWOF COND]
have gen-step s ≤ SPEC gen-rwof .

hence nofail (gen-step s) by (simp add: pw-le-iff)

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

lemma gds-get-pending-refine:
assumes PRE: pre-get-pending s
shows gds-get-pending gds s ≤ SPEC (\(\lambda(u, Vs, s'). \, \) post-get-pending u Vs s s' 
∧ gen-discovered s ⊆ gen-discovered s')
(proof –

have gds-get-pending gds s ≤ SPEC (\(\lambda(u, Vs, s'). \, \) post-get-pending u Vs s s')
unfolding post-get-pending-def
apply (simp add: PRE)
using get-pending-nofail[OF PRE]
apply (simp add: pw-le-iff)
done

moreover note get-pending-incr[OF PRE]
ultimately show ?thesis by (simp add: pw-le-iff pw-leaf-iff)
qed

lemma gds-finish-refine:
assumes PRE: pre-finish u s0 s
shows gds-finish gds u s ≤ SPEC (\(\lambda s'. \, \) gen-rwof s' 
∧ gen-discovered s ⊆ gen-discovered s')

86
apply (rule SPEC-rule-conj-leofI1)
apply (rule order-trans[OF - rwof-step[OF nofail]])
using PRE
apply (unfold gen-step-def gen-cond-def pre-finish-def
       post-get-pending-def pre-get-pending-def) [3]
apply (simp add: pw-le-iff refine-pw-simps split: option.split, blast)
apply simp
apply blast
apply (rule finish-incr[OF PRE])
done

lemma gds-cross-edge-refine:
assumes PRE: pre-cross-edge u v s0 s
shows gds-cross-edge gds u v s \leq SPEC (\lambda s'. gen-rwof s'
       \land gen-discovered s \subseteq gen-discovered s')
apply (rule SPEC-rule-conj-leofI1)
apply (rule order-trans[OF - rwof-step[OF nofail]])
using PRE
apply (unfold gen-step-def gen-cond-def pre-cross-edge-def
       post-get-pending-def pre-get-pending-def) [3]
apply (simp add: pw-le-iff refine-pw-simps split: option.split, blast)
apply simp
apply blast
apply (rule cross-edge-incr[OF PRE])
done

lemma gds-back-edge-refine:
assumes PRE: pre-back-edge u v s0 s
shows gds-back-edge gds u v s \leq SPEC (\lambda s'. gen-rwof s'
       \land gen-discovered s \subseteq gen-discovered s')
apply (rule SPEC-rule-conj-leofI1)
apply (rule order-trans[OF - rwof-step[OF nofail]])
using PRE
apply (unfold gen-step-def gen-cond-def pre-back-edge-def
       post-get-pending-def pre-get-pending-def) [3]
apply (simp add: pw-le-iff refine-pw-simps split: option.split, blast)
apply simp
apply blast
apply (rule back-edge-incr[OF PRE])
done

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

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

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

end

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

theorem tailrec-impl: tailrec-impl \leq gen-dfs
unfolding gen-dfs-def
apply (rule WHILE-refine-rwof)
unfolding tailrec-impl-def
apply (refine-reg refine-vec
  order-trans[OF gds-init-refine]
  order-trans[OF gds-new-root-refine]
)
order-trans[OF gds-get-pending-refine]
order-trans[OF gds-finish-refine]
order-trans[OF gds-cross-edge-refine]
order-trans[OF gds-back-edge-refine]
order-trans[OF gds-discover-refine]

apply (auto
    simp: it-step-insert-iff gen-cond-def
    pre-new-root-def pre-get-pending-def pre-finish-def
    pre-cross-edge-def pre-back-edge-def pre-discover-def)
done

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

lemma tailrecT-impl: tailrec-implT \leq gen-dfsT
proof (rule le-nofailI)
  let ?V = rwof-rel (gds-init gds) gen-cond gen-step
  assume NF: nofail gen-dfsT
  from nofail-WHILET-wf-rel[of gds-init gds λ-. True gen-cond gen-step]
    and this[unfolded gen-dfsT-def WHILET-def]
  have WF: wf (?V⁻¹) by simp
  from NF have NF': nofail gen-dfs using gen-dfs-le-gen-dfsT
    by (auto simp: pw-le-iff)
  from rwof-rel-spec[of gds-init gds gen-cond gen-step] have
    \A s. [gen-rwof s; gen-cond s] \impl gen-step s \leq SPEC (\A s'. (s,s')\in ?V)
  hence
    aux: \A s. [gen-rwof s; gen-cond s] \impl gen-step s \leq SPEC (\A s'. (s,s')\in ?V)
    apply (rule leofD[rotated])
    apply assumption
    apply assumption
    using NF[unfolded gen-dfsT-def]
    by (drule (1) WHILET-nofail-imp-rwof-nofail)
  show ?thesis
    apply (rule order-trans[OF - gen-dfs-le-gen-dfsT])
    apply (rule order-trans[OF - tailrec-impl])
    unfolding tailrec-implT-def tailrec-impl-def
    unfolding tr-impl-while-body-def[symmetric]
    apply (rule refine-IdD)
    apply (refine-reg bind-refine' inj-on-id)
    apply refine-dref-type
    apply simp-all

89
apply (subst WHILEIT-eq-WHILEI-tproof[where V=V⁻¹])
apply (rule WF; fail)
subgoal
apply clarsimp
apply (rule order-trans[OF tr-impl-while-body-gen-step], assumption)
apply (rule aux, assumption, (simp add: gen-cond-def; fail))
done
apply (simp; fail)
done
qed
end

1.6 Recursive DFS Implementation

theory Rec-Impl
imports General-DFS-Structure
begin
locale rec-impl-defs =
  graph-defs G + gen-dfs-defs gds V0
for G :: (′v, ′more) graph-rec-scheme
and gds :: (′v,′s)gen-dfs-struct +
fixes pending :: ′s ⇒ ′v rel
fixes stack :: ′s ⇒ ′v list
fixes choose-pending :: ′v ⇒ ′v option ⇒ ′s ⇒ ′s nres
begin

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

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

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

rec-impl ≡ do {
  s ← gds-init gds;

  FOREACH\text{ci}
  (λit s.
    gen-rwof' s
    ∧ (∼gds-is-break gds s → gds-is-empty-stack gds s
    ∧ V0—it \subseteq 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
        ∧ (3 stk. stack s = u#stk)
        ∧ E \cap \{u\} \times UNIV \subseteq pending s)
        (λ(u,s)' s'.
          gen-rwof' s'
          ∧ (∼gds-is-break gds s' →
            stack s' = tl (stack s)
          ∧ pending s' = pending s \{u\} \times UNIV
          ∧ gen-discovered s' \supseteq gen-discovered s
        ))
        (λD (u,s). do {
          s ← FOREACH\text{ci}
          (λit s'. gen-rwof' s'
          ∧ (∼gds-is-break gds s' →
            stack s' = stack s
          ∧ pending s' = (pending s \{u\} \times (E''{u} \setminus it))
          ∧ gen-discovered s' \supseteq gen-discovered s \cup (E''{u} \setminus it)
        ))
        (E''{u}) (λs. ∼gds-is-break gds s)
        (λv s. do {
          s ← choose-pending u (Some v) s;
          if gds-is-discovered gds v s then do {
            if gds-is-finished gds v s then
              gds-cross-edge gds u v s
            else
              gds-back-edge gds u v s
          } else do {
        })
  })
}
\[
\begin{align*}
s & \leftarrow \text{gds-discover } gds \ u \ v \ s; \\
& \quad \text{if gds-is-break } gds \ s \ \text{then RETURN } s \ \text{else } (v,s) \\
\end{align*}
\]

\[
\begin{align*}
&) s; \\
& \quad \text{if gds-is-break } gds \ s \ \text{then} \\
& \quad \quad \text{RETURN } s \\
& \quad \text{else do } \{} \\
& \quad \quad s \leftarrow \text{choose-pending } u \ (\text{None}) \ s; \\
& \quad \quad s \leftarrow \text{gds-finish } gds \ u \ s; \\
& \quad \quad \text{RETURN } s \\
& \quad \} \\
& \} (v0,s) \\
& \} \\
& \} s
\]

**definition** \text{rec-impl-for-paper} \quad \text{where} \quad \text{rec-impl-for-paper} \equiv \{} \\
\quad s \leftarrow \text{gds-init } gds; \\
\quad \\text{FOREACHc } v0 \ (\text{Not } o \ \text{gds-is-break } gds) \ (\lambda v0 \ s. \ \text{do } \{ \\
\quad \quad \text{if gds-is-discovered } gds \ v0 \ s \ \text{then RETURN } s \\
\quad \quad \text{else do } \{} \\
\quad \quad \quad s \leftarrow \text{gds-new-root } gds \ v0 \ s; \\
\quad \quad \quad \text{if gds-is-break } gds \ s \ \text{then RETURN } s \\
\quad \quad \quad \text{else do } \{} \\
\quad \quad \quad \quad \text{REC } (\lambda D \ (u,s). \ \text{do } \{ \\
\quad \quad \quad \quad \quad s \leftarrow \text{FOREACHc } (E^c{u}) \ (\lambda s. \ \neg \text{gds-is-break } gds \ s) \ (\lambda v \ s. \ \text{do } \{ \\
\quad \quad \quad \quad \quad \quad s \leftarrow \text{choose-pending } u \ (\text{Some } v) \ s; \\
\quad \quad \quad \quad \quad \quad \text{if gds-is-discovered } gds \ v \ s \ \text{then do } \{ \\
\quad \quad \quad \quad \quad \quad \quad \text{if gds-is-finished } gds \ v \ s \ \text{then gds-cross-edge } gds \ u \ v \ s \\
\quad \quad \quad \quad \quad \quad \quad \text{else gds-back-edge } gds \ u \ v \ s \\
\quad \quad \quad \quad \quad \quad \} \ \text{else do } \{} \\
\quad \quad \quad \quad \quad \quad \quad s \leftarrow \text{gds-discover } gds \ u \ v \ s; \\
\quad \quad \quad \quad \quad \quad \quad \text{if gds-is-break } gds \ s \ \text{then RETURN } s \ \text{else } D \ (v,s) \\
\quad \quad \quad \quad \quad \quad \}) \ \} \\
\quad \quad \quad \}) \\
\quad \quad \}) \ \} (v0,s) \\
\quad \}) \ \} s
\}
locale rec-impl =
  fb-graph G + gen-dfs gds V0 + rec-impl-defs G gds pending stack choose-pending

for G :: (v, 'more) graph-rec-scheme
and gds :: (v',s)gen-dfs-struct
and pending :: 's ⇒ 'v rel
and stack :: 's ⇒ 'v list
and choose-pending :: 'v ⇒ 'v option ⇒ 's ⇒ 's nres
+
assumes [simp]: gds-is-empty-stack gds s ←→ stack s = []
assumes init-spec:
gds-init gds ≤ n SPEC (λs. stack s = [] ∧ pending s = {})
assumes new-root-spec:
[pre-new-root v0 s] 
⇒ gds-new-root gds v0 s ≤ n SPEC (λs'.
  stack s' = [v0] ∧ pending s' = {v0} × E''{v0} ∧
  gen-discovered s' = insert v0 (gen-discovered s))
assumes get-pending-fmt: [pre-get-pending s] ⇒
do {
  let u = hd (stack s);
  vo ← SELECT (λv. (u,v)∈pending s);
  s ← choose-pending u vo s;
  RETURN (u,vo,s)
}
≤ gds-get-pending gds s
assumes choose-pending-spec: [pre-get-pending s; u = hd (stack s); case vo of
  None ⇒ pending s "" {u} = {}
  | Some v ⇒ v∈pending s "" {u}]
⇒
choose-pending u vo s ≤ n SPEC (λs'.
  (case vo of
    None ⇒ pending s' = pending s
    | Some v ⇒ pending s' = pending s - {(u,v)}) ∧
    stack s' = stack s ∧
    (∀ x. gds-is-discovered gds x s' = gds-is-discovered gds x s)
)
assumes finish-spec: [pre-finish u s0 s]
⇒ gds-finish gds u s0 s ≤ n SPEC (λs'.
  pending s' = pending s ∧
  stack s' = tl (stack s) ∧
  (∀ x. gds-is-discovered gds x s' = gds-is-discovered gds x s))
assumes cross-edge-spec: pre-cross-edge u v s0 s
⇒ gds-cross-edge gds u v s ≤ n SPEC (λs').
pending \( s' = \text{pending} \ s \land \text{stack} \ s' = \text{stack} \ s \land \\
(\forall x. \text{gds-is-discovered} \ \text{gds} \ x \ s' = \text{gds-is-discovered} \ \text{gds} \ x \ s))

assumes back-edge-spec: \( \text{pre-back-edge} \ u \ v \ s_0 \ s \Rightarrow \text{gds-back-edge} \ \text{gds} \ u \ v \ s \leq_n \text{SPEC} \ (\lambda s'. \\
\text{pending} \ s' = \text{pending} \ s \land \text{stack} \ s' = \text{stack} \ s \land \\
(\forall x. \text{gds-is-discovered} \ \text{gds} \ x \ s' = \text{gds-is-discovered} \ \text{gds} \ x \ s)) \)

assumes discover-spec: \( \text{pre-discover} \ u \ v \ s_0 \ s \Rightarrow \text{gds-discover} \ \text{gds} \ u \ v \ s \leq_n \text{SPEC} \ (\lambda s'. \\
\text{pending} \ s' = \text{pending} \ s \cup (\{v\} \times E' \{v\}) \land \text{stack} \ s' = v \# \text{stack} \ s \land \\
\text{gen-discovered} \ s' = \text{insert} \ v \ (\text{gen-discovered} \ s)) \)

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

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

lemma gen-rwof'\text{-imp-rwof}:
  assumes NF: nofail \text{gen-dfs}
  assumes A: \text{gen-rwof}' \ s
  shows \text{gen-rwof} \ s
  apply (rule \text{rwof-step-refine})
  apply (rule NF[unfolded \text{gen-dfs-def}])
  apply fact
  apply (rule leof-lift[\text{OF \text{gen-step}'\text{-refine}}, assumption+] [])
  done

lemma reachable-invar:
  \text{gen-rwof}' \ s \Rightarrow \text{set} \ (\text{stack} \ s) \subseteq \text{reachable} \land \text{pending} \ s \subseteq E
  \land \text{set} \ (\text{stack} \ s) \subseteq \text{gen-discovered} \ s \land \text{distinct} \ (\text{stack} \ s)
\end{verbatim}

94
∧ pending s ⊆ set (stack s) × UNIV

apply (erule establish-rwof-invar[rotated -1])
apply (rule leaf-trans[OF init-spec], auto) []
apply (subst gen-step' -def)
apply (refine-reg refine-wcg
leaf-trans[OF new-root-spec]
SELECT-rule[THEN leaf-lift]
leaf-trans[OF choose-pending-spec[THEN leaf-strengthen-SPEC]]
leaf-trans[OF finish-spec]
leaf-trans[OF cross-edge-spec]
leaf-trans[OF back-edge-spec]
leaf-trans[OF discover-spec])

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

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

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

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

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

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

subgoal
  apply clarsimp
  by blast

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

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

lemma mk-spec-aux:
  \[ m \leq n \text{ SPEC } \Phi \quad m \leq \text{SPEC gen-rwof'} \quad \implies \quad m \leq \text{SPEC } (\lambda s. \text{gen-rwof'} s \land \Phi s) \]
  by (rule SPEC-rule-conj-leofI1)

definition post-choose-pending u vo s0 s ≡
gen-rwof' s0
\land \text{ gen-cond } s0
\land \text{ stack } s0 \neq []
\land \ u = \text{hd (stack } s0) 
\land \text{ inres (choose-pending u vo s0)} s
\land \text{ stack } s = \text{stack } s0
\land \ (\forall x. \text{ gds-is-discovered gds x s} = \text{gds-is-discovered gds x s0})

//gdr/inductive/inductive-proof-lean/gdr/gds/refine-pw.s/3/gds/inductive-proof-lean-gds/
\( (\text{case } v o \text{ of} \)
\nNone \( \Rightarrow \) pending \( s'' \{ \{ u \} \} = \{ \} \land \text{pending } s = \text{pending } s_0 \)
\| Some \( v \) \( \Rightarrow v \in \text{pending } s'' \{ \{ u \} \} \land \text{pending } s = \text{pending } s_0 - \{(u,v)\} \)
\)

context

assumes nofail:
  nofail (\text{gds-init gds } \Rightarrow \text{WHILE gen-cond gen-step'})
assumes nofail2:
  nofail (\text{gen-dfs})

begin

lemma pcp-imp-pgp:
  post-choose-pending \( u \) \( v o \) \( s_0 \) \( s \) \( \Rightarrow \) post-get-pending \( u \) \( v o \) \( s_0 \) \( s \)
unfolding post-choose-pending-def pre-defs
apply (intro conjI)
apply (simp add: gen-rwof''-imp-rwof[\text{OF nofail2}])
apply simp
apply (simp add: gen-cond-def)
apply (rule pwD2[\text{OF get-pending-fmt}])
apply (simp add: pre-defs gen-cond-def
  gen-rwof''-imp-rwof[\text{OF nofail2}])
apply (auto simp add: refine-pw-simps select-def split: option.splits) []
done

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

schematic-goal gds-new-root-refine:
\[ \text{[pre-new-root } v o \text{ s; gen-rwof' s] } \Rightarrow \text{gds-new-root gds } v o \text{ s } \leq \text{SPEC } \Phi \]
apply (rule mk-spec-aux[\text{OF new-root-spec}, assumption])
apply (rule order-trans[\text{OF - rwof-step[\text{OF nofail, where s=s}]}])
unfolding gen-step'-def pre-new-root-def gen-cond-def
apply (auto simp: pw-le-iff refine-pw-simps)
done

schematic-goal gds-choose-pending-refine:
assumes 1: \text{pre-get-pending } s
assumes 2: \text{gen-rwof' s}
assumes [simp]: \( u = \text{hd} \) \( \text{stack } s \)
assumes 3: case \( v o \) of
  None \( \Rightarrow \text{pending } s '' \{ \{ u \} \} = \{ \} \)
  | Some \( v \) \( \Rightarrow v \in \text{pending } s '' \{ \{ u \} \} \)
shows choose-pending \( u \) \( v o \) \( s \) \( \leq \text{SPEC } \) (post-choose-pending \( u \) \( v o \) \( s \))

proof
from \text{WHILE-nofail-imp-rwof-nofail[\text{OF nofail} 2] 1 3 have nofail (choose-pending \( u \) \( v o \) \( s \))}
unfolding pre-defs gen-step'-def gen-cond-def
by (auto simp: refine-pw-simps select-def

97
also have \( \text{choose-pending } u \text{ vo } s \leq_n \text{SPEC} \) (post-choose-pending \( u \text{ vo } s \))

apply (rule leaf-trans[\( OF \text{ choose-pending-spec} [OF 1 - 3, \text{THEN leaf-strengthen-SPEC}] \)])

apply simp

apply (rule leaf-RES-rule)

using 1

apply (simp add: post-choose-pending-def 2 pre-defs gen-cond-def split: option.splits)

using 3

apply auto
done

finally (leofD) show \( ?\text{thesis} \).

qed

schematic-goal gds-finish-refine:

\[
\text{[pre-finish } u \text{ s0 s; post-choose-pending } u \text{ None s0 s]} \implies \text{gds-finish gds u s } \leq \text{SPEC} \ ? \Phi
\]

apply (rule mk-spec-aux[\( OF \text{ finish-spec} \), assumption])

apply (rule order-trans[\( OF \text{ - rwof-step} \text{[OF nofail, where s=s0]}) \)])

unfolding gen-step'-def pre-defs gen-cond-def post-choose-pending-def

apply (auto simp: pw-le-iff refine-pw-simps split: option.split)

done

schematic-goal gds-cross-edge-refine:

\[
\text{[pre-cross-edge } u \text{ v s0 s; post-choose-pending } u \text{ (Some v) s0 s]} \implies \text{gds-cross-edge gds u v s } \leq \text{SPEC} \ ? \Phi
\]

apply (rule mk-spec-aux[\( OF \text{ cross-edge-spec} \), assumption])

apply (rule order-trans[\( OF \text{ - rwof-step} \text{[OF nofail, where s=s0]}) \)])

unfolding gen-step'-def pre-defs gen-cond-def post-choose-pending-def

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

apply simp

apply blast
done

schematic-goal gds-back-edge-refine:

\[
\text{[pre-back-edge } u \text{ v s0 s; post-choose-pending } u \text{ (Some v) s0 s]} \implies \text{gds-back-edge gds u v s } \leq \text{SPEC} \ ? \Phi
\]

apply (rule mk-spec-aux[\( OF \text{ back-edge-spec} \), assumption])

apply (rule order-trans[\( OF \text{ - rwof-step} \text{[OF nofail, where s=s0]}) \)])

unfolding gen-step'-def pre-defs gen-cond-def post-choose-pending-def

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

apply simp

apply blast
done

schematic-goal gds-discover-refine:

\[
\text{[pre-discover } u \text{ v s0 s; post-choose-pending } u \text{ (Some v) s0 s]} \implies \text{gds-discover}
\]
$gds \ u \ v \ s \leq \ SPEC \ \Phi$

apply (rule mk-spec-aux[OF discover-spec], assumption)
apply (rule order-trans[OF - rwof-step[OF nfail, where \ s=s0]])
unfolding gen-step'-def pre-defs gen-cond-def post-choose-pending-def
apply (simp add: pw-le-iff refine-pw-simps select-def split: option.split, blast)

apply simp
apply blast
done
end

lemma rec-impl-aux: \[ xd \in Domain \ P \] $\Rightarrow P - \{y\} \times (\text{succ } y - \text{ita}) - \{(y, xd)\} - UNIV = P - \text{insert } (y, xd) (\{y\} \times (\text{succ } y - \text{ita}))$
apply auto
done

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

using [[goals-limit = 1]]
apply (auto simp add: pre-defs; fail)
apply (auto simp add: pre-defs gen-rwof'-imp-rwof; fail)
apply (auto; fail)
apply (auto dest: reachable-invar; fail)
apply (auto simp add: pre-defs gen-rwof'-imp-rwof; fail)
apply (auto; fail)
apply (auto; fail)
apply ((drule pcp-imp-pgp, auto simp add: pre-defs gen-rwof'-imp-rwof); fail)
apply (auto simp: post-choose-pending-def; fail)
apply (auto simp: post-choose-pending-def; fail)
apply (auto simp: post-choose-pending-def; fail)
apply ((drule pcp-imp-pgp, auto simp: pre-defs gen-rwof'-imp-rwof); fail)
apply (auto simp: post-choose-pending-def; fail)
apply (auto simp: post-choose-pending-def; fail)
apply (auto simp: post-choose-pending-def; fail)
apply ((drule pcp-imp-pgp, auto simp: pre-defs gen-rwof'-imp-rwof); fail)
apply (rule order-trans)
apply rprems
apply (auto; fail) []
subgoal
  apply (rule SPEC-rule)
  apply (simp add: post-choose-pending-def gen-rwof'-imp-rwof
  split: if-split-asm)
  apply (clarsimp simp: post-choose-pending-def gen-rwof'-imp-rwof Un-Diff
  split: if-split-asm) []
  apply (clarsimp simp: it-step-insert-iff neq-Nil-cone)
  apply (rule conjI)
subgoal
  apply (rule rec-impl-aux)
  apply (drule reachable-invar)+
  apply (metis Domain.cases SigmaD1 mem-Collect-eq rev-subsetD)
done
subgoal
  apply (rule conjI)
  apply auto []
  apply (metis order-trans)
done
done
apply (auto simp add: pre-defs gen-rwof'-imp-rwof; fail)
apply (auto; fail)
apply (auto dest: reachable-invar; fail)
apply ((drule pcp-imp-pgp, auto simp add: pre-defs gen-rwof'-imp-rwof); fail)
apply (auto simp: post-choose-pending-def; fail)
apply (auto simp: post-choose-pending-def; fail)
apply (auto simp: post-choose-pending-def; fail)
apply (auto; fail)
apply (auto simp: gen-cond-def; fail)
apply (auto simp: gen-cond-def; fail)
done
end
end

1.7 Simple Data Structures

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

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

1.7.1 Stack, Pending Stack, and Visited Set

record 'v simple-state =
  ss-stack :: ('v × 'v set) list
  on-stack :: 'v set
  visited :: 'v set

definition [to-relAPP]: simple-state-rel erel ≡ { (s,s') .
  ss-stack s = map (\u. (u,pending s' " {u} )) (stack s') ∧
  on-stack s = set (stack s') ∧
  visited s = dom (discovered s') ∧
  dom (finished s') = dom (discovered s') − set (stack s') ∧ — TODO: Hmm, this
  is an invariant of the abstract
  set (stack s') ⊆ dom (discovered s') ∧
  (simple-state.more s, state.more s') ∈ erel
}]

lemma simple-state-relI:
  assumes
  dom (finished s') = dom (discovered s') − set (stack s')
  set (stack s') ⊆ dom (discovered s')
  (m', state.more s') ∈ erel
  shows (⟨
    ss-stack = map (\u. (u,pending s' " {u} )) (stack s'),
    on-stack = set (stack s'),
    visited = dom (discovered s'),
    ... = m'
  ⟩, s') ∈ erel) simple-state-rel
  using assms
  unfolding simple-state-rel-def
  by auto
lemma simple-state-more-refine [param]:
(simple-state.more-update, state.more-update)
∈ (R → R) → (R) simple-state-rel → (R) simple-state-rel
apply (clarsimp simp: simple-state-rel-def)
apply parametricity
done

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

locale pre-simple-impl = graph-defs
begin

definition init-impl e
≡ RETURN (| ss-stack = [], on-stack = {}, visited = {e})
definition is-empty-stack-impl s ≡ (ss-stack s = [])
definition is-discovered-impl u s ≡ (u ∈ visited s)
definition is-finished-impl u s ≡ (u ∈ visited s − (on-stack s))
definition finish-impl u s ≡ do
  ASSERT (ss-stack s ≠ [] ∧ u ∈ on-stack s);
  let s = s (ss-stack := tl (ss-stack s));
  let s = s (on-stack := on-stack s − {u});
  RETURN s
end

definition get-pending-impl s ≡ do
  ASSERT (ss-stack s ≠ []);
  let (u, Vs) = hd (ss-stack s);
  if Vs = {} then
    RETURN (u, None, s)
  else do
    v ← SPEC (λv. v ∈ Vs);
    let Vs = Vs − {v};
    let s = s (ss-stack := (u, Vs) # tl (ss-stack s));
    RETURN (u, Some v, s)
  end
end

definition discover-impl u v s ≡ do
  ASSERT (v ∉ on-stack s ∧ v ∉ visited s);
  let s = s (ss-stack := (v, E« v «)) # ss-stack s);
  let s = s (on-stack := insert v (on-stack s));
  let s = s (visited := insert v (visited s));
  RETURN s
end

definition new-root-impl v0 s ≡ do
  ASSERT (v0 ∉ visited s);
let s = s[ss-stack := [(v0,E''\{v0\})]];  
let s = s[on-stack := \{v0\}];  
let s = s[visited := insert v0 (visited s)];  
RETURN s  
}  

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

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

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

end

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

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

sublocale tailrec-impl-defs G c.gds .

definition get-pending s ≡ \bigcup set (map (\lambda(u, Vs). \{u\} \times Vs) (ss-stack s))  
definition get-stack s ≡ map fst (ss-stack s)  
definition choose-pending

103
\[:: \ 'v \Rightarrow (\ 'v, \ 'd) \ simple-state-scheme \Rightarrow (\ 'v, \ 'd) \ simple-state-scheme \]

where [DFS-code-unfold]:

```plaintext
choose-pending u vo s \equiv
  case vo of
    None \Rightarrow \ RETURN s
  \mid Some v \Rightarrow \ do
      \ ASSERT (ss-stack s \neq []);
      let \ (u, V_s) = \ hd (ss-stack s);
      \ RETURN \ ((s || ss-stack := (u, V_s - \{v\}) \# tl (ss-stack s)))
```

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

end

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

begin

lemma init-impl: \ (e_i, e) \in ES \implies c.init-impl \ e_i \leq \down\langle (ES) \ simple-state-rel \rangle \ (\ RETURN \ (a.empty-state e))
unfolding c.init-impl-def a.empty-state-def simple-state-rel-def
by (auto)

lemma new-root-impl:
[a.gen-dfs.pre-new-root v0 s;
 (v0i, v0) \in Id; (si, s) \in \langle ES \rangle \ simple-state-rel]
\implies c.new-root-impl v0 si \leq \down\langle (ES) \ simple-state-rel \rangle \ (\ RETURN \ (a.new-root v0 s))
unfolding simple-state-rel-def a.gen-dfs.pre-new-root-def c.new-root-impl-def
by (auto simp add: a.pred-defs)

lemma get-pending-impl:
[a.gen-dfs.pre-get-pending s; (si, s) \in \langle ES \rangle \ simple-state-rel]
\implies c.get-pending-impl si
\leq \down\langle Id \times_r Id \times_r \langle ES \rangle \ simple-state-rel \rangle \ (a.get-pending s)
apply (unfold a.get-pending-def c.get-pending-impl-def) []
apply (refine-reg bind-refine' Let-refine' IdI)
```

104
apply (refine-dref-type)
apply (auto
  simp: simple-state-rel-def a.gen-dfs.pre-defs a.pred-defs neq-nil-conv
  dest: DFS-invar.stack-distinct)
)
done

lemma inres-get-pending-None-conv: inres a.get-pending s0 (v, None, s)
  ⟷ s = s0 ∧ v = hd (stack s) ∧ pending s0''{hd (stack s0)} = {}
unfolding a.get-pending-def
by (auto simp add: refine-pw-simps)

lemma inres-get-pending-Some-conv: inres a.get-pending s0 (v, Some Vs, s)
  ⟷ v = hd (stack s) ∧ s = s0{(pending := pending s0 − {hd (stack s0)}, Vs)}
  ∧ (hd (stack s0), Vs) ∈ pending s0
unfolding a.get-pending-def
by (auto simp add: refine-pw-simps)

lemma finish-impl:
  [ a.gen-dfs.pre-finish v s0 s; (vi, v) ∈ Id; (si, s) ∈ ⟨ES⟩ simple-state-rel ]
  ⟹ c.finish-impl v si ≤⇓⟨ES⟩ simple-state-rel {RETURN (a.finish v s)}
unfolding simple-state-rel-def a.gen-dfs.pre-defs c.finish-impl-def
apply (clarsimp simp: inres-get-pending-None-conv)
apply (frule DFS-invar.stack-distinct)
apply (simp add: a.pred-defs map-tl)
apply (clarsimp simp: neq-nil-conv)
apply blast
done

lemma cross-edge-impl:
  [ a.gen-dfs.pre-cross-edge u v s0 s;
    (ui, u) ∈ Id; (vi, v) ∈ Id; (si, s) ∈ ⟨ES⟩ simple-state-rel ]
  ⟹ (si, a.cross-edge u v s) ∈ ⟨ES⟩ simple-state-rel
unfolding simple-state-rel-def a.gen-dfs.pre-defs
by simp

lemma back-edge-impl:
  [ a.gen-dfs.pre-back-edge u v s0 s;
    (ui, u) ∈ Id; (vi, v) ∈ Id; (si, s) ∈ ⟨ES⟩ simple-state-rel ]
  ⟹ (si, a.back-edge u v s) ∈ ⟨ES⟩ simple-state-rel
unfolding simple-state-rel-def a.gen-dfs.pre-defs
by simp

lemma discover-impl:
  [ a.gen-dfs.pre-discover u v s0 s; (ui, u) ∈ Id; (vi, v) ∈ Id; (si, s) ∈ ⟨ES⟩ simple-state-rel ]
  ⟹ c.discover-impl ui vi si ≤⇓⟨ES⟩ simple-state-rel {RETURN (a.discover}
Main outcome of this locale: The simple DFS-Algorithm, which is a general
DFS scheme itself (and thus open to further refinements), and a refinement theorem that states correct refinement of the original DFS

```
lemma simple-refine[refine]: c.gen-dfs ≤ \nu((ES) simple-state-rel) a.it-dfs
  using gen-dfs-refine
  by simp

lemma simple-refineT[refine]: c.gen-dfsT ≤ \nu((ES) simple-state-rel) a.it-dfsT
  using gen-dfsT-refine
  by simp
```

Link with tail-recursive implementation

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

lemma simple-tailrec-refine[refine]: tailrec-impl ≤ \nu((ES) simple-state-rel) a.it-dfs
  proof –
    note tailrec-impl also note simple-refine finally show ?thesis .
  qed

lemma simple-tailrecT-refine[refine]: tailrec-implT ≤ \nu((ES) simple-state-rel) a.it-dfsT
  proof –
    note tailrecT-impl also note simple-refineT finally show ?thesis .
  qed
```

Link to recursive implementation

```
lemma reachable-invar:
  assumes c.gen-rwof s
  shows set (map fst (ss-stack s)) ⊆ visited s
    ∧ distinct (map fst (ss-stack s))
  using assms
  apply (induct rule: establish-rwof-invar[rotated −1, consumes 1])
  apply (simp add: c.do-action-defs c.impl-defs[abs-def])
  apply (refine-reg refine-veg)
  apply simp
  apply (refine-reg refine-veg)
  apply simp-all
  apply (fastforce simp: neq-Nil-conv)
  apply (fastforce simp: neq-Nil-conv)
  apply (fastforce simp: neq-Nil-conv)
  done
```
sublocale rec-impl G c.gds get-pending get-stack choose-pending
apply unfold-locales
unfolding get-pending-def get-stack-def choose-pending-def
apply (simp-all add: c.do-action-defs c.impl-defs[abs-def])
apply (auto simp: pw-leof-iff refine-pw-simps pw-le-iff select-def
split: prod.split) []
apply (auto simp: pw-leof-iff refine-pw-simps pw-le-iff select-def
split: prod.split) []
apply (rule le-ASSERTI)
apply (unfold c.pre-defs, clarify) []
apply (frule reachable-invar)

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

lemma simple-rec-refine[refine]: rec-impl ≤ ⇓⟨ES⟩simple-state-rel a.it-dfs
proof –
note rec-impl also note simple-refine finally show thesis .
qed

end

Autoref Setup

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

definition [to-relAPP]: ss-impl-rel s-rel vis-rel erel ≡
{((ss-stack-impl = si, ss-on-stack-impl = osi, ss-visited-impl = visi, ... = mi),}
\[(ss\text{-}stack = s, on\text{-}stack = os, visited = vis, \ldots = m)\] |
\(si \, osi \, visi \, mi \, s \, os \, vis \, m.\)
\((si, s) \in s\text{-}rel \land \)
\((osi, os) \in vis\text{-}rel \land \)
\((visi, vis) \in vis\text{-}rel \land \)
\((mi, m) \in erel\)

\textbf{consts}
\begin{align*}
i\text{-}simple\text{-}state :: & \text{ interface } \Rightarrow \text{ interface } \Rightarrow \text{ interface } \Rightarrow \text{ interface} \\
\text{lemmas} & \quad \text{[autoref-rel-intf]} = REL\text{-}INTFI[\text{of ss\text{-}impl-rel i\text{-}simple\text{-}state}] \\
\text{term} & \quad \text{simple\text{-}state\text{-}ext} \\
\text{lemma} & \quad \text{[autoref-rules, param]:} \\
\text{fixes} & \quad s\text{-}rel \, ps\text{-}rel \, vis\text{-}rel \, erel \\
\text{defines} & \quad R \equiv (s\text{-}rel, vis\text{-}rel, erel)\text{ss\text{-}impl-rel} \\
\text{shows} & \quad (ss\text{-}stack\text{-}impl, ss\text{-}stack) \in R \Rightarrow s\text{-}rel \\
& \quad (ss\text{-}on\text{-}stack\text{-}impl, on\text{-}stack) \in R \Rightarrow vis\text{-}rel \\
& \quad (ss\text{-}visited\text{-}impl, visited) \in R \Rightarrow vis\text{-}rel \\
& \quad (simple\text{-}state\text{-}impl.more, simple\text{-}state.more) \in R \Rightarrow erel \\
& \quad (ss\text{-}stack\text{-}impl\text{-}update, ss\text{-}stack\text{-}update) \in (s\text{-}rel \Rightarrow s\text{-}rel) \Rightarrow R \Rightarrow R \\
& \quad (ss\text{-}on\text{-}stack\text{-}impl\text{-}update, on\text{-}stack\text{-}update) \in (vis\text{-}rel \Rightarrow vis\text{-}rel) \Rightarrow R \Rightarrow R \\
& \quad (ss\text{-}visited\text{-}impl\text{-}update, visited\text{-}update) \in (vis\text{-}rel \Rightarrow vis\text{-}rel) \Rightarrow R \Rightarrow R \\
& \quad (simple\text{-}state\text{-}impl.more\text{-}update, simple\text{-}state.more\text{-}update) \in (erel \Rightarrow erel) \Rightarrow R \Rightarrow R \\
\text{unfolding} & \quad ss\text{-}impl\text{-}rel\text{-}def \, R\text{-}def \\
\text{apply} & \quad auto \\
\text{apply} & \quad parametricity+ \\
\text{done} \\
\end{align*}

1.7.2 Simple state without on-stack

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

\textbf{record} \quad ('si, 'nsi)\text{simple\text{-}state\text{-}nos\text{-}impl} = \\
\quad \text{ssnos\text{-}stack\text{-}impl :: 'si} \\
\quad \text{ssnos\text{-}visited\text{-}impl :: 'nsi} \\

\textbf{definition} \quad \text{[to\text{-}relAPP]:} \quad \text{ssnos\text{-}impl-rel} \, s\text{-}rel \, vis\text{-}rel \, erel \equiv \\
\quad \{(ssnos\text{-}stack\text{-}impl = si, ssnos\text{-}visited\text{-}impl = visi, \ldots = mi), \\
\quad (ss\text{-}stack = s, on\text{-}stack = os, visited = vis, \ldots = m)\} | \\
\quad si \, mi \, s \, os \, vis \, m. \\
\quad (si, s) \in s\text{-}rel \land \\
\quad (visi, vis) \in vis\text{-}rel \land \\
\quad (mi, m) \in erel \}
lemmas [autoref-rel-intf] = REL-INTFI[of ssnos-impl-rel i-simple-state]

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

where op-nos-on-stack-update ≡ on-stack-update

context begin interpretation autoref-syn.

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

end

lemmas ssnos-unfolds — To be unfolded before autoref when using ssnos-impl-rel

= op-nos-on-stack-update-def[symmetric]

lemma [autoref-rules, param]:

fixes s-rel vis-rel erel

defines R ≡ ⟨s-rel, vis-rel, erel⟩ ssnos-impl-rel

shows (ssnos-stack-impl, ss-stack) ∈ R ⇒ s-rel

(ssnos-visited-impl, visited) ∈ R ⇒ vis-rel

(simple-state-nos-impl,more, simple-state,more) ∈ R ⇒ erel

(ssnos-stack-impl-update, ss-stack-update) ∈ (s-rel ⇒ s-rel) ⇒ R ⇒ R

(λz. x, op-nos-on-stack-update f) ∈ R ⇒ R

(ssnos-visited-impl-update, visited-update) ∈ (vis-rel ⇒ vis-rel) ⇒ R ⇒ R

(simple-state-nos-impl,more-update, simple-state,more-update) ∈ (erel ⇒ erel) ⇒ R ⇒ R

→ R ⇒ R

(λns - ps vs. simple-state-nos-impl-ext ns ps vs, simple-state-ext)

∈ s-rel ⇒ ANY-rel ⇒ vis-rel ⇒ erel ⇒ R

unfolding ssnos-impl-rel-def R-def op-nos-on-stack-update-def

apply auto

apply parametricity+

done

1.7.3 Simple state without stack and on-stack

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

record (′si,′nsi) simple-state-ns-impl =

ssns-visited-impl :: ′nsi

definition [to-relAPP]: ssnos-impl-rel (R::′a×′b) set vis-rel erel ≡

 provincialis s o u s i . . . = mi),

]| 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.

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

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

end

context simple-impl-defs begin

thm choose-pending-def[unfolded op-ns-stack-update-def[symmetric], no-vars]

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

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

end


**lemma** [antoref-rules, param]:

**fixes** s-rel vis-rel erel ANY-rel

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

**shows**

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

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

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

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

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

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

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

**unfolding** ssns-impl-rel-def R-def op-ns-on-stack-update-def op-ns-stack-update-def

**apply** auto

**apply** parametricity+

**done**

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

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

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

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

**by** auto

**done**

### 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 $G' = (\text{rel-restrict } E R, V0 - R)$ for some fb-graph $G = (E, V0)$. If, additionally, the parameterization is not "too sensitive" to the visited set, we can pre-initialize the visited set with $R$, and use the $V0$ and $E$ of $G$. This may be a more efficient implementation than 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 $R \equiv \{ (s, s') \}$

$\{\text{ss-stack } s, \text{ss-stack } s' \} \in \langle \text{Id } \times_r \{ (U, U') \}. U - R = U' \rangle \text{list-rel}$
∧ on-stack \( s = \text{on-stack } s' \)
∧ visited \( s = \text{visited } s' \cup R \land \text{visited } s' \cap R = \{\} \)
∧ simple-state\_more \( s = \text{simple-state\_more } s' \)

**Lemma** restr\_rel\_simps:
- **Assumes** \( (s,s') \in \text{restr-rel } R \)
- **Shows** visited \( s = \text{visited } s' \cup R \)
- and simple-state\_more \( s = \text{simple-state\_more } s' \)
- **Using** assms unfolding restr\_rel\_def by auto

**Lemma**
- **Assumes** \( (s,s') \in \text{restr-rel } R \)
- **Shows** restr\_rel\_stackD: \( (\text{ss-stack } s, \text{ss-stack } s') \in (\text{Id} \times_r \{(U,U')\}. U - R = U')\)\_list-rel
- and restr\_rel\_vis\_djD: visited \( s' \cap R = \{\} \)
- **Using** assms unfolding restr\_rel\_def by auto

**Context** fixes \( R :: 'v \text{ set} \)
**Begin**

**Definition** \([\text{to-relAPP}]; \text{restr-simple-state-rel } ES \equiv \{ (s,s') . (\text{ss-stack } s, \text{map } (\lambda u. (u\text{-pending } s' \cup \{u\}) \\text{-stack } s') \in (\text{Id} \times_r \{(U,U')\}. U - R = U')\)\_list-rel ∧
- on-stack \( s = \text{set } (\text{stack } s') \land \)
- visited \( s = \text{dom } (\text{discovered } s') \cup R \land \text{dom } (\text{discovered } s') \cap R = \{\} \land \)
- dom \( (\text{finished } s') = \text{dom } (\text{discovered } s') - \text{set } (\text{stack } s') \land \)
- set \( (\text{stack } s') \subseteq \text{dom } (\text{discovered } s') \land \)
- (simple-state\_more \( s, \text{state\_more } s' \) \in ES)
}

**End**

**Lemma** restr\_simple-state-rel\_combine:
\( (ES)\text{restr-simple-state-rel } R = \text{restr-rel } R \circ (ES)\text{simple-state-rel} \)
**Unfolding** restr\_simple-state-rel\_def
**Apply** (intro equalityI subsetI)
**Apply** clarify
**Apply** (rule relcompI[OF - simple-state-relI], auto simp: restr-rel-def) []
**Apply** (auto simp: restr-rel-def simple-state-rel-def) []
**Done**

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

**Locale** restricted-impl-defs =
graph\_defs \( G + \)
- \( a: \text{simple-impl-defs } \text{graph\_restrict } G \)
for \( G :: ('v, 'more) \text{ graph\_rec\_scheme} \)
and \( R \)
**Begin**
sublocale pre-simple-impl $G$ .

abbreviation $rel \equiv restr-rel R$

definition $gbs' \equiv gbs []$

$gbs$-init := $\lambda e\ldotp RETURN$

($ss\operatorname{-stack}=[]$, on-stack=$\{}$, visited = $R$, $\ldots=e \ldots\}$)

lemmas $gbs'$-simps[simp, DFS-code-unfold]

$\equiv gen$-basic-dfs-struct.simps[mk-record-simp, OF $gbs'$-def [unfolded $gbs$-simps]]

sublocale gen-param-dfs-defs $gbs'$ $param\ imi$ simple-state$m ore$-update $V0$ .

sublocale tailrec-impl-defs $G\ gds$ .

end

locale restricted-impl =

$fb$-graph +

$a$ : simple-impl graph-restrict $G\ R$ +

restricted-impl-defs +

assumes [simp]: on-cross-edge $param\ imi$ $= (\lambda u\ v\ s\ RETURN\ (simple-state.m ore\ s))$

assumes [simp]: on-back-edge $param\ imi$ $= (\lambda u\ v\ s\ RETURN\ (simple-state.m ore\ s))$

assumes is-break-refine:

\[ [ (s,s') \in restr-rel R ] \implies is$-break\ param\ imi$ s \leftrightarrow is-break param\ imi s' \]

assumes on-new-root-refine:

\[ [ (s,s') \in restr-rel R ] \implies on$-new-root\ param\ imi$ v0 s \leq on-new-root param\ imi v0 s' \]

assumes on-finish-refine:

\[ [ (s,s') \in restr-rel R ] \implies on$-finish\ param\ imi$ u s \leq on-finish param\ imi u s' \]

assumes on-discover-refine:

\[ [ (s,s') \in restr-rel R ] \implies on$-discover\ param\ imi$ u v s \leq on-discover param\ imi u v s' \]

begin
lemmas rel-def = restr-rel-def[where R=R]

sublocale gen-param-dfs gbs' parami simple-state.more-update V0 .

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

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

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

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

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

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

lemma do-new-root-refine[refine]:
\[(v_0i, v_0) \in \text{Id}; (s, s) \in \text{rel}; v_0 \notin R\]
\[\implies \text{do-new-root } v_0i \leq \Downarrow \text{rel (a.c.do-new-root } v_0 \ s)\]

**unfolding** do-action-defs a.c.do-action-defs

**apply** refine-rcg

**apply** (rule intro-prgR[where \(R=\text{rel}\)])

**apply** (simp add: a.c.new-root-impl-def new-root-impl-def)

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

**apply** (rule intro-prgR[where \(R=\text{Id}\)])

**apply** (simp add: on-new-root-refine)

**apply** (simp add: rel-def)

**done**

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

**unfolding** do-action-defs a.c.do-action-defs

**apply** refine-rcg

**apply** (rule intro-prgR[where \(R=\text{rel}\)])

**apply** (simp add: finish-impl-def is-empty-stack-impl-def)

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

**apply** parametricity

**apply** (rule intro-prgR[where \(R=\text{Id}\)])

**apply** (simp add: on-finish-refine)

**apply** (simp add: rel-def)

**done**

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

**unfolding** rel-def is-empty-stack-impl-def

**apply** (cases ss-stack s', simp)

**apply** (auto elim: list-relE)

**done**

**lemma** get-pending-refine:

**assumes** (s, s') \in \text{rel gen-cond } s \neg \text{is-empty-stack-impl } s

**shows**
\[
\text{get-pending-impl } s \leq \text{sup } (\Downarrow(\text{Id } \times, (\text{Id } \text{option-rel } \times, \text{rel} ) (\text{inf} \\
(\text{get-pending-impl } s'))
\]

(SPEC (\(\lambda(u, Vs, s'). \ \text{case Vs of None } \Rightarrow \text{True } | \text{Some } v \Rightarrow v \notin R\))))

(\(\Downarrow(\text{Id } \times, (\text{Id } \text{option-rel } \times, \text{rel} )
\)

SPEC (\(\lambda(u, Vs, s''). \ \exists v. Vs=\text{Some } v \land v \in R \land s''=s')\))

116
proof -
from assms have
  \[simp\]: ss-stack s' ≠ []
  and \[simp\]: ss-stack s ≠ []
unfolding rel-def impl-defs
apply (auto)
done

from assms show ?thesis
unfolding get-pending-impl-def
apply (subst Let-def, subst Let-def)
apply (rule ASSERT-leI)
apply (auto simp: impl-defs gen-cond-def rel-def)
apply (auto simp: impl-defs gen-cond-def rel-def)
apply (split prod.split)
apply (rule lhs-step-If)
apply (rule le-supI1)
apply (simp add: pred-rel-simps no-pending-refine restr-rel-simps
                  RETURN-RES-refine-iff)
apply (rule lhs-step-bind, simp)
apply (simp split del: if-split)
apply (rename-tac v)
apply (case-tac v ∈ R)
apply (rule le-supI2)
apply (rule RETURN-SPEC-refine)
apply (auto simp: rel-def is-empty-stack-impl-def neq-Nil-conv)
apply (cases ss-stack s' simp)
apply (auto elim!: list-relE)
apply (rule rhs-step-bind-RES, blast)
apply (simp add: rel-def is-empty-stack-impl-def)
apply (cases ss-stack s' simp)
apply (auto elim: list-relE)
done
qed

lemma do-discover-refine[refine]:
\[ (s, s') ∈ rel; (u,u')∈Id; (v,v')∈Id; v' ∉ R \]
\[ \implies \text{do-discover } u \, v \, s \leq \Downarrow \text{rel } (a \cdot c \cdot \text{do-discover } u' \, v' \, s') \]

**unfolding** do-action-defs a.c.do-action-defs

**apply** refine-rcg

- **apply** (rule intro-prgR[where \( R=\text{rel} \)])
- **apply** (simp add: discover-impl-def a.c.discover-impl-def)
- **apply** (refine-rcg, auto simp: rel-def rel-restrict-def) []

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

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

**done**

**lemma** aux-R-node-discovered: \[ [(s,s') \in \text{rel}; \, v \in R] \implies \text{is-discovered-impl } v \, s \]

by (auto simp: pred-rel-simps)

**lemma** re-refine-aux: gen-dfs \leq \Downarrow \text{rel } a \cdot c \cdot \text{gen-dfs}

**unfolding** a.c.gen-dfs-def gen-dfs-def

**apply** (simp del: graph-restrict-simps)

**apply** (rule bind-refine)

**apply** (refine-rcg)

**apply** (erule WHILE-invisible-refine)

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

**apply** (frule (1) gen-cond-param[param-fo], THEN IdD, THEN iffD1])

**apply** (simp del: graph-restrict-simps)

**unfolding** gen-step-def

**apply** (simp del: graph-restrict-simps cong: if-cong option.case-cong split del: if-split)

**apply** (rule lhs-step-If)

**apply** (frule (1) pred-rel-simps[THEN iffD1])

**apply** (rule le-supI1)

**apply** (simp add: a.c.gen-step-def del: graph-restrict-simps)

**apply** refine-rcg

**apply** (auto simp: pred-rel-simps) [2]

**apply** (frule (1) pred-rel-simps[THEN Not-eq-iff[symmetric, THEN iffD1],

THEN iffD1])

**thm** order-trans[OF bind-mono(1)[OF get-pending-refine order-refl]]

**apply** (rule order-trans[OF bind-mono(1)[OF get-pending-refine order-refl]])

**apply** assumption+

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

apply (rule le-supI1)
  apply (rule bind-refine'[OF conc-fun-mono[THEN monoD]], simp)
  apply (clarsimp simp: refine-pw-simps)
  apply (refine-rcg, refine-dref-type, simp-all add: pred-rel-simps)

apply (rule le-supI2)
apply (rule RETURN-as-SPEC-refine)
apply (clarsimp simp add: conc-fun-SPEC)
apply (clarsimp)
apply (erule (1) aux-R-node-discovered, blast)
done

theorem re-refine-aux2: gen-dfs ≤⇓ (rel O (ES) simple-state-rel) a.a.it-dfs
proof -
  note re-refine-aux
  also note a.gen-dfs-refine
  finally show ?thesis by (simp add: conc-fun-chain del: graph-restrict-simps)
qed

theorem re-refine: gen-dfs ≤⇓ ((ES) restr-simple-state-rel R) a.a.it-dfs
unfolding restr-simple-state-rel-combine
by (rule re-refine-aux2)

sublocale tailrec-impl G gds
  unfolding restr-simple-state-rel
  apply unfold-locales
  apply (simp add: do-action-defs impl-defs[abs-def])
  apply (auto simp: pw-leof-iff refine-pw-simps split: prod.split)
done

lemma tailrec-refine: tailrec-impl ≤⇓ ((ES) restr-simple-state-rel R) a.a.it-dfs
proof -
  note tailrec-impl also note re-refine finally show ?thesis.
  qed

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 ex-
amples 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' ⇒ break s = break s'
by (cases s, cases s', simp)

**lemma** [simp]: s[] state.more := [] break = foo [] s [] break := foo 
by (cases s) simp

Define the parameterization. We start at a default parameterization, where all operations default to skip, and just add the operations we are interested in: Initially, the break flag is false, it is set if we encounter a back-edge, and once set, the algorithm shall terminate immediately.

**definition** cycc-params :: ('v,unit cycc-state-ext) parameterization
where cycc-params ≡ dfilt-parametrization state.more
  (RETURN [] break = False []) []
  on-back-edge := λ- -. RETURN [] break = True ],
  is-break := break []
**lemmas** cycc-params-simp[simp] =
gen-parameterization.simps[mk-record-simp, OF cycc-params-def[simplified]]

**interpretation** cycc: param-DFS-defs where param=cycc-params for G.

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

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

The total correct variant asserts finitely many reachable nodes.

**definition** cyc-checkerT G ≡ do {
  ASSERT (graph G ∧ finite (graph-defs.reachable G));
  s ← cycc.it-dfsT TYPE('a) G;
  RETURN (break s)
}

Next, we define a locale for the cyclicity checker’s precondition and invariant, by specializing the param-DFS locale.

**locale** cycc = param-DFS G cycc-params for G :: ('v, 'more) graph-rec-scheme
begin
We can easily show that our parametrization does not fail, thus we also get the DFS-locale, which gives us the correctness theorem for the DFS-scheme

**sublocale** DFS G cycc-params
  apply unfold-locales
  apply (simp-all add: cycc-params-def)
done
thm it-dfs-correct — Partial correctness
thm it-dfsT-correct — Total correctness if set of reachable states is finite
end

lemma cyccI:
  assumes fb-graph G
  shows cycc G
proof –
  interpret fb-graph G by fact
  show ?thesis by unfold-locales
qed

lemma cyccI':
  assumes graph G
  and FR: finite (graph-defsreachable G)
  shows cycc G
proof –
  interpret graph G by fact
  from FR interpret fb-graph G by (rule fb-graphI-fr)
  show ?thesis by unfold-locales
qed

Next, we specialize the DFS-invar locale to our parameterization. This
locale contains all proven invariants. When proving new invariants, this
locale is available as assumption, thus allowing us to re-use already proven
invariants.

locale cycc-invar = DFS-invar where
  param = cycc-params + cycc

The lemmas to establish invariants only provide the DFS-invar locale. This
lemma is used to convert it into the cycc-invar locale.

lemma cycc-invar-eq[simp]:
  shows DFS-invar G cycc-params s ⟷ cycc-invar G s
proof
  assume DFS-invar G cycc-params s
  interpret DFS-invar G cycc-params s by fact
  show cycc-invar G s by unfold-locales
next
  assume cycc-invar G s
  then interpret cycc-invar G s .
  show DFS-invar G cycc-params s by unfold-locales
qed

2.1.2 Correctness Proof

We now enter the cycc-invar locale, and show correctness of our cyclicity
checker.

context cycc-invar begin

123
We show that we break if and only if there are back edges. This is straightforward from our parameterization, and we can use the \textit{establish-invarI} rule provided by the DFS framework.

We use this example to illustrate the general proof scheme:

\textbf{lemma (in cycc) i-break-eq-back: is-invar (λs. break s ←→ back-edges s ≠ {})}

\textbf{proof (induct rule: establish-invarI)}

The \([\text{on-init cycc-params} ≤_n \text{SPEC (λx. ?I (empty-state x))}] \land s′ \in_{V0} [\text{DFS-invar G cycc-params s; ?I s; cond s;}\neg is-break cycc-params s; stack s = [] ]; v0 \in V0; v0 \notin\text{dom (discovered s)}; s′ = new-root v0 s] \implies on-new-root cycc-params v0 s′ \leq_n \text{SPEC (λx. DFS-invar G cycc-params (s′[state.more := x])}) \implies ?I (s′[state.more := x])]; \land s′ u. [\text{DFS-invar G cycc-params s; ?I s; cond s;}\neg is-break cycc-params s; stack s ≠ [] ]; u = hd (stack s); pending s = {} ]; s′ = finish u s] \implies on-finish cycc-params u s′ \leq_n \text{SPEC (λx. DFS-invar G cycc-params (s′[state.more := x])}) \implies \neg ?I (s′[state.more := x])]; \land s′ u v. [\text{DFS-invar G cycc-params s; ?I s; cond s;}\neg is-break cycc-params s; stack s ≠ [] ]; (u, v) ∈ pending s; u = hd (stack s); v ∈ dom (discovered s); s′ = cross-edge u v (s[pending := pending s − {(u, v)}]) \implies on-cross-edge cycc-params u v s′ \leq_n \text{SPEC (λx. DFS-invar G cycc-params (s′[state.more := x])}) \implies ?I (s′[state.more := x])]; \land s′ u v. [\text{DFS-invar G cycc-params s; ?I s; cond s;}\neg is-break cycc-params s; stack s ≠ [] ]; (u, v) ∈ pending s; u = hd (stack s); v ∈ dom (discovered s); s′ = back-edge u v (s[pending := pending s − {(u, v)}]) \implies on-back-edge cycc-params u v s′ \leq_n \text{SPEC (λx. DFS-invar G cycc-params (s′[state.more := x])}) \implies ?I (s′[state.more := x])]; \land s′ u v. [\text{DFS-invar G cycc-params s; ?I s; cond s;}\neg is-break cycc-params s; stack s ≠ [] ]; (u, v) ∈ pending s; u = hd (stack s); v ∈ dom (discovered s); s′ = discover u v (s[pending := pending s − {(u, v)}]) \implies on-discover cycc-params u v s′ \leq_n \text{SPEC (λx. DFS-invar G cycc-params (s′[state.more := x])}) \implies \neg ?I (s′[state.more := x])]) \implies is-invar ?I]

rule is used with the induction method, and yields cases

\textbf{print-cases}

Our parameterization has only hooked into initialization and back-edges, so only these two cases are non-trivial

\textbf{case init thus ?case by (simp add: empty-state-def)}

\textbf{next}

\textbf{case (back-edge s s′ u v)}

For proving invariant preservation, we may assume that the invariant holds on the previous state. Interpreting the invariant locale makes available all invariants ever proved into this locale (i.e., the invariants from all loaded libraries, and the ones you proved yourself.).

\textbf{then interpret cycc-invar G s by simp}

However, here we do not need them:

\textbf{from back-edge show ?case by simp}

\textbf{qed (simp-all cong: break-more-cong)}

For technical reasons, invariants are proved in the basic locale, and then transferred to the invariant locale:
lemmas brk-eq-back = i-brk-eq-back[THEN make-invar-thm]

The above lemma is simple enough to have a short apply-style proof:

lemma (in cycc) i-brk-eq-back-short-proof:
  is-invar (λs. break s ↔ back-edges s ≠ {})
  apply (induct rule: establish-invarI)
  apply (simp-all add: cond-def cong: break-more-cong)
  apply (simp add: empty-state-def)
  done

Now, when we know that the break flag indicates back-edges, we can easily prove correctness, using a lemma from the invariant library:

thm cycle-iff-back-edges
lemma cycc-correct-aux:
  assumes NC: ¬cond s
  shows break s ↔ ¬acyclic (E ∩ reachable × UNIV)
  proof (cases break s, simp-all)
    assume break s
    with brk-eq-back have back-edges s ≠ {} by simp
    with cycle-iff-back-edges have ¬acyclic (edges s) by simp
    with acyclic-subset[OF - edges-ss-reachable-edges]
    show ¬acyclic (E ∩ reachable × UNIV) by blast
  next
    assume A: ¬break s
    from A brk-eq-back have back-edges s = {} by simp
    with cycle-iff-back-edges have acyclic (edges s) by simp
    also from A nc-edges-covered[OF NC] have edges s = E ∩ reachable × UNIV
    by simp
    finally show acyclic (E ∩ reachable × UNIV) .
  qed

Again, we have a short two-line proof:

lemma cycc-correct-aux-short-proof:
  assumes NC: ¬cond s
  shows break s ↔ ¬acyclic (E ∩ reachable × UNIV)
  using nc-edges-covered[OF NC] brk-eq-back cycle-iff-back-edges
  by (auto dest: acyclic-subset[OF - edges-ss-reachable-edges])

end

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

definition cyc-checker-spec G ≡ do {
  ASSERT (fb-graph G);
  SPEC (λr. r ↔ ¬acyclic (g-E G ∩ ((g-E G)* ∘ g-V0 G) × UNIV))}
theorem cyc-checker-correct: cyc-checker $G \leq$ cyc-checker-spec $G$
unfolding cyc-checker-def cyc-checker-spec-def
proof (refine-vcg le-ASSERTI order-trans[OF DFS.it-dfs-correct], clarsimp-all)
  assume fb-graph $G$
  then interpret fb-graph $G$.
  interpret cycc by unfold-locales
  show DFS $G$ cycc-params by unfold-locales
next
  fix $s$
  assume cycc-invar $G$ $s$
  then interpret cycc-invar $G$ $s$.
  assume $\neg$cycc.cond TYPE($'b$) $G$ $s$
  thus break $s$ = ($\neg$ acyclic ($g$-E $G$ $\cap$ cycc.reachable TYPE($'b$) $G$ $\times$ UNIV))
    by (rule cycc-correct-aux)
qed

The same for the total correct variant:

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

theorem cyc-checkerT-correct: cyc-checkerT $G$ $\leq$ cyc-checkerT-spec $G$
unfolding cyc-checkerT-def cyc-checkerT-spec-def
proof (refine-vcg le-ASSERTI order-trans[OF DFS.it-dfsT-correct], clarsimp-all)
  assume graph $G$ then interpret graph $G$.
  assume finite (graph-defs.reachable $G$)
  then interpret fb-graph $G$ by (rule fb-graphI-fr)
  interpret cycc by unfold-locales
  show DFS $G$ cycc-params by unfold-locales
next
  fix $s$
  assume cycc-invar $G$ $s$
  then interpret cycc-invar $G$ $s$.
  assume $\neg$cycc.cond TYPE($'b$) $G$ $s$
  thus break $s$ = ($\neg$ acyclic ($g$-E $G$ $\cap$ cycc.reachable TYPE($'b$) $G$ $\times$ UNIV))
    by (rule cycc-correct-aux)
qed

2.1.3 Implementation
The implementation has two aspects: Structural implementation and data implementation. The framework provides recursive and tail-recursive implementations, as well as a variety of data structures for the state.

We will choose the simple-state implementation, which provides a stack, an on-stack and a visited set, but no timing information.

Note that it is common for state implementations to omit details from the very detailed abstract state. This means, that the algorithm’s operations
must not access these details (e.g. timing). However, the algorithm’s correctness proofs may still use them.

We extend the state template to add a break flag

```plaintext
record 'v cycc-state-impl = 'v simple-state +
  break :: bool
```

Definition of refinement relation: The break-flag is refined by identity.

```plaintext
definition cycc-erel \equiv \{ 
  (| cycc-state-impl.break = b |, | cycc-state.break = b|) | b. True \}
abbreviation cycc-rel \equiv (cycc-erel) simple-state-rel
```

Implementation of the parameters

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

Note: In this simple case, the reformulation of the extension state and parameterization is just redundant, However, in general the refinement will also affect the parameterization.

```plaintext
lemma break-impl: (si,s)\in cycc-rel 
  ==> cycc-state-impl.break si = cycc-state.break s 
by (cases si, cases s, simp add: simple-state-rel-def cycc-erel-def)
```

**interpretation** cycc-impl: simple-impl-defs G cycc-params-impl cycc-params

for G.

The above interpretation creates an iterative and a recursive implementation

term cycc-impl.tailrec-impl term cycc-impl.rec-impl

term cycc-impl.tailrec-implT — Note, for total correctness we currently only support tail-recursive implementations.

We use both to derive a tail-recursive and a recursive cyclicity checker:

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

definition [DFS-code-unfold]:
cyc-checker-implT G ≡ do {
    ASSERT (graph G ∧ finite (graph-defs.reachable G));
    s ← cyc-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 cyc begin
  sublocale simple-impl G cycc-params cycc-params-impl cycc-erel
  apply unfold-locales
  apply (intro fun-rel1, clarsimp simp: simple-state-rel-def, parametricity) []
  apply (auto simp: cycc-erel-def break-impl simple-state-rel-def)
  done

We get that our implementation refines the 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
cyc-checker-impl T TYPE('more) G ≤⇓ cycc-rel it-dfs
cyc-checker-impl rec-impl TYPE('more) G ≤⇓ cycc-rel it-dfs
  using impl-refine .

end

Finally, we get correctness of our cyclicity checker implementations

lemma cyc-checker-impl-refine: cyc-checker-impl G ≤⇓ Id (cyc-checker G)
  unfolding cyc-checker-impl-def cyc-checker-def
  apply (refine-vcg cycc.impl-refine)
  apply (simp-all add: break-impl cyccI)
  done

lemma cyc-checker-rec-impl-refine:
cyc-checker-rec-impl G ≤⇓ Id (cyc-checker G)
  unfolding cyc-checker-rec-impl-def cyc-checker-def
  apply (refine-vcg cycc.impl-refine)
  apply (simp-all add: break-impl cyccI)
  done

lemma cyc-checker-implT-refine:
cyc-checker-implT G ≤⇓ Id (cyc-checkerT G)
  unfolding cyc-checker-implT-def cyc-checkerT-def
  apply (refine-vcg cycc.impl-refine)
\end{verbatim}
apply (simp-all add: break-impl cyccI')
done

2.1.4 Synthesizing Executable Code

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

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

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

We define the refinement relation for the state extension

definition [to-relAPP]: cycc-state-erel erel ≡ {
  (break = bi, ..., m) | bi mi b m
  (bi,b)∈bool-rel ∧ (mi,m)∈erel
}

And register it with the Autoref tool:

consts
  i-cycc-state-ext :: interface ⇒ interface

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

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

lemma [autoref-rules]:
  fixes ns-rel vis-rel erel
defines R ≡ (ns-rel,vis-rel,⟨erel⟩cycc-state-erel) ss-impl-rel
shows
  (cycc-state-impl'-ext, cycc-state-impl-ext) ∈ bool-rel → erel → ⟨erel⟩cycc-state-erel
  (break-impl, cycc-state-impl,break) ∈ R → bool-rel
unfolding cycc-state-erel-def ss-impl-rel-def R-def
by auto

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

schematic-goal cyc-checker-impl:
defines V ≡ Id :: ('v × 'v::hashable) set
assumes [unfolded V-def,autoref-rules]:
  (Gi, G) ∈ (Rm, V) g-impl-rel-ext
notes [unfolded V-def,autoref-tyrel] =
  TYRELI[where R=(V) dflt-ahs-rel]
  TYRELI[where R=(V × (V)list-set-rel) ras-rel]
shows nres-of {?c::?c dres} ≤⇓?R (cyc-checker-impl G)
unfolding DFS-code-unfold
using [[autoref-trace-failed-id, goals-limit=1]]
apply (autoref-monadic (trace))
done
concrete-definition cyc-checker-code uses cyc-checker-impl
export-code cyc-checker-code checking SML

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

**theorem cyc-checker-code-correct:**
assumes 1: fb-graph $G$
assumes 2: ($Gi, G) \in (Rm, Id)g$-impl-rel-ext
assumes 4: cyc-checker-code $Gi = dRETURN x$
shows $x \leftrightarrow (\neg acyclic (g-E G \cap ((g-E G)* \cdot g-V0 G) \times UNIV))$

**proof**
- note cyc-checker-code.refine[OF 2]
  also note cyc-checker-impl-refine
  also note cyc-checker-correct
  finally show ?thesis using 1 4
unfolding cyc-checker-spec-def by auto
qed

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

**schematic-goal cyc-checker-rec-impl:**
defines $V \equiv Id :: (v \times v::hashable) set$
assumes [unfolded V-def, autoref-rules]:
  ($Gi, G) \in (Rm, V)g$-impl-rel-ext
notes [unfolded V-def, autoref-tyrel] =
  TYREL{where $R=V\times(V\times(V\times\text{list-set-rel}\times\text{ras-rel})$}
shows nres-of (?c::'c dres) \leq\Downarrow ?R (cyc-checker-rec-impl $G$)
unfolding DFS-code-unfold
using [[autoref-trace-failed-id, goals-limit=1]]
apply (autoref-monadic (trace))
done
concrete-definition cyc-checker-rec-code uses cyc-checker-rec-code-def
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: ($Gi, G) \in (Rm, Id)g$-impl-rel-ext
assumes 4: cyc-checker-rec-code $Gi = dRETURN x$
shows $x \leftrightarrow (\neg acyclic (g-E G \cap ((g-E G)* \cdot g-V0 G) \times UNIV))$

**proof**
- note cyc-checker-rec-code.refine[OF 2]
  also note cyc-checker-rec-impl-refine
  also note cyc-checker-correct
  finally show ?thesis using 1 4
And, again, for the total correct version. Note that we generate a plain
implementation, not inside a monad:

schematic-goal cyc-checker-implT:
defines $V \equiv \text{Id} :: (\forall v \cdot (v :: \text{hashable}) \cdot \text{set})$
assumes [unfolded $V$-def, autoref-rules]:
\[(G_i, G) \in \langle R_m, V \rangle \cdot \text{g-impl-rel-ext}\]
notes [unfolded $V$-def, autoref-tyrel] =
\[\text{TYRELI[where } R=(\forall \cdot \text{dflt-ahs-rel})\]
shows $\text{RETURN } (\exists \cdot ?c) \leq ?R (\text{cyc-checker-implT } G)$

unfolding DFS-code-unfold
using [[autoref-trace-failed-id, goals-limit=1]]
apply (autoref-monadic (trace, plain))
done

concrete-definition cyc-checker-codeT uses cyc-checker-implT
export-code cyc-checker-codeT checking SML

theorem cyc-checker-codeT-correct:
assumes 1: graph $G$ finite (graph-defsreachable $G$)
assumes 2: $(G_i, G) \in \langle R_m, \text{Id} \rangle \cdot \text{g-impl-rel-ext}$
shows cyc-checker-codeT $G_i \iff (\neg \text{acyclic } (g-E G \cap ((g-E G) \ast (\forall \cdot g-V \emptyset G) \times UNIV)))$

proof –
\[\text{note cyc-checker-codeT, refine[OF 2]}\]
\[\text{also note cyc-checker-implT-refine}\]
\[\text{also note cyc-checkerT-correct}\]
finally show $?\text{thesis using 1}$
unfolding cyc-checkerT-spec-def by auto
qed
end

2.2 Finding a Path between Nodes

theory DFS-Find-Path
imports
  ../DFS-Framework
  CAVA-Automata.Digraph-Impl
  ../Misc/Impl-Rev-Array-Stack
begin

We instantiate the DFS framework to find a path to some reachable node
that satisfies a given predicate. We present four variants of the algorithm:
Finding any path, and finding path of at least length one, combined with
searching the whole graph, and searching the graph restricted to a given
set of nodes. The restricted variants are efficiently implemented by pre-initializing the visited set (cf. DFS-Framework.Restr-Impl).

The restricted variants can be used for incremental search, ignoring already searched nodes in further searches. This is required, e.g., for the inner search of nested DFS (Buchi automaton emptiness check).

### 2.2.1 Including empty Path

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

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

**lemma** [simp]: \( s | \text{state}.\text{more} := ( ) \text{ppath} = \text{foo} ) ) = s | \text{ppath} := \text{foo} ) 

**by** (cases \( s \)) simp

**abbreviation** \( \text{no-path} \equiv ( ) \text{ppath} = \text{None} ) \)

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

**definition** \( \text{fp0-params} :: ( 'v \Rightarrow \text{bool} ) \Rightarrow 'v \text{fp0-param} \)

where \( \text{fp0-params} \, P \equiv ( ) \)

- **on-init** = \( \text{RETURN} \, \text{no-path} \),
- **on-new-root** = \( \lambda v0 \, s. \, \text{if} \, P \, v0 \, \text{then} \, \text{RETURN} \, (\text{a-path} \, [ ] \, v0) \, \text{else} \, \text{RETURN} \, \text{no-path} \),
- **on-discover** = \( \lambda u \, v \, s. \, \text{if} \, P \, v \, \text{then} \, \text{—} \, v \, \text{is already on the stack, so we need to pop it again} \, \text{RETURN} \, (\text{a-path} \, (\text{rev} \, (\text{tl} \, (\text{stack} \, s))) \, v) \, \text{else} \, \text{RETURN} \, \text{no-path} \),
- **on-finish** = \( \lambda u \, s. \, \text{RETURN} \, (\text{state}.\text{more} \, s) \),
- **on-back-edge** = \( \lambda u \, v \, s. \, \text{RETURN} \, (\text{state}.\text{more} \, s) \),
- **on-cross-edge** = \( \lambda u \, v \, s. \, \text{RETURN} \, (\text{state}.\text{more} \, s) \),
- **is-break** = \( \lambda s. \, \text{ppath} \, s \neq \text{None} ) \)

**lemmas** \( \text{fp0-params-simps[simp]} \)

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

**interpretation** \( \text{fp0}: \text{param-DFS-defs} \)

where \( \text{param} = \text{fp0-params} \, P \) for \( G \, P \).

**locale** \( \text{fp0} = \text{param-DFS} \, G \, \text{fp0-params} \, P \)

for \( G \) and \( P :: ( 'v \Rightarrow \text{bool} ) \)

begin

**lemma** [simp]:

- \( \text{ppath} \, (\text{empty-state} \, (\text{ppath} \, e)) = e \)

  **by** (simp add: empty-state-def)

**lemma** [simp]:

- \( \text{ppath} \, (s | \text{state}.\text{more} := \text{state}.\text{more} \, s') = \text{ppath} \, s' \)

132
by (cases s, cases s') auto

sublocale DFS where param = fp0-params P 
  by unfold-locales simp-all

end

lemma fp0I: assumes fb-graph G shows fp0 G 
proof - interpret fb-graph G by fact show ?thesis by unfold-locales qed

locale fp0-invar = fp0 + DFS-invar where param = fp0-params P

lemma fp0-invar-eq[simp]:
  DFS-invar G (fp0-params P) = fp0-invar G P
proof (intro ext iffI)
  fix s
  assume DFS-invar G (fp0-params P) s
  interpret DFS-invar G fp0-params P s by fact
  show fp0-invar G P s by unfold-locales
next
  fix s
  assume fp0-invar G P s
  interpret fp0-invar G P s by fact
  show DFS-invar G (fp0-params P) s by unfold-locales
qed

context fp0 begin

lemma i-no-path-no-P-discovered:
  is-invar (\lambda s. ppath s = None \rightarrow dom (discovered s) \cap Collect P = {})
by (rule establish-invarI) simp-all

lemma i-path-to-P:
  is-invar (\lambda s. ppath s = Some (vs,v) \rightarrow P v)
by (rule establish-invarI) auto

lemma i-path-invar:
  is-invar (\lambda s. ppath s = Some (vs,v) \rightarrow
                                   (vs \neq [] \rightarrow hd vs \in V0 \land path E (hd vs) vs v)
                                   \land (vs = [] \rightarrow v \in V0 \land path E v vs v)
                                   \land (distinct (vs@[v])))
proof (induct rule: establish-invarI)
  case (discover s s' u v) then interpret fp0-invar where s=s
  by simp

from discover have ne: stack s \neq [] by simp
from discover have vnis: v \notin set (stack s) using stack-discovered by auto

end
from pendingD discover have \( v \in \text{succ} (\text{hd} (\text{stack} s)) \) by simp
with \( \text{hd}-\text{succ-\text{is-path}} \) OF \( \text{ne} \)
have \( \exists v0 \in V0. \; \text{path} E v0 \; (\text{rev} (\text{stack} s)) \; v \).
morerover from last-stack-in-V0 \( \text{ne} \)
have \( \text{last} (\text{stack} s) \in V0 \) by simp
ultimately have \( \text{path} E (\text{hd} (\text{rev} (\text{stack} s))) \; \text{rev} (\text{stack} s) \; v \; \text{hd} (\text{rev} (\text{stack} s)) \) \( \in V0 \)
using \( \text{hd-rev} \) OF \( \text{ne} \)
with \( \text{ne} \) discover \( \text{vnis} \) show \( ?\text{case} \) by (auto simp: stack-distinct)
qued auto
e nd

c ontect fp0-invar
b egin
le mmas no-path-no-P-discovered
= i-no-path-no-P-discovered[THEN make-invar-thm, rule-format]
le mmas path-to-P
= i-path-to-P[THEN make-invar-thm, rule-format]
le mmas path-invar
= i-path-invar[THEN make-invar-thm, rule-format]
le mm a path-invar-nonempty:
ass umes \( \text{ppath} s = \text{Some} (\text{vs},v) \)
and \( \text{vs} \neq [] \)
shows \( \text{hd} \; \text{vs} \in V0 \; \text{path} E (\text{hd} \; \text{vs}) \; \text{vs} \; v \)
using assms path-invar
by auto
le mm a path-invar-empty:
ass umes \( \text{ppath} s = \text{Some} (\text{vs},v) \)
and \( \text{vs} = [] \)
shows \( v \in V0 \; \text{path} E v \; \text{vs} \; v \)
using assms path-invar
by auto
le mm a fp0-correct:
ass umes \( \neg \text{cond} s \)
s hows case ppath s of
None \( \Rightarrow \neg(\exists v0 \in V0. \exists v. \; (v0,v) \in E^* \; \land P \; v) \)
| Some (p,v)\( \Rightarrow (\exists v0 \in V0. \; \text{path} E v0 \; p \; v \; \land P \; v \; \land \text{distinct} (p\@[v])) \)
proof (cases ppath s)
c ase None with assms ne-discovered-eq-reachable no-path-no-P-discovered
 reachable \( \cap \) Collect \( P = \{\} \) by auto
thus \( ?\text{thesis} \) by (auto simp add: None)
next
c ase \( \text{Some} \; \text{vvs} \) then obtain \( v \) \( \text{vs} \) where [simp]: \( \text{vvs} = (\text{vs},v) \)
by (cases vs) auto

from Some path-invar[of vs v] path-to-P[of - v] show ?thesis
by auto
qed

end

context fp0 begin

lemma fp0-correct: it-dfs ≤ SPEC (λs. case ppath s of 
   None ⇒ ¬(∃v0∈V0. ∃v. (v0,v) ∈ E∗ ∧ P v) 
   | Some (p,v) ⇒ (∃v0∈V0. path E v0 v ∧ P v ∧ distinct (p@[v])))
apply (rule weaken-SPEC[OF it-dfs-correct])
apply clarsimp
apply (simp add: fp0-invar
   fp0-correct)
done
end

Basic Interface

Use this interface, rather than the internal stuff above!

type-synonym 'v fp-result = ('v list × 'v) option

definition find-path0-pred G P ≡ λr. case r of 
   None ⇒ (g-E G)∗ "g-V0 G ∩ Collect P = {} 
   | Some (vs,v) ⇒ P v ∧ distinct (vs@[v]) ∧ (∃ v0 ∈ g-V0 G. path (g-E G) v0 vs v)

definition find-path0-spec :: ('v, -) graph-rec-scheme ⇒ ('v ⇒ bool) ⇒ 'v fp-result nres
— Searches a path from the root nodes to some target node that satisfies a given
   predicate. If such a path is found, the path and the target node are returned
where
find-path0-spec G P ≡ do 
   ASSERT (fb-graph G);
   SPEC (find-path0-pred G P)
}

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

lemma find-path0-correct:
shows find-path0 G P ≤ find-path0-spec G P
unfolding find-path0-def find-path0-spec-def find-path0-pred-def
apply (refine-vcg le-ASSERTI order-trans[OF fp0 fp0-correct])
apply (erule fp0I)
apply (auto split: option.split) []
done

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

2.2.2 Restricting the Graph

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

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

where
restr-invar E R P ≡ E ∩ R ∩ Collect P = {}

lemma restr-invar-triv[simp, intro!]: restr-invar E {} P
unfolding restr-invar-def by simp

lemma restr-invar-imp-not-reachable: restr-invar E R P ⇒ E * R ∩ Collect P = {}
unfolding restr-invar-def by (simp add: Image-closed-trancl)


type-synonym 'v fpr-result = 'v set + ('v list × 'v)
definition find-path0-restr-pred G P R ≡ λr.
case r of
  Inl R' ⇒ R' = R ∪ (g-E G) * R ∩ restr-invar (g-E G) R' P
  | Inr (vs,v) ⇒ P v ∧ (∃ v0 ∈ g-V0 G - R. path (rel-restrict (g-E G) R) v0 vs v)

definition find-path0-restr-spec
— Find a path to a target node that satisfies a predicate, not considering nodes
from the given node restriction. If no path is found, an extended restriction is
returned, that contains the start nodes

where find-path0-restr-spec G P R ≡ do {
  ASSERT (fb-graph G ∧ restr-invar (g-E G) R P);
  SPEC (find-path0-restr-pred G P R))

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

definition find-path0-restr
:: ('v, 'more) graph-rec-scheme ⇒ ('v ⇒ bool) ⇒ 'v fpr-result nres
where find-path0-restr G P R ≡ do {
  ASSERT (fb-graph G);
  ASSERT (fp0 (graph-restrict G R));
\[ s \leftarrow \text{fp0.it-dfs}\text{ TYPE('more) (graph-restrict G R) P; } \]

\[ \text{case ppath s of} \]

\[ \text{None } \Rightarrow \text{ do } \}

\[ \text{ASSERT } (\text{dom (discovered s) } = \text{ dom (finished s)}); \]

\[ \text{RETURN } (\text{Inl } (R \cup \text{ dom (finished s)})) \]

\[ | \text{Some } (vs,v) \Rightarrow \text{ RETURN } (\text{Inr } (vs,v)) \]

\[ \text{lemma find-path0-restr-correct: } \]

\[ \text{shows find-path0-restr G P R } \leq \text{ find-path0-restr-spec G P R} \]

\[ \text{proof (rule } \text{le-ASSERT-defI1}[\text{OF find-path0-restr-spec-def}], \text{ clarify)} \]

\[ \text{assume fb-graph G } \]

\[ \text{interpret a: fb-graph G by fact} \]

\[ \text{interpret fb-graph graph-restrict G R by (rule a.fb-graph-restrict)} \]

\[ \text{assume } I: \text{ restr-invar (g-E G) R P} \]

\[ \text{define reachable where reachable } = \text{ graph-defs.reachable (graph-restrict G R)} \]

\[ \text{interpret fp0 graph-restrict G R by unfold-locales} \]

\[ \text{show } \text{thesis unfolding find-path0-restr-def find-path0-restr-spec-def} \]

\[ \text{apply (refine-reg refine-veg le-ASSERTI order-trans[OF it-dfs-correct])} \]

\[ \text{apply unfold-locales} \]

\[ \text{apply (clarsimp-all)} \]

\[ \text{proof } – \]

\[ \text{fix } s \]

\[ \text{assume } \text{fp0-invar (graph-restrict G R) P s} \]

\[ \text{and NC[simp]: } \neg \text{fp0.cond TYPE('b) (graph-restrict G R) P s} \]

\[ \text{then interpret fp0-invar graph-restrict G R P s by simp} \]

\{ \]

\[ \text{assume [simp]: ppath s = None} \]

\[ \text{from nc-discovered-eq-finished} \]

\[ \text{show dom (discovered s) } = \text{ dom (finished s) by simp} \]

\[ \text{from nc-finished-eq-reachable} \]

\[ \text{have DFR[simp]: dom (finished s) = reachable} \]

\[ \text{by (simp add: reachable-def)} \]

\[ \text{from I have g-E G } \subseteq \text{ R unfolding restr-invar-def by auto} \]

\[ \text{have reachable } \subseteq \text{ (g-E G) ' ' g-V0 G} \]

\[ \text{unfolding reachable-def} \]

\[ \text{by (rule Image-mono, rule rtrancl-mono) (auto simp: rel-restrict-def)} \]

137
hence \( R \cup \text{dom} \ (\text{finished } s) = R \cup (g\cdot E \ G)^+ \ \overset{\text{g-V0 } G}{=} \ \text{g-V0 } G \)

apply –
apply \(\text{rule equalityI}\)
apply \(\text{auto} [\]
unfolding \(\text{DFR reachable-def}\)
apply \(\text{auto elim: E-closed-restr-cases}[\text{OF} - (g\cdot E \ G \ \overset{\text{R } \subseteq R}{=} \ ]]
done
moreover from \(\text{nc-fin-closed I}\)
have \(g\cdot E \ G \ '' (R \cup \text{dom} \ (\text{finished } s)) \subseteq R \cup \text{dom} \ (\text{finished } s)\)
unfolding \(\text{restr-invar-def by (simp add: rel-restrict-def)}\) blast
moreover from \(\text{no-path-no-P-discovered nc-discovered-eq-finished I}\)
have \( (R \cup \text{dom} \ (\text{finished } s)) \cap \text{Collect } P = \{\} \)
unfolding \(\text{restr-invar-def by auto}\)
ultimately
show \(\text{find-path0-restr-pred } G \ P \ R \ (\text{Inl } (R \cup \text{dom} \ (\text{finished } s)))\)
unfolding \(\text{restr-invar-def find-path0-restr-pred-def by auto}\)

\}\{ fix \ v \ vs
assume \[\text{simp]: ppath } s = \text{Some } (vs,v)\)
from \(\text{fp0-correct}\)
show \(\text{find-path0-restr-pred } G \ P \ R \ (\text{Inr } (vs, v))\)
unfolding \(\text{find-path0-restr-pred-def by auto}\)
\}

qed

2.2.3 Path of Minimal Length One, with Restriction

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

— 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 \(\text{find-path1-restr-spec } G \ P \ R \equiv \text{do } \{
\text{ASSERT } (\text{fb-graph } G \ \land \text{restr-invar } (g\cdot E \ G) \ R \ P);\n\text{SPEC } (\text{find-path1-restr-pred } G \ P \ R)\}\)

lemmas \(\text{find-path1-restr-spec-rule[refine-vcg]} =
\text{ASSERT-le-defI}[\text{OF } \text{find-path1-restr-spec-def}]\)
\text{ASSERT-leof-defI}[\text{OF } \text{find-path1-restr-spec-def}]\)
definition \(\text{find-path1-restr}\)
\(:= (\text{'}v, \text{'}more) \ \text{graph-rec-scheme } \Rightarrow (\text{'}v \Rightarrow \text{bool}) \Rightarrow \text{'}v \ \text{set } \Rightarrow \text{'}v \ \text{fpr-result nres}\)
where \( \text{find-path1-restr } G \ P \ R \equiv \)

\[ \text{FOREACH}_{c} (g-V0 \ G) \text{ is-Inl } (\lambda v0 \ s. \ do \{ \text{ASSERT } (\text{is-Inl } s); \text{ — TODO: Add FOREACH-condition as precondition in autorefer!} \)} \]

\[
\text{let } R \equiv \text{projl } s; \\
f0 \leftarrow \text{find-path0-restr-spec } (G \ G | g-V0 := g-E \ G " \{v0\} \ ) \ P \ R; \\
\text{case } f0 \text{ of} \\
\text{Inl } \Rightarrow \text{RETURN } f0 \\
| \text{Inr } (vs,v) \Rightarrow \text{RETURN } (\text{Inr } (v0 \# vs,v)) \\
\} \ (\text{Inl } R) \\
\]

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

\[
\text{lemma } \text{find-path1-restr-correct}; \\
\text{shows } \text{find-path1-restr } G \ P \ R \leq \text{find-path1-restr-spec } G \ P \ R \\
\text{proof } (\text{rule le-ASSERT-defII[OF find-path1-restr-spec-def]}, \text{clarify}) \\
\text{assume } \text{fb-graph } G \\
\text{interpret } a: \text{fb-graph } G \text{ by fact} \\
\text{interpret } f0: \text{fb-graph } (G | g-E := g-E \ G \cap \text{UNIV} \times -R0) \ \text{by } (\text{rule a.fb-graph-subset}, \text{auto}) \\
\text{assume } I: \text{restr-invar } (g-E \ G) \ R \ P \\
\text{have } aux2: \forall v0. \ v0 \in g-V0 \ G \Rightarrow \text{fb-graph } (G | g-V0 := g-E \ G " \{v0\} \ ) \\
\text{by } (\text{rule a.fb-graph-subset}, \text{auto}) \\
\} \\
\text{fix } v0 \ it \ s \\
\text{assume } IT: \ it \subseteq g-V0 \ G \ v0 \in it \\
\text{and is-Inl } s \\
\text{and FPI: find-path1-tailrec-invar } G \ P \ R \ it \ s \\
\text{and RI: restr-invar } (g-E \ G) \ (\text{projl } s \cup (g-E \ G)^{+} " \{v0\}) \ P \\
\text{then obtain } R' \text{ where } [\text{simp}]: s = \text{Inl } R' \text{ by } (\text{cases } s) \text{ auto} \\
\text{from FPI have } [\text{simp}]: R' = R \cup (g-E \ G)^{+} " (g-V0 \ G - it) \\
\text{unfolding find-path1-tailrec-invar-def by simp} \\
\text{have } \text{find-path1-tailrec-invar } G \ P \ R \ (it - \{v0\}) \\
\text{ (Inl } (\text{projl } s \cup (g-E \ G)^{+} " \{v0\})) \\
\text{using RI} \\
\text{by } (\text{auto simp: find-path1-tailrec-invar-def it-step-insert iff[OF IT']}) \\
\} \text{ note aux4 = this} \
\]

139
\{ 
  \textbf{fix } v_0 \ u \ s \ v \ p \\
  \textbf{assume } IT: \text{ it } \subseteq \text{ g-V0 G v0 } \in \text{ it} \\
  \text{and } \text{is-Inl } s \\
  \text{and } FPI: \text{ find-path1-tailrec-invar } G \ P \ R \ \text{ it } s \\
  \text{and } PV: \text{ P v} \\
  \text{and } PATH: \text{ path } (\text{ rel-restrict } (\text{ g-E G }) (\text{ projl s })) \ u \ p \ v \ (v_0,u) \in (\text{ g-E G }) \\
  \text{and } PR: \ u \notin \text{ projl } s \\

  \text{then obtain } R' \ \text{ where } [\text{simp}]: \ s = \text{ Inl } R' \ \text{ by (cases } s \text{ auto) }

  \text{from FPI have } [\text{simp}]: \ R' = R \cup (\text{ g-E G })^+ " (g-V0 G - \text{ it}) \\
  \text{unfolding } \text{ find-path1-tailrec-invar-def by simp}

  \text{have } \text{find-path1-tailrec-invar } G \ P \ R \ (\text{ it } \setminus \{v_0\}) (\text{ Inr } (v_0 \neq p, v)) \\
  \text{apply } (\text{ simp add: find-path1-tailrec-invar-def PV }) \\
  \text{apply } (\text{ rule bexI[where } x=v_0]) \\
  \text{using } PR \ PATH(2) \ \text{ path-mono OF rel-restrict-mono2[of } R \text{] PATH(1)]} \\
  \text{apply } (\text{ auto simp: path1-restr-conv }) \]

  \text{using } IT \ \text{apply blast}

\} \text{ note aux5 } = \text{ this}

\text{show } \text{thesis} \\
\text{unfolding find-path1-restr-def find-path1-restr-spec-def find-path1-restr-pred-def} \\
\text{apply } (\text{ refine-req le-ASSERTI}) \\
\text{refine-vcg FOREACHc-rule[where } I=\text{find-path1-tailrec-invar } G \ P \ R]\)

\} \text{ apply simp} \\
\text{using } I \ \text{apply } (\text{ auto simp add: find-path1-tailrec-invar-def restr-invar-def}) \]

\text{apply (blast intro: aux2)}
\text{apply } (\text{ auto simp add: find-path1-tailrec-invar-def split: sum.splits }) \]
\text{apply } (\text{ auto simp: find-path0-restr-pred-def aux4 aux5 simp: trancl-Image-unfold-left[symmetric] split: sum.splits }) \]

\text{apply } (\text{ auto simp add: find-path1-tailrec-invar-def split: sum.splits }) [2]
\text{done}
\text{qed}

\text{definition find-path1-pred } G \ P \equiv \lambda r. \\
\text{case } r \text{ of } \\
\ None \Rightarrow (g-E G)^+ " g-V0 G \cap \text{ Collect } P = \{\} \\
| \text{ Some } (vs, v) \Rightarrow P \ v \land vs \neq \[] \\
\ \land (\exists v_0 \in g-V0 G. \text{ path } (g-E G) v_0 \ vs \ v)

\text{definition find-path1-spec}

\}

140
— Find a path of length at least one to a target node that satisfies a given predicate.

where find-path1-spec G P ≡ do {
    ASSERT (fb-graph G);
    SPEC (find-path1-pred G P)}

lemmas find-path1-spec-rule [refine-vcg]
= ASSERT-le-defI [OF find-path1-spec-def]
  ASSERT-leof-defI [OF find-path1-spec-def]

2.2.4 Path of Minimal Length One, without Restriction

definition find-path1 :: ('v, 'more) graph-rec-scheme ⇒ ('v ⇒ bool) ⇒ 'v fp-result nres
where find-path1 G P ≡ do {
    r ← find-path1-restr-spec G P {}
    case r of
        Inl - ⇒ RETURN None
        | Inr vsv ⇒ RETURN (Some vsv)
}

lemma find-path1-correct:
shows find-path1 G P ≤ find-path1-spec G P
unfolding find-path1-def find-path1-spec-def find-path1-pred-def
apply (refine-reg refine-vcg le-ASSERTI order-trans[OF find-path1-restr-correct])
apply simp
apply (fastforce simp: find-path1-restr-spec-def find-path1-restr-pred-def
        split: sum.splits
        dest!: restr-invar-imp-not-reachable tranclD)
done

2.2.5 Implementation

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

definition fp0-erel ≡ 
    (fp0-state-impl.ppath = p, (fp0-state.ppath = p)) | p. True }

abbreviation fp0-rel R ≡ (fp0-erel)restr-simple-state-rel R
abbreviation no-path-impl ≡ (fp0-state-impl.ppath = None )
abbreviation a-path-impl p v ≡ (fp0-state-impl.ppath = Some (p,v ) )

lemma fp0-rel-ppath-cong [simp]:
(s,s') ∈ fp0-rel R ⇒ fp0-state-impl.ppath s = fp0-state.ppath s'
unfolding restr-simple-state-rel-def fp0-erel-def
by (cases s, cases s', auto)
lemma fp0-ss-rel-ppath-cong[simp]:
(s,s') ∈ ⟨fp0-erel⟩ ⟹ simple-state-rel ⇒ fp0-state-impl.ppath s = fp0-state.ppath s'
unfolding simple-state-rel-def fp0-erel-def by (cases s, cases s', auto)

lemma fp0i-cong[cong]: simple-state.more s = simple-state.more s'
⇒ fp0-state-impl.ppath s = fp0-state-impl.ppath s'
by (cases s, cases s', auto)

lemma fp0-erelI: p=p'
⇒ ((| fp0-state-impl.ppath = p |), (| fp0-state.ppath = p' |)) ∈ fp0-erel
unfolding fp0-erel-def by auto

definition fp0-params-impl
:: - ⇒ ('v, 'v fp0-state-impl, ('v, unit)fp0-state-impl-ext) gen-parameterization
where
fp0-params-impl P ≡ ()
on-init = RETURN no-path-impl,
on-new-root = λ v0 s.
   if P v0 then RETURN (a-path-impl [] v0) else RETURN no-path-impl,
on-discover = λ u v s.
   if P v then RETURN (a-path-impl (map fst (rev (tl (CAST (ss-stack s)))))) v)
   else RETURN no-path-impl,
on-finish = λ u s. RETURN (simple-state.more s),
on-back-edge = λ u v s. RETURN (simple-state.more s),
on-cross-edge = λ u v s. RETURN (simple-state.more s),
is-break = λ s. ppath s ≠ None)

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

interpretation fp0-impl:
restricted-impl-defs fp0-params-impl P fp0-params P G R
for G P R.
locale fp0-restr = fb-graph
begin
sublocale fp0 graph-restrict G R
apply (rule fp0I)
apply (rule fb-graph-restrict)
done

sublocale impl: restricted-impl G fp0-params P fp0-params-impl P
fp0-erel R
apply unfold-locales
apply parametricity
   apply (simp add: fp0-erel-def)
apply (auto) [1]
apply (auto simp: rev-map[symmetric] map-tl comp-def
  simp: fp0-restr-def simple-state-rel-def) [7]

apply (auto simp: restr-rel-def) [3]
apply (clarsimp simp: restr-rel-def)
apply (rule IdD) apply (subst list-id-simp[symmetric])
apply parametricity
done
end

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

lemma find-path0-restr-impl[refine];
  shows find-path0-restr-impl \ G P R ≤⇓
    (⟨Id,Id×rId⟩ sum-rel) (find-path0-restr G P R)
proof (rule refine-ASSERT-defI2[OF find-path0-restr-def])
  assume fb-graph G
  then interpret fb-graph G .
  interpret fp0-restr G by unfold-locales

  show ?thesis
  unfolding find-path0-restr-impl-def find-path0-restr-def
  apply (refine-reg impl.tailrec-refine)
  apply refine-dref-type
  apply (auto simp: restr-simple-state-rel-def)
  done
qed

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

lemma find-path0-impl[refine]; find-path0-impl \ G P ≤ ⇓ (⟨Id×r,Id⟩ option-rel) (find-path0 G P)
proof (rule refine-ASSERT-defI1[OF find-path0-def])
  assume fp0 G
  then interpret fp0 G .
interpret \( r \): \( \text{fp0-restr} \) G by unfold-locales

show \(?\)thesis
  unfolding \text{find-path0-impl-def find-path0-def}
  apply (refine-reg \ r.\impl.tail-rec-refine[where \( R=\{\}, \) simplified])
  apply (auto)
done

qed

2.2.6 Synthesis of Executable Code

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

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

consts
  \( i\text{-fp0-state-ext} :: \text{interface} \Rightarrow \text{interface} \)

lemmas [\text{autoref-rel-intf}] = \text{REL-INTFI[of \text{fp0-state-erel i-fp0-state-ext]}

term \( \text{fp0-state-impl-ext} \)

lemma [\text{autoref-rules}]:
  fixes \( \text{ns-rel vis-rel erel} \)
  defines \( R \equiv \langle \text{ns-rel, vis-rel, (erel)}\text{fp0-state-erel}\rangle \text{ssnos-impl-rel} \)
  shows
    \( (\text{fp0-state-impl'-ext}, \text{fp0-state-impl-ext}) \in (\langle \text{Id} \rangle \text{list-rel} \times_r \text{Id}) \text{option-rel} \rightarrow \text{erel} \rightarrow (\langle \text{erel})\text{fp0-state-erel} \)
    \( (\text{ppath-impl, fp0-state-impl.ppath}) \in R \rightarrow (\langle \text{Id} \rangle \text{list-rel} \times_r \text{Id}) \text{option-rel} \)
  unfolding \( \text{fp0-state-erel-def ssnos-impl-rel-def R-def} \)
  by auto

schematic-goal \text{find-path0-code:}
  fixes \( G :: ('v :: \text{hashable}, -) \text{ graph-rec-scheme} \)
  assumes [\text{autoref-rules}]:
    \( (Gi, G) \in (\text{Rm, Id})\text{g-impl-rel-ext} \)
    \( (Pi, \text{P}) \in \text{Id} \rightarrow \text{bool-rel} \)
  notes [\text{autoref-tyrel}] = \text{TYRELI[where \( R=\langle \text{Id;('v\times'v) set})\text{dflt-abs-rel] \)
    shows [\text{nres-of (\?c::\?c dres)}]: \text{find-path0-impl G P} \in \text{\?R} \)
  unfolding \text{find-path0-impl-def abs-def} \text{ DFS-code-unfold ssnos-unfolds}
  unfolding \text{if-cancel not-not comp-def nres-monad-laws}
  using [[\text{autoref-trace-failed-id}]]
  apply (autoref-monadic (\text{trace}))-)
done

concrete-definition \text{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 (λG P. nres-of (find-path0-code G P), find-path0-spec)
                    ∈ ⟨Rm, Rv⟩g-impl-rel-ext → (Rv → bool-rel)
                  → ⟨⟨⟨Rv⟩⟩list-rel ×, Rv⟩option-rel⟩nres-rel
  apply (intro fun-relI nres-relI)
  unfolding Vid
  apply (rule
        order-trans[OF find-path0-code.refine[par
        assumption+
      )
  using find-path0-impl find-path0-correct
  apply (simp add: pw-le-iff refine-pw-simps)
  apply blast
  done

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

schematic-goal find-path0-restr-code:
  fixes vis-rel :: (′v × ′v) set ⇒ (′visi × ′v set) set
  notes [autoref-rel-intf] = REL-INTFI[of vis-rel i-set for I]
  assumes [autoref-rules]: (op-vis-insert, insert)∈Id → ⟨Id⟩vis-rel → ⟨Id⟩vis-rel
  assumes [autoref-rules]: (op-vis-memb, (∈))∈Id → ⟨Id⟩vis-rel → bool-rel
  assumes [autoref-rules]:
    (Gi, G) ∈ ⟨Rm, Id⟩g-impl-rel-ext
    (Pi, P)∈Id → bool-rel
    (Ri, R)∈⟨Id⟩vis-rel
  shows (nres-of (?:: (?′c dres)),
        find-path0-restr-impl
        G
        P
        ⟨R::=(Id)vis-rel⟩) ∈ ?R
  unfolding find-path0-restr-impl-def[abs-def] DFS-code-unfold ssnos-unfolds
  unfolding if-cancel not-not comp-def nres-monad-laws
  using [[autoref-trace-failed-id]]
  apply (autoref-monadic (trace))
  done

concrete-definition find-path0-restr-code uses find-path0-restr-code

export-code find-path0-restr-code checking SML

lemma find-path0-restr-autoref-aux:
  assumes 1: (op-vis-insert, insert)∈Rv → ⟨Rv⟩vis-rel → ⟨Rv⟩vis-rel
  assumes 2: (op-vis-memb, (∈))∈Rv → ⟨Rv⟩vis-rel → bool-rel

145
assumes \( \text{Vid}: \text{Rv} = \text{Id} \)

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

\[
\text{find-path0-restr-spec} \\
\in (\text{Rm}, \text{Rv}) \text{g-impl-rel-ext} \to (\text{Rv} \to \text{bool-rel}) \to (\text{Rv}) \text{vis-rel} \to \\
((Rv) \text{vis-rel}, (Rv) \text{list-rel} \times_{r} \text{Rv}) \text{sum-rel} \text{nres-rel}
\]

apply \( (\text{intro fun-relI nres-relI}) \)

unfolding \( \text{Vid} \)

apply \( (\text{rule order-trans:OF find-path0-restr-code.refine:OF 1[unfolded Vid]} 2[unfolded Vid], \text{param-fo, THEN nres-relD]) \)

apply assumption+

using \( \text{find-path0-restr-impl find-path0-restr-correct} \)

apply \( (\text{simp add: pu-le-iff refine-pw-simps}) \)

apply blast
done

lemmas \( \text{find-path0-restr-autoref[autoref-rules]} = \text{find-path0-restr-autoref-aux[OF GEN-OP-D GEN-OP-D PREFER-id-D]} \)

schematic-goal \( \text{find-path1-restr-code}: \)

fixes \( \text{vis-rel :: ('v x 'v) set} \Rightarrow ('visi x 'v set) set \)

notes \( [\text{autoref-rel-intf}] = \text{REL-INTFI[of vis-rel i-set for I]} \)

assumes \( [\text{autoref-rules}]: (\text{op-vis-insert}, \text{insert}) \in \text{Id} \to (\text{Id}) \text{vis-rel} \to (\text{Id}) \text{vis-rel} \)

assumes \( [\text{autoref-rules}]: (\text{op-vis-memb}, (\in)) \in \text{Id} \to (\text{Id}) \text{vis-rel} \to \text{bool-rel} \)

assumes \( [\text{autoref-rules}]: \)

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

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

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

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

\( \in (\langle\text{Id}\rangle \text{vis-rel}, \langle\text{Id}\rangle \text{list-rel} \times_{r} \text{Id}) \text{sum-rel} \text{nres-rel} \)

unfolding \( \text{find-path1-restr-def[abs-def]} \)

using \( [[\text{autoref-trace-failed-needed}]] \)

apply \( (\text{autoref-monadic (trace)}) \)
done

concrete-definition \( \text{find-path1-restr-code} \) uses \( \text{find-path1-restr-code} \)

export-code \( \text{find-path1-restr-code checking SML} \)

lemma \( \text{find-path1-restr-autoref-aux}: \)

assumes \( \text{G: (op-vis-insert}, \text{insert}) \in V \to (V) \text{vis-rel} \to (V) \text{vis-rel} \)

\( (\text{op-vis-memb}, (\in)) \in V \to (V) \text{vis-rel} \to \text{bool-rel} \)

assumes \( \text{Vid[simp]}: V = \text{Id} \)

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

\( \in (\text{Rm}, V) \text{g-impl-rel-ext} \to (V \to \text{bool-rel}) \to (V) \text{vis-rel} \to \\
(\langle\langle V \rangle\rangle \text{vis-rel}, \langle V \rangle \text{list-rel} \times_{r} V) \text{sum-rel} \text{nres-rel} \)

proof --
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:

- \textit{find-path0-spec} \ G \ P — find path to node that satisfies \( P \).
- \textit{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,{}))∈(R)Rs
  assumes I: (i,insert)∈R→(R)Rs→(R)Rs
  assumes EE: ∀x. x∈S ⇒ ∃xi. (xi,x)∈R
  shows ∃Si. (Si,S)∈(R)Rs
proof –
  define S' where S' = S

  have S⊆S' by (simp add: S'-def)
  from F this show (∃Si. (Si,S)∈(R)Rs)
  proof (induction)
    case empty thus ?case
    using E by (blast)
  next
    case (insert x S)
    then obtain xi Si where 1: (Si,S)∈(R)Rs and 2: (xi,x)∈R
    using EE unfolding S'-def by blast
    from I[THEN fun-relD, OF 2, THEN fun-relD, OF 1] show ?case ..
  qed
  qed
lemma obtain-finite-achs: finite $S \implies \exists x. (x, S) \in \Id \cdot \text{dflt-achs-rel}$
apply (erule gen-obtain-finite-set)
apply autoref
apply autoref
by blast

2.3.2 Framework Instantiation

definition unit-parametrization \equiv \text{dflt-parametrization} (\lambda \cdot (\text{RETURN} ()))

lemmas unit-parametrization-simp[simp, DFS-code-unfold] =
\text{dflt-parametrization-simp}[\text{mk-record-simp}, \text{OF}, \text{OF unit-parametrization-def}]

interpretation unit-dfs: \text{param-DFS-defs} where param=unit-parametrization for $G$.

locale unit-DFS = param-DFS $G$ unit-parametrization for $G :: (\nu, \text{more})$ graph-rec-scheme
begin
sublocale DFS $G$ unit-parametrization by unfold-locales simp-all
end

lemma unit-DFSI[Pure.intro?, intro?!]:
  assumes fb-graph $G$
  shows unit-DFS $G$
proof –
  interpret fb-graph $G$ by fact
  show ?thesis by unfold-locales
qed

definition find-reachable $G \equiv$ do {
  ASSERT (fb-graph $G$);
  $s \leftarrow \text{unit-dfs.it-dfs TYPE('a) }G; 
  \text{RETURN } (\text{dom } (\text{discovered } s))$
}

definition find-reachableT $G \equiv$ do {
  ASSERT (fb-graph $G$);
  $s \leftarrow \text{unit-dfs.it-dfsT TYPE('a) }G; 
  \text{RETURN } (\text{dom } (\text{discovered } s))$
}

2.3.3 Correctness

context unit-DFS begin
lemma find-reachable-correct: find-reachable $G \leq \text{SPEC} (\lambda r. \text{r = reachable})$
  unfolding find-reachable-def
apply (refine-vcg order-trans[OF it-dfs-correct])
apply unfold-locales
apply clarify
apply (drule (1) DFS-invar.nc-discovered-eq-reachable)
by auto

lemma find-reachableT-correct:
finite reachable ⇒ find-reachableT G ≤ SPEC (∃r. r = reachable)

unfolding find-reachableT-def
apply (refine-vcg order-trans[OF it-dfsT-correct])
apply unfold-locales
apply clarify
apply (drule (1) DFS-invar.nc-discovered-eq-reachable)
by auto

end

context unit-DFS begin

sublocale simple-impl G unit-parametrization unit-parametrization unit-rel
apply unfold-locales
apply (clarsimp simp: simple-state-rel-def) []
by auto

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

interpretation unit-simple-impl:

simple-impl-defs G unit-parametrization unit-parametrization
for G .

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

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

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

definition [DFS-code-unfold]: find-reachable-rec-impl G ≡ do {
  ASSERT (fb-graph G);
  s ← unit-simple-impl.rec-impl TYPE('a) G;
  RETURN (visited s)
}
lemma find-reachable-impl-refine:
  find-reachable-impl G ≤ ⇓ Id (find-reachable G)
unfolding find-reachable-impl-def find-reachable-def
apply (refine-vcg unit-DFS.impl-refine)
apply (simp-all add: unit-DFS.simple-state-rel-def)
done

lemma find-reachable-implT-refine:
  find-reachable-implT G ≤ ⇓ Id (find-reachableT G)
unfolding find-reachable-implT-def find-reachableT-def
apply (refine-vcg unit-DFS.impl-refine)
apply (simp-all add: unit-DFS.simple-state-rel-def)
done

lemma find-reachable-rec-impl-refine:
  find-reachable-rec-impl G ≤ ⇓ Id (find-reachable G)
unfolding find-reachable-rec-impl-def find-reachable-def
apply (refine-vcg unit-DFS.impl-refine)
apply (simp-all add: unit-DFS.simple-state-rel-def)
done

2.3.4 Synthesis of Executable Implementation

schematic-goal find-reachable-impl:
defines V ≡ Id :: ('v × 'v::hashable) set
assumes [unfolded V-def,autoref-rules]:
  (Gi, G) ∈ ⟨Rm, V⟩g-impl-rel-ext
notes [unfolded V-def,autoref-tyrel] =
  TYRELI[where R=(V)dflt-ahs-rel]
  TYRELI[where R=(V × r) (V)list-set-rel ras-rel]
shows nres-of (?c::?'c dres) ≤ ⇓ R (find-reachable-impl G)
unfolding if-cancel DFS-code-unfold ssnos-unfolds
using [[autoref-trace-failed-id, goals-limit=1]]
apply (autoref-monadic (trace))
done
concrete-definition find-reachable-code uses find-reachable-impl
export-code find-reachable-code checking SML

lemma find-reachable-code-correct:
assumes 1: fb-graph G
assumes 2: (Gi, G) ∈ ⟨Rm, Id⟩g-impl-rel-ext
assumes 4: find-reachable-code Gi = dRETURN r
shows (r, (g-E G)* 'g-V0 G)∈⟨Id⟩dflt-ahs-rel
proof –
from 1 interpret unit-DFS by rule
note find-reachable-code.refine[OF 2]
also note find-reachable-impl-refine
also note find-reachable-correct
finally show \(?\)thesis using 1 4 by (auto simp: RETURN-RES-refine-iff)
qed

schematic-goal find-reachable-implT:
fixes V :: (′vi×′a) set
assumes [autoref-ga-rules]: is-bounded-hashcode V eq bhc
assumes [autoref-rules]: (eq,=) ∈ V → V → bool-rel
assumes [autoref-ga-rules]: is-valid-def-hm-size TYPE (′vi) sz
assumes [autoref-rules]:
(Gi, G) ∈ ⟨Rm, V⟩g-impl-rel-ext
notes [autoref-tyrel] =
  TYRELI|where R=(V)ahs-rel bhc
  TYRELI|where R=(V ×ₗ (V)list-set-rel)ras-rel
shows RETURN (?c::′c) ≤⇓?R (find-reachable-implT G)
unfolding if-cancel DFS-code-unfold ssnos-unfolds
using [(autoref-trace-failed-id, goals-limit=1)]
apply (autoref-monadic (plain,trace))
done

concrete-definition find-reachable-codeT for eq bhc sz Gi

lemma find-reachable-codeT-correct:
fixes V :: (′vi×′a) 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 –
from G interpret graph by this
from FR interpret fb-graph using fb-graphI-fr by simp
interpret unit-DFS by unfold-locales

note find-reachable-codeT-refine[OF BHC EQ VDS 2]
also note find-reachable-implT-refine
also note find-reachableT-correct
finally show \(?\)thesis using FR by (auto simp: RETURN-RES-refine-iff)
qed

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

lemma all-unit-refine[simp]:
(()x)∈all-unit-rel unfolding all-unit-rel-def by simp
definition unit-list-rel :: ('c × 'a) set ⇒ (unit × 'a list) set
  where [to-relAPP]: unit-list-rel R ≡ UNIV

lemma unit-list-rel-refine[simp]: ((),y)∈(R)unit-list-rel
  unfolding unit-list-rel-def by auto

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

lemma [autoref-rules]:
  ((),[])∈(R)unit-list-rel
  (λ-. (),t)∈(R)unit-list-rel→(R)unit-list-rel
  (λ- -. (),(#))∈R → (R)unit-list-rel→(R)unit-list-rel
  by auto

schematic-goal find-reachable-rec-impl:
defines V ≡ Id :: ('v × 'v:::hashable) set
assumes [unfolded V-def,autoref-rules]:
  (Gi, G) ∈ (Rm, V)g-impl-rel-ext
notes [unfolded V-def,autoref-tyrel] =
  TYRELI[where R=⟨V⟩dflt-ahs-rel]
serves nres-of (?c::?'c dres) ≤⇓?R (find-reachable-rec-impl G)
  unfolding unit-simple-impl.ssns-unfolds
  DFS-code-unfold if-cancel if-False option.case
using [[autoref-trace-failed-id, goals-limit=1]]
apply (autoref-monadic (trace))
done
concrete-definition find-reachable-rec-code uses find-reachable-rec-impl
prepare-code-thms find-reachable-rec-code-def
export-code find-reachable-rec-code checking SML

lemma find-reachable-rec-code-correct:
  assumes 1: fb-graph G
  assumes 2: (Gi, G) ∈ (Rm, Id)g-impl-rel-ext
  assumes 4: find-reachable-rec-code Gi = dRETURN r
  shows (r, (g-E G)* " g-V0 G)∈⟨Id⟩dflt-ahs-rel
proof –
  from 1 interpret unit-DFS by rule
  note find-reachable-rec-code-refine[OF 2]
  also note find-reachable-rec-impl-refine
  also note find-reachable-correct
  finally show ?thesis using 1 4 by (auto simp: RETURN-RES-refine-iff)
qed

definition [simp]: op-reachable G ≡ (g-E G)* " g-V0 G
lemmas [autoref-op-pat] = op-reachable-def[symmetric]
context begin interpretation autoref-syn.

lemma autoref-op-reachable[autoref-rules]:
  fixes V :: ('vi ×'v) set
  assumes G: SIDE-PRECOND (graph G)
  assumes FR: SIDE-PRECOND (finite ((g-E G)'' g-V0 G))
  assumes BHC: SIDE-GEN-ALGO (is-bounded-hashcode V eq bhc)
  assumes EQ: GEN-OP eq (=) (V → V → bool-rel)
  assumes VDS: SIDE-GEN-ALGO (is-valid-def-hm-size TYPE ('vi) sz)
  shows (find-reachable-codeT eq bhc sz Gi, (OP op-reachable :: ⟨Rm, V⟩ g-impl-rel-ext)
    using assms 
  by (simp add: find-reachable-codeT-correct)
end

2.3.5 Conclusions

We have defined an efficient DFS-based implementation for op-reachable, and declared it to Autoref.
end

2.4 Find a Feedback Arc Set

theory Feedback-Arcs
imports 
  ../DFS-Framework
  CAVA-Automata.Digraph-Impl
  Reachable-Nodes
begin

A feedback arc set is a set of edges that breaks all reachable cycles. In this theory, we define an algorithm to find a feedback arc set.

definition is-fas :: ('v, 'more) graph-rec-scheme ⇒ 'v rel ⇒ bool where
  is-fas G EC ≡ ¬(∃ u ∈ (g-E G)'' g-V0 G. (u, u) ∈ (g-E G − EC)+)

lemma is-fas-alt:
  is-fas G EC = acyclic ((g-E G ∩ ((g-E G)'' g-V0 G × UNIV) − EC))
unfolding is-fas-def acyclic-def
proof (clarsimp, safe)
  fix u
  assume A: (u, u) ∈ (g-E G ∩ (g-E G)'' g-V0 G × UNIV − EC)⁺
  hence (u, u) ∈ (g-E G − EC)⁺ by (rule trancl-mono) blast
  moreover from A have u ∈ (g-E G)'' g-V0 G by (cases rule: converse-tranclE)
  auto
  moreover assume ∄ u ∈ (g-E G)'' g-V0 G. (u, u) ∉ (g-E G − EC)⁺
end
ultimately show \( \text{False} \) by blast

next

fix \( u \ v 0 \)

assume 1: \( v 0 \in g-V 0 \ G \) and 2: \((v 0 ,u)\in(g-E \ G)^*\) and 3: \((u,u)\in(g-E \ G - EC)^+\)

have \((u ,u) \in (\text{Restr} (g-E \ G - EC) (g-E \ G)^* g-V 0 \ G)^+)\)

apply (rule trancl-restrict-reachable[\( \text{OF 3} \), where \( S = (g-E \ G)^* g-V 0 \ G \)])

apply (rule order-trans[\( \text{OF - rtrancl-image-unfold-right} \)])

using 1 2 by auto

hence \((u ,u) \in (g-E \ G \cap (g-E \ G)^* g-V 0 \ G \times UNIV - EC)^+)\)

by (rule trancl-mono) auto

moreover assume \( \forall x . \ (x , x) \notin (g-E \ G \cap (g-E \ G)^* g-V 0 \ G \times UNIV - EC)^+)\)

ultimately show \( \text{False} \) by blast

qed

2.4.1 Instantiation of the DFS-Framework

record \( 'v \ \text{fas-state} = 'v \ \text{state} + \)

\( f a s :: ( 'v \times 'v) \ \text{set} \)

lemma \( \text{fas-more-cong} \): \( \text{state}.more \ s = \text{state}.more \ s' \Rightarrow fas \ s = fas \ s' \)

by (cases \( s \), cases \( s' \), simp)

lemma [simp]: \( s() \ \text{state}.more := () \ fas = foo () \ | = s () \ fas := foo () \)

by (cases \( s \)) simp

definition \( \text{fas-params} :: ( 'v . ('v,unit) \ \text{fas-state-ext}) \ \text{parameterization} \)

where \( \text{fas-params} :: \text{dflt-parametrisation state}.more \)

\( (\text{RETURN} () \ fas = () |) |\)

\( \text{on-back-edge} := \lambda u v s . \ \text{RETURN} () \ fas = \text{insert} (u,v) (fas \ s) |) |\)

lemmas \( \text{fas-params-simp}[\text{simp}] = \)

\( \text{gen-parameterization.simps[mk-record-simp, OF fas-params-def[simplified]]} \)

interpretation \( fas : \ \text{param-DFS-defs} \ where \ param=fas-params \ \text{for} \ G \).

Find feedback arc set

definition \( \text{find-fas} \ G \equiv \text{do} \{
\text{ASSERT} (\text{graph} \ G); \)
\text{ASSERT} (\text{finite} (g-E \ G)^* g-V 0 \ G));
\( s \leftarrow f a s . \text{it-dfsT} \ \text{TYPE} ('a) \ G; \)
\text{RETURN} (fas.state.fas \ s) \}

locale \( fas = \)

\( \text{param-DFS} \ G \ \text{fas-params} \)

for \( G :: ( 'v , 'more) \ \text{graph-rec-scheme} \)

+ \n
assumes \( \text{finite-reaching}[\text{simp, intro}]: \text{finite} ((g-E \ G)^* g-V 0 \ G) \)

155
begin

sublocale DFS G fas-pars
  apply unfold-locales
  apply (simp-all add: fas-params-def)
  done

end

lemma fasI:
  assumes graph G
  assumes finite ((g-E G)∗ " g-V0 G)
  shows fas G
proof –
  interpret graph G by fact
  interpret fb-graph G by (rule fb-graphI-fr[OF assms(2)])
  show ?thesis by unfold-locales fact
qed

2.4.2 Correctness Proof

locale fas-invar = DFS-invar where param = fas-pars + fas
begin

lemma (in fas) i-fas-eq-back: is-invar (λs. fas-state.fas s = back-edges s)
  apply (induct rule: establish-invarI)
  apply (simp-all add: cond-def cong: fas-more-cong)
  apply (simp add: empty-state-def)
  done
lemmas fas-eq-back = i-fas-eq-back[THEN make-invar-thm]

lemma find-fas-correct-aux:
  assumes NC: ¬cond s
  shows is-fas G (fas-state.fas s)
proof –
  note [simp] = fas-eq-back

  from nc-edges-covered[OF NC] edges-disjoint have
  E ∩ reachable × UNIV = back-edges s = tree-edges s ∪ cross-edges s
  by auto

  with tree-cross-acyclic show is-fas G (fas-state.fas s)
    unfolding is-fas-alt by simp
qed

end

lemma find-fas-correct:
  assumes graph G
  assumes finite ((g-E G)∗ " g-V0 G)
shows \text{find-fas} G \leq \text{SPEC} (\text{is-fas} G)

unfolding \text{find-fas-def}

proof (\text{refine-req le-ASSERTI order-trans}[OF DFS.it-dfsT-correct], clarsimp-all)

\begin{itemize}
\item \text{interpret} graph G by fact
\item \text{assume} finite \((g\cdot E G) \times g\cdot V0 G\)
\item then \text{interpret} \text{fb-graph} G by (rule \text{fb-graph-fr})
\item \text{interpret} \text{fas} by unfold-locales fact
\item \text{show} DFS G \text{fas-params} by unfold-locales
\end{itemize}

next

\begin{itemize}
\item fix s
\item \text{assume} DFS-invar G \text{fas-params} s
\item \text{then interpret} DFS-invar G \text{fas-params} s.
\item \text{interpret} \text{fas-invar} G s by unfold-locales fact
\item \text{assume} \neg \text{fas}.
\item \text{cond TYPE}'b G s
\item thus \text{is-fas} G (\text{fas-state.fas} s)
\item by (rule find-fas-correct-aux)
\item qed (rule assms)+
\end{itemize}

2.4.3 Implementation

record \text{'v fas-state-impl} = \text{'v simple-state} +
\text{fas} :: (\text{'v} \times \text{'v}) set

definition \text{fas-erel} \equiv \{
\text{fas-state-impl.fas} = f \mid \exists \text{fas-state.fas} = f \}_f \mid f \cdot \text{True} \}

abbreviation \text{fas-rel} \equiv (\text{fas-erel})\text{simple-state-rel}

definition \text{fas-params-impl}
:: (\text{'v,'v fas-state-impl,(\text{'v,unit} fas-state-impl-ext)} gen-parameterization

where \text{fas-params-impl}
\equiv dflt-parametrization simple-state.more (RETURN \{ \text{fas} = \{ \}_f \} \mid \text{on-back-edge} \equiv \lambda u v s.\text{RETURN}\{ \text{fas} = \text{insert} (u,v) (\text{fas} s) \}_f\})

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

lemma \text{fas-impl}: (s_i,s) \in \text{fas-rel}
    \Rightarrow \text{fas-state-impl.fas} s_i = \text{fas-state.fas} s
    \by (cases s_i, cases s, simp add: simple-state-rel-def fas-erel-def)

interpretation \text{fas-impl}:: simple-impl-defs G \text{fas-params-impl} \text{fas-params}
for G.

term \text{fas-impl.tailrec-impl} term \text{fas-impl.tailrec-implT} term \text{fas-impl.rec-impl}

definition [DFS-code-unfold]: \text{find-fas-impl} G \equiv do {\text{ASSERT} (\text{graph} G);}
\[
\text{ASSERT } \text{finite } ((g \cdot E G)^* \text{ "g-V0 G});
\]
\[
s \leftarrow \text{fas-impl.tailrec-implT TYPE('a) G; RETURN } \text{fas } s)
\]

context fas begin

sublocale simple-impl G fas-params fas-params-impl fas-erel
apply unfold-locales
apply (intro fun-relI, clarsimp simp: simple-state-rel-def, parametricity) []
apply (auto simp: fas-erel-def fas-impl simple-state-rel-def)
done

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

lemma find-fas-impl-refine: find-fas-impl G ≤⇓ Id (find-fas G)
unfolding find-fas-impl-def find-fas-def
apply (refine-vcg fas.impl-refine)
apply (simp-all add: fas-impl fasI)
done

2.4.4 Synthesis of Executable Code

record ('si,'nsi,'fsi)fas-state-impl' = ('si,'nsi)simple-state-impl +
fas-impl :: 'fsi
definition [to-relAPP]: fas-state-erel frel erel ≡ {
  \((\text{fas-impl} = fi, \ldots = mi), (\text{fas} = f, \ldots = m)) \mid fi f m.
  (f,i) \in frel \land (m,i) \in erel\}
consts
  i-fas-state-ext :: interface ⇒ interface ⇒ interface
lemmas [autoref-rel-intf] = REL-INTFI[of fas-state-erel i-fas-state-ext]

term fas-update
term fas-state-impl'.fas-impl-update
lemma [autoref-rules]:
  fixes ns-rel vis-rel frel erel
defines R ≡ (ns-rel,vis-rel,(frel,erel)fas-state-erel)ss-impl-rel
shows
  (fas-state-impl'.ext, fas-state-impl-ext) ∈ frel → erel → (frel,erel)fas-state-erel
  (fas-impl, fas-state-impl,fas) ∈ R → frel
  (fas-state-impl'.fas-impl-update, fas-update) ∈ (frel → frel) → R → R
unfolding fas-state-erel-def ss-impl-rel-def R-def
schematic-goal find-fas-impl:
  fixes V :: ('vi×'v) set
  assumes [autoref-ga-rules]: is-bounded-hashcode V eq bhc
  assumes [autoref-rules]: (eq, (=)) ∈ V → V → bool-rel
  assumes [autoref-ga-rules]: is-valid-def-hm-size TYPE ('vi) sz
  assumes [autoref-rules]:
    (Gi, G) ∈ ⟨Rm, V⟩ g-impl-rel-ext
  shows RETURN (find-fas-code eq bhc sz Gi)
proof −
  note find-fas-code-refine[OF assms]
  also note find-fas-impl-refine
  finally show ?thesis .
qed

context begin interpretation autoref-syn .

Declar this algorithm to Autoref:

theorem find-fas-code-autoref[autoref-rules]:
  fixes V :: ('vi×'v) set and bhc
  defines RR ≡ ⟨⟨V×r,V⟩ahs-rel (prod-bhc bhc bhc)⟩nres-rel
  assumes BHC: SIDE-GEN-ALGO (is-bounded-hashcode V eq bhc)
  assumes EQ: GEN-OP eq (=) (V → V → bool-rel)
  assumes VDS: SIDE-GEN-ALGO (is-valid-def-hm-size TYPE ('vi) sz)
  assumes 2: (Gi, G) ∈ ⟨Rm, V⟩ g-impl-rel-ext
  shows (RETURN (find-fas-code eq bhc sz Gi),
(OP find-fas
::: (Rm, V)g-impl-rel-ext \rightarrow RR)$\in RR

unfolding RR-def
apply (rule nres-relI)
using assms
by (simp add: find-fas-code-refine)

end

2.4.5 Feedback Arc Set with Initialization

This algorithm extends a given set to a feedback arc set. It works in two steps:

1. Determine set of reachable nodes

2. Construct feedback arc set for graph without initial set

definition find-fas-init where
find-fas-init G FI \equiv do 
  ASSERT (graph G);
  ASSERT (finite \((g-E G)^\ast g-V0 G\));
  let nodes = \((g-E G)^\ast g-V0 G\);
  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 \((F1 \cup F2) \leftrightarrow \)
\((is-fas \((g-V = g-V G, g-E = g-E G - F2, g-V0 = (g-E G)^\ast g-V0 G \}) \cup F1)\)

unfolding is-fas-def
apply (auto simp: set-diff-diff-left Un-commute)
by (metis ImageI rtrancl-trans subsetCE rtrancl-mono[OF g-E G - F2 g-E G, OF Diff-subset])

lemma graphI-init:
assumes graph G
shows graph \((g-V = g-V G, g-E = g-E G - FI, g-V0 = (g-E G)^\ast g-V0 G \})

proof -
interpret graph G by fact
show \(?thesis
apply unfold-locales
using reachable-V apply simp
using E-ss apply force
done
qed

160
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 (λfas. is-fas G fas ∧ FI ⊆ fas)

unfolding find-fas-init-def
apply (refine-vcg order-trans[OF find-fas-correct])
apply clarsimp-all
apply (rule graphI-init)
apply simp
apply (rule finite-subset[rotated], rule assms)
apply (metis Diff-subset Image-closed-trancl reachable-mono
      rtrancl-image-unfold-right rtrancl-reflcl rtrancl-trancl-reflcl
      trancl-rtrancl-absorb)
apply (simp add: is-fas-join[where ?F2.0=FI] Un-commute)
done

lemma gen-cast-set[autoref-rules-raw]:
assumes PRIO-TAG-GEN-ALGO
assumes INS: GEN-OP ins Set.insert (Rk→(Rk)Rs2→(Rk)Rs2)
assumes EM: GEN-OP emp {} ((Rk)Rs2)
assumes IT: SIDE-GEN-ALGO (is-set-to-list Rk Rs1 tsl)
shows ((λs. gen-union (λx. foldli (tsl x)) ins s emp,CAST) ∈ ((Rk)Rs1) → ((Rk)Rs2)

proof –
  note [autoref-rules] = GEN-OP-D[OF INS]
  note [autoref-rules] = GEN-OP-D[OF EM]
  note [autoref-ga-rules] = SIDE-GEN-ALGO-D[OF IT]
  have 1: CAST = (λs. s ∪ {}) by auto
  show ?thesis
    unfolding 1
          by autoref
qed

lemma gen-cast-fun-set-rel[autoref-rules-raw]:
assumes INS: GEN-OP mem (∈) (Rk→(Rk)Rs→bool-rel)
shows (λs x. mem x s,CAST) ∈ ((Rk)Rs) → ((Rk)fun-set-rel)
proof –
  have A: \A. (λx. x∈s,CAST s) ∈ br Collect (λx. True)
        by (auto simp: br-def)
  show ?thesis
    unfolding fun-set-rel-def
    apply rule
    apply rule
    defer
    apply (rule A)
    using INS[simplified]

161
apply parametricity
done
qed

lemma find-fas-init-impl-aux-unfolds:
  Let \((E^{\ast}''V0) = \text{Let} (\text{CAST} (E^{\ast}''V0))\)
  \((\lambda S. \text{RETURN} (FI \cup S)) = (\lambda S. \text{RETURN} (FI \cup \text{CAST} S))\)
by simp-all

schematic-goal find-fas-init-impl:
  fixes \(V :: (\prime vi \times \prime v)\) set and bhc
  assumes [autoref-ga-rules]: is-bounded-hashcode V eq bhc
  assumes [autoref-rules]: \((\text{eq},(=)) \in V \rightarrow V \rightarrow \text{bool-rel}\)
  assumes [autoref-ga-rules]: is-valid-def-hm-size TYPE ('vi) sz
  assumes [autoref-rules]:
    \((Gi, G) \in (\langle Rm, V \rangle) \text{g-impl-rel-ext} \quad (FIi,FI)\in(\langle V \times V \rangle) \text{fun-set-rel}\)
  shows \(\text{RETURN} (\langle c::c \rangle) \leq \downarrow R (\text{find-fas-init G FI})\)
unfolding find-fas-init-def
unfolding find-fas-init-impl-aux-unfolds
by (autoref-monadic (plain,trace))

concrete-definition find-fas-init-code for eq bhc sz Gi FIi
  uses find-fas-init-impl
thm find-fas-code
export-code find-fas-init-code checking SML

context begin interpretation autoref-syn .

The following theorem declares our implementation to Autoref:

theorem find-fas-init-code-autoref [autoref-rules]:
  fixes \(V :: (\prime vi \times \prime v)\) set and bhc
defines \(RR \equiv \langle V \times V \rangle\) fun-set-rel
  assumes SIDE-GEN-ALGO (is-bounded-hashcode V eq bhc)
  assumes GEN-OP eq (=) \((V \rightarrow V \rightarrow \text{bool-rel})\)
  assumes SIDE-GEN-ALGO (is-valid-def-hm-size TYPE ('vi) sz)
  shows \((\lambda Gi FI. \text{RETURN} (\text{find-fas-init-code eq bhc sz Gi FIi})\text{,find-fas-init}) \in (\langle Rm, V \rangle) \text{g-impl-rel-ext} \rightarrow RR \rightarrow (RR)\text{nres-rel}\)
unfolding RR-def
apply (intro fun-relI nres-relI)
using assms
by (simp add: find-fas-init-code.refine)
end
2.4.6 Conclusion

We have defined an algorithm to find a feedback arc set, and one to extend a given set to a feedback arc set. We have registered them to Autoref as implementations for \textit{find-fas} and \textit{find-fas-init}.

For preliminary refinement steps, you need the theorems \textit{find-fas-correct} and \textit{find-fas-init-correct}.

\begin{verbatim}
thm find-fas-code-autoref find-fas-init-code-autoref
thm find-fas-correct thm find-fas-init-correct
\end{verbatim}

2.5 Nested DFS

theory \textit{Nested-DFS}
imports \textit{DFS-Find-Path}
begin

Nested DFS is a standard method for Buchi-Automaton emptiness check.

2.5.1 Auxiliary Lemmas

\begin{verbatim}
lemma \textit{closed-restrict-aux}: 
assembles CL: E''F \subseteq F \cup S
assembles NR: E''U \cap S = \{\}
assembles SS: U \subseteq F
shows E''U \subseteq F
— Auxiliary lemma to show that nodes reachable from a finished node must be finished if, additionally, no stack node is reachable
proof clarify
  fix u v
  assume A: (u,v) \in E^* u \in U
  hence M: E''\{u\} \cap S = \{\} u \in F using NR SS by blast+

  from A(1) M show v \in F
  apply (induct rule: converse-rtrancl-induct)
  using CL apply (auto dest: rtrancl-Image-advance-ss)
done
\end{verbatim}

qed

2.5.2 Instantiation of the Framework

\begin{verbatim}
record \textit{'v blue-dfs-state = 'v state +
  lasso ::= ('v list \times 'v list) option
red ::= 'v set
\end{verbatim}
type-synonym 'v blue-dfs-param = ('v, ('v, unit) blue-dfs-state-ext) parameterization

lemma lasso-more-cong[cong]: state.more s = state.more s' \implies \lasso s = \lasso s'
by (cases s, cases s') simp

lemma red-more-cong[cong]: state.more s = state.more s' \implies \red s = \red s'
by (cases s, cases s') simp

lemma [simp]: s[] state.more := ( | \lasso = foo, \red = bar | ) = s ( | \lasso := foo, \red := bar | )
by (cases s) simp

abbreviation dropWhileNot v ≡ dropWhile ((̸=) v)
abbreviation takeWhileNot v ≡ takeWhile ((̸=) v)

locale BlueDFS-defs = graph-defs G
for G :: ('v, 'more) graph-rec-scheme +
fixes acpt :: 'v ⇒ bool
begin

definition blue s ≡ dom (finished s) − \red s
definition cyan s ≡ set (stack s)
definition white s ≡ − dom (discovered s)

abbreviation red-dfs R ss x ≡ find-path1-restr-spec (G (| g-V0 := {x} |)) ss R

definition mk-blue-witness :: 'v blue-dfs-state ⇒ 'v fpr-result ⇒ ('v,unit) blue-dfs-state-ext
where
mk-blue-witness s redS ≡ case redS of
  Inl R' ⇒ ( | \lasso = None, \red = (R' [ fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fiss fis

definition run-red-dfs
:: 'v ⇒ 'v blue-dfs-state ⇒ ('v,unit) blue-dfs-state-ext nres
where
run-red-dfs u s ≡ case lasso s of None ⇒ do {
  redS ← red-dfs (red s) (λx. x = u ∨ x ∈ cyan s) u;
  RETURN (mk-blue-witness s redS)
}
| - ⇒ NOOP s

Schwoon-Esparza extension

definition se-back-edge u v s ≡ case lasso s of
  None ⇒
  — it’s a back edge, so u and v are both on stack
  — we differentiate whether u or v is the ‘culprit’
  — to generate a better counter example

164
if accept u then
  let rs = rev (tl (stack s));
  ur = rs;
  ul = u # dropWhileNot v rs
  in RETURN (lasso = Some (ur, ul), red = red s)
else if accept v then
  let rs = rev (stack s);
  vr = takeWhileNot v rs;
  vl = dropWhileNot v rs
  in RETURN (lasso = Some (vr, vl), red = red s)
else NOOP s
| - => NOOP s

definition blue-dfs-params :: 'v blue-dfs-param
  where blue-dfs-params = ()
  on-init = RETURN (lasso = None, red = {}),
  on-new-root = \v0 s. NOOP s,
  on-discover = \u v s. NOOP s,
  on-finish = \u s. if accept u then run-red-dfs u s else NOOP s,
  on-back-edge = se-back-edge,
  on-cross-edge = \u v s. NOOP s,
  is-break = \s. lasso s \neq None |

schematic-goal blue-dfs-params-simps[simp]:
  on-init blue-dfs-params = ?OI
  on-new-root blue-dfs-params = ?ONR
  on-discover blue-dfs-params = ?OD
  on-finish blue-dfs-params = ?OF
  on-back-edge blue-dfs-params = ?OBE
  on-cross-edge blue-dfs-params = ?OCE
  is-break blue-dfs-params = ?IB
unfolding blue-dfs-params-def gen-parameterization.simps
by (rule refl)+

sublocale param-DFS-defs G blue-dfs-params
  by unfold-locales
end

locale BlueDFS = BlueDFS-defs G accept + param-DFS G blue-dfs-params
  for G :: ('v, 'more) graph-rec-scheme and accept :: 'v => bool

lemma BlueDFSI:
  assumes fb-graph G
  shows BlueDFS G
proof –
  interpret fb-graph G by fact
  show ?thesis by unfold-locales
qed

locale BlueDFS-invar = BlueDFS +
DFS-invar where param = blue-dfs-params

context BlueDFS-defs begin

lemma BlueDFS-invar-eq[simp]:
shows DFS-invar G blue-dfs-params s ←→ BlueDFS-invar G accept s
proof
assumes DFS-invar G blue-dfs-params s
interpret DFS-invar G blue-dfs-params s by fact
show BlueDFS-invar G accept s by unfold-locales
next
assumes BlueDFS-invar G accept s
then interpret BlueDFS-invar G accept s.
show DFS-invar G blue-dfs-params s by unfold-locales
qed

end

2.5.3 Correctness Proof

context BlueDFS begin

definition blue-basic-invar s ≡
case lasso s of
  None ⇒ restr-invar E (red s) (λx. x∈set (stack s))
  ∧ red s ⊆ dom (finished s)
  | Some l ⇒ True

lemma (in BlueDFS-invar) red-DFS-precond-aux:
assumes BI: blue-basic-invar s
assumes [simp]: lasso s = None
assumes SNE: stack s ≠ []
shows
  fb-graph (G (g-V0 := {hd (stack s)} []))
and fb-graph (G (g-E := E ∩ UNIV × − red s, g-V0 := {hd (stack s)} []))
and restr-invar E (red s) (λx. x ∈ set (stack s))
using stack-reachable (stack s ≠ []);
apply (rule-tac fb-graph-subset, auto) []
using stack-reachable (stack s ≠ []);
apply (rule-tac fb-graph-subset, auto) []
using BI apply (simp add: blue-basic-invar-def)
done

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

\text{proof} –

have [simp]: \( (\lambda x. \ x = \text{hd} (\ stack \ s) \lor x \in \text{cyan} (\text{finish} (\ \text{hd} (\ stack \ s)) \ s)) = (\lambda x. x \in \text{set} (\ stack \ s)) \)
using \( \text{stack} \ s \neq [] \)
unfolding \( \text{finish-def cyan-def} \) by (auto simp: neq-Nil-conv)

show \( ?\text{thesis} \)
unfolding run-red-dfs-def
apply simp
apply (refine-vcg)
apply simp

proof –
fix \( \text{fp1} \)
define \( s' \) where \( s' = \text{finish} (\ \text{hd} (\ stack \ s)) \ s \)
assume FP-spec:
\( \text{find-path1-restr-pred} (G (g-V0 := \{\text{hd} (\ stack \ s)\}) (\\lambda x. x \in \text{set} (\ stack \ s)) \ (\text{red} s) \ \text{fp1}) \)
assume BlueDFS-invar \( G \ ) \text{ accept} (s'(\state\ more := \text{mk-blue-witness} \ s' \ \text{fp1})) \)
then interpret \( i: \text{BlueDFS-invar} \ G \ ) \text{ accept} (s'(\state\ more := \text{mk-blue-witness} \ s' \ \text{fp1})) \)
by simp

have [simp]:
\( \text{red} \ s' = \text{red} \ s \)
\( \text{discovered} \ s' = \text{discovered} \ s \)
dom (\text{finished} \ s') = \text{insert} (\text{hd} (\ stack \ s)) \ (\text{dom} (\text{finished} \ s)) \)
unfolding s'-def finish-def by auto

\{ 
fix \( R' \)
assume [simp]: \( \text{fp1} = \text{Inl} R' \)
from FP-spec[unfolded \( \text{find-path1-restr-pred-def} \), simplified]
have \( R'FMT: \ R' = \text{red} \ s \cup \text{E}^+ \ \{ \text{hd} (\ stack \ s) \} \)
and \( RI: \text{restr-invar} \ E \ R' (\\lambda x. x \in \text{set} (\ stack \ s)) \)
by auto

from BI have \( \text{red} \ s \subseteq \text{dom} (\text{finished} \ s) \)
unfolding blue-basic-invar-def by auto
also have \( \text{E}^+ \ \{ \text{hd} (\ stack \ s) \} \subseteq \text{dom} (\text{finished} \ s) \)
proof (intro subsetI, elim ImageE, simp)
fix \( v \)
assume \((\text{hd} \ (\text{stack} \ s),v) \in E^+\)

then obtain \(u\) where \((\text{hd} \ (\text{stack} \ s),u) \in E\) and \((u,v) \in E^*\)
by (auto simp: trancl-unfold-left)

from RI have NR: \(E^+ \ (\{\text{hd} \ (\text{stack} \ s)\} \cap \text{set} \ (\text{stack} \ s) = \{\}\)
unfolding restr-invar-def by (auto simp: R'FMT)

with \((\text{hd} \ (\text{stack} \ s),u) \in E\) have u\(\notin\)set (\text{stack} \ s) by auto
with i.finished-closed[simplified] \((\text{hd} \ (\text{stack} \ s),u) \in E\)
have UID: \(u \in \text{dom} \ (\text{finished} \ s)\) by (auto simp: stack-set-def)

from NR \((\text{hd} \ (\text{stack} \ s),u) \in E\) have NR': \(E^+ \ (\{u\} \cap \text{set} \ (\text{stack} \ s) = \{\}\)
by (auto simp: trancl-unfold-left)

have CL: \(E^\\{u\} \subseteq \text{dom} \ (\text{finished} \ s) \cup \text{set} \ (\text{stack} \ s)\)
using finished-closed discovered-eq-finished-un-stack
by simp

from closed-restrict-aux[OF CL NR'] UID
have \(E^* \ (\{u\} \cap \text{dom} \ (\text{finished} \ s) = \{\}\)
with \((u,v) \in E^*\) show \(v \in \text{dom} \ (\text{finished} \ s)\) by auto
qed

finally (sup-least)
have \(R' \subseteq \text{dom} \ (\text{finished} \ s) \land \text{red} \ s \subseteq \text{dom} \ (\text{finished} \ s)\)
by (simp add: R'FMT)

note aux1 = this

show blue-basic-invar \((\forall (s'|(\text{state}.\ more := \text{mk-blue-witness} \ s' \ fp1))\)
unfolding blue-basic-invar-def mk-blue-witness-def
apply (simp split: option.splits sum.splits)
apply (intro allI conjI impI)
using FP-spec SNE
apply (auto)
simp: s'-def blue-basic-invar-def find-path1-restr-pred-def
simp: restr-invar-def
simp: neq-Nil-conv]

apply (auto dest!: aux1] []
done
qed

lemma blue-basic-invar: is-invar blue-basic-invar
proof (induct rule: establish-invarI)
case (finish s) then interpret BlueDFS-invar where \(s=s\) by simp

have \([\text{simp}]: (\lambda x. \ x = \text{hd} \ (\text{stack} \ s) \lor x \in \text{cyan} \ (\text{finish} \ (\text{hd} \ (\text{stack} \ s)) \ s)) =\)
\( (\lambda x. x \in \text{set} (\text{stack } s)) \)
using \( \text{stack } s \neq [] \)
unfolding \text{finish-def cyan-def by} \( \text{(auto simp: neq-Nil-conv)} \)

from \text{finish show} \ ?case
apply (simp)
apply (intr conjI impI)
apply (rule leaf-trans[OF \text{dfs-pres-bbi}], assumption+, simp)

apply (auto simp: \text{restr-invar-def blue-basic-invar-def neq-Nil-conv}) []
done

qed (auto simp: \text{blue-basic-invar-def cond-def se-back-edge-def}
  simp: \text{restr-invar-def empty-state-def pred-defs}
  simp: \text{DFS-invar.discovered-eq-finished-un-stack}
  simp del: \text{BlueDFS-invar-eq}
  split: option.splits)

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

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

sublocale \text{DFS G blue-dfs-params}
  apply unfold-locales

apply (clarsimp-all
  simp: \text{se-back-edge-def run-red-dfs-def refine-pw-simps pr-on-defs}
  split: option.splits)

unfolding \text{nofail-SPEC-iff}
apply (refine-vcg)
apply (erule \text{BlueDFS-invar.red-DFS-precond}, auto) []

apply (simp add: \text{cyan-def finish-def})
apply (erule \text{BlueDFS-invar.red-DFS-precond}, auto) []

apply (rule TrueI)
done

end

context \text{BlueDFS-invar}
begin

context assumes [simp]: \( \text{lasso } s = \text{None} \)
begin
lemma \text{red-closed}:
  \( E \models \text{red } s \subseteq \text{red } s \)
using s-blue-basic-invar
unfolding blue-basic-invar-def restr-invar-def
by simp

lemma red-stack-disjoint:
set (stack s) ∩ red s = {}
using s-blue-basic-invar
unfolding blue-basic-invar-def restr-invar-def
by auto

lemma red-finished: red s ⊆ dom (finished s)
using s-blue-basic-invar
unfolding blue-basic-invar-def
by auto

lemma all-nodes-colored: white s ∪ blue s ∪ cyan s ∪ red s = UNIV
unfolding white-def blue-def cyan-def
by (auto simp: stack-set-def)

lemma colors-disjoint:
white s ∩ (blue s ∪ cyan s ∪ red s) = {}
blue s ∩ (white s ∪ cyan s ∪ red s) = {}
cyan s ∩ (white s ∪ blue s ∪ red s) = {}
red s ∩ (white s ∪ blue s ∪ cyan s) = {}
unfolding white-def blue-def cyan-def
using finished-discovered red-finished
unfolding stack-set-def
by blast

end

lemma (in BlueDFS) i-no-accept-cyle-in-finish:
is-invar (λs. lasso s = None → (∀ x. accept x ∧ x ∈ dom (finished s) → (x,x) /∈ E+))
proof (induct rule: establish-invarI)
  case (finish s s' u) then interpret BlueDFS-invar where s=s by simp
  let ?onstack = λx. x∈set (stack s)
  let ?rE = rel-restrict E (red s)
  from finish obtain sh st where [simp]: stack s = sh#st
  by (auto simp: neg-Nil-conv)
  have 1: g-E (G [] g-V0 := {hd (stack s)} []) = E by simp
  { fix R::'v set
  let ?R' = R' /\ff/

170
let \( \gamma = \gamma' \) (lasso := None, red := \gamma \))

assume \( \forall v. (\text{hd} (\text{stack} s), v) \in \gamma E^+ \implies \neg \text{onstack} v \)
and accept; accept u
and NL[simp]: lasso s = None
hence no-hd-cycle: (\text{hd} (\text{stack} s), \text{hd} (\text{stack} s)) \notin \gamma E^+
  by auto
from finish have stack s \neq [] by simp
from hd-in-set[OF this] have \( \text{hd} (\text{stack} s) \notin \text{red} s \)
  using red-stack-disjoint
  by auto
hence \( (\text{hd} (\text{stack} s), \text{hd} (\text{stack} s)) \notin E^+ \)
  using no-hd-cycle rel-restrict-tranclI red-closed[OF NL]
  by metis
with accept finish have
  \( \forall x. \text{accept} x \land x \in \text{dom} (\text{finished} \gamma s) \implies (x,x) \notin E^+ \)
  by auto
} with finish have
red-dfs (\text{red} s) ?onstack (\text{hd} (\text{stack} s))
\leq \text{SPEC} (\lambda x. \forall R. x = \text{Inl} R \implies 
  \text{DFS-invar} G \text{ blue-dfs-params} (\text{lasso-update \ Map.empty} s'(\mid \text{red} := R \mid))
\text{[[ this]]} (\forall x. \text{accept} x \land x \in \text{dom} (\text{finished} \gamma s') \implies (x,x) \notin E^+))
apply --
apply (rule find-path1-restr-spec-rule, intro conjI)
apply (rule red-DFS-precond, simp-all) []
unfolding 1
apply (rule red-DFS-precond, simp-all) []
apply (auto simp: find-path1-restr-pred-def restr-invar-def)
done
note aux = leof-trans[OF this[simplified, THEN leof-lift]]

note [refine-vcg del] = find-path1-restr-spec-rule

from finish show ?case
apply simp
apply (intro conjI impl)
unfolding run-red-dfs-def mk-blue-witness-def cyan-def
apply clarsimp
apply (refine-vcg aux)
apply (auto split: sum.splits)
done
next
  case back-edge thus ?case
  by (simp add: se-back-edge-def split: option.split)
qed simp-all

171
lemma no-accept-cycle-in-finish:
\[ \text{lasso } s = \text{None; accept } v; v \in \text{ dom } (\text{finished } s) \implies (v, v) \notin E^+ \]  
using i-no-accept-cycle-in-finish \[ \text{THEN make-invar-thm} \]  by blast

end

context BlueDFS

begin

definition lasso-inv where
\[ \text{lasso-inv } s \equiv \forall \text{ pr pl. lasso } s = \text{Some } (\text{pr}, \text{pl}) \implies \]
\[ \text{pl} \neq [] \]
\[ \land (\exists v \in V_0. \text{ path } E v \varnothing \text{ pr (hd pl)}) \]
\[ \land \text{ accept (hd pl)} \]
\[ \land \text{ path } E (\text{hd pl}) \text{ pl (hd pl)} \]

lemma (in BlueDFS-invar) se-back-edge-lasso-inv:

assumes b-inv: lasso-inv s  
and ne: stack s \neq []  
and R: lasso s = None  
and p: (hd (stack s), v) \in \text{pending } s  
and v: v \in \text{ dom } (\text{discovered } s) v \notin \text{ dom } (\text{finished } s)  
and s': s' = \text{back-edge (hd (stack s))} v (s\{\text{pending := pending } s - \{(u, v)\}\})  
shows se-back-edge (hd (stack s)) v s'  
\leq \text{SPEC (\lambda e. DFS-invar G blue-dfs-params (s'|state.more := e)) } \rightarrow \]
\[ \text{lasso-inv } (s'|\text{state.more := e})) \]

proof –  

from v stack-set-def have v-in: v \in \text{set } (\text{stack } s) by simp  
from p have uv-edg: (hd (stack s), v) \in E by (auto dest: pendingD)  

{  
assume accept: accept (hd (stack s))  
let ?ur = rev (tl (stack s))  
let ?ul = hd (stack s) \# \text{dropWhileNot } v (rev (tl (stack s)))  
let ?s = s'[lasso := \text{Some } (?ur, ?ul), \text{red := red } s]  

assume DFS-invar G blue-dfs-params ?s  

have [simp]: stack ?s = stack s  
by (simp add: s')  

have hd-ul[simp]: hd ?ul = hd (stack s) by simp  

have ?ul \neq [] by simp  

moreover have P: \exists v \in V_0. \text{ path } E v \varnothing ?ur (hd ?ul)  
using stack-is-path[OF ne]  

172
by auto

moreover

from accepts have accept (hd ?ul) by simp

moreover have path E (hd ?ul) ?ul (hd ?ul)

proof (cases v = hd (stack s))

case True

with distinct-hd-tl stack-distinct have ul: ?ul = [hd (stack s)]

by force

from True uv-edg show ?thesis

by (subst ul)+ (simp add: path1)

next

case False with v-in ne have v ∈ set ?ur

by (auto simp add: neq-Nil-conv)

with P show ?thesis

by (fastforce intro: path-prepend dropWhileNot-path[where p=?ur]

uv-edg)

qed

ultimately have lasso-inv ?s by (simp add: lasso-inv-def)

}

moreover

{

assume accepts: accepts v

let ?vr = takeWhileNot v (rev (stack s))

let ?vl = dropWhileNot v (rev (stack s))

let ?s = s′ (lasso := Some (?vr, ?vl), red := red s)

assume DFS-invar G blue-dfs-params ?s

have [simp]: stack ?s = stack s

by (simp add: s′)

from ne v-in have hd-vl[simp]: hd ?vl = v

by (induct (stack s) rule: rev-nonempty-induct) auto

from v-in have ?vl ≠ [] by simp

moreover from hd-succ-stack-is-path[OF ne] uv-edg have

P: ∃v0∈V0. path E v0 (rev (stack s)) v

by auto

with ne v-in have ∃v0∈V0. path E v0 ?vr (hd ?vl)

by (force intro: takeWhileNot-path)

moreover from accepts have accepts (hd ?vl) by simp

moreover from P ne v-in have path E (hd ?vl) ?ul (hd ?ul)
by (force intro: dropWhileNot-path)

ultimately have lasso-inv ?s by (simp add: lasso-inv-def)

moreover
{
  assume ¬ accept (hd (stack s)) ¬ accept v
  let ?s = s'[state.more := state.more s']

  assume DFS-invar G blue-dfs-params ?s

  from assms have lasso-inv ?s
    by (auto simp add: lasso-inv-def)
}

ultimately show ?thesis
  using R s'
  unfolding se-back-edge-def
  by (auto split: option.splits)
qed

lemma lasso-inv:
  is-invar lasso-inv
proof (induct rule: establish-invarI)
case (finish s s' u)
  then interpret BlueDFS-inv where s'=s by simp

let ?onstack = λx. x ∈ set (stack s)
let ?rE = rel-restrict E (red s)
let ?revs = rev (tl (stack s))

note ne = (stack s ≠ []);
note [simp] = (u=hd (stack s);

from finish have [simp];
  ∀x. x = hd (stack s) ∨ x ∈ set (stack s') ↔ x ∈ set (stack s)
  red s' = red s
  lasso s' = lasso s
  by (auto simp: neq-Nil-conv)

{ fix v vs
  let ?cyc = vs @ dropWhileNot v ?revs
  let ?s = s'[lasso := Some (?revs, ?cyc), red := red s]
assume DFS-invar G blue-dfs-params ?s
and vs: vs ≠ [] path ?rE (hd (stack s)) vs v
and v: ?onstack v
and accept: accept (hd (stack s))
from vs have P: path E (hd (stack s)) vs v
  by (metis path-mono rel-restrict-sub)

have hds[simp]: hd vs = hd (stack s) hd ?cyc = hd (stack s)
using vs path-hd
by simp-all

from vs have ?cyc ≠ [] by simp

moreover have P0: ∃v0∈V0. path E v0 ?revs (hd ?cyc)
  using stack-is-path[OF ne]
  by auto

moreover from accept have accept (hd ?cyc) by simp

moreover have path E (hd ?cyc) ?cyc (hd ?cyc)
proof (cases tl (stack s) = [])
  case True with ne last-stack-in-V0 obtain v0 where v0 ∈ V0
    and [simp]: stack s = [v0]
    by (auto simp: neq-Nil-conv)
  with v True finish have [simp]: v = v0 by simp

from True P show ?thesis by simp
next
case False note tl-ne = this

show ?thesis
proof (cases v = hd (stack s))
  case True hence v ∉ set ?revs
    using ne stack-distinct by (auto simp: neq-Nil-conv)
  hence ?cyc = vs by fastforce
  with P True show ?thesis by (simp del: dropWhile-eq-Nil-conv)
next
case False with finish v have v ∈ set ?revs
  by (auto simp: neq-Nil-conv)
with tl-ne False P0 show ?thesis
  by (force intro: path-conc[OF P]
    dropWhileNot-path[where p=?revs])
qed

qed

ultimately have lasso-inv ?s by (simp add: lasso-inv-def)
}
hence accept (hd (stack s)) → lasso s = None →
  red-dfs (red s) ?onstack (hd (stack s)) ≤ SPEC (λrs. ∀ vs v.

175
\( rs = \text{Inr} (vs,v) \rightarrow \)
\[ \text{DFS-invar } G \text{ blue-dfs-params } (s' | \text{lasso} := \text{Some } (?\text{revs}, vs \odot \text{dropWhileNot } v ?\text{revs}), \text{red} := \text{red } s)) \rightarrow \]
\[ \text{lasso-inv } (s' | \text{lasso} := \text{Some } (?\text{revs}, vs \odot \text{dropWhileNot } v ?\text{revs}), \text{red} := \text{red } s)) \]

\[ \begin{align*}
\text{apply clarsimp} \\
\text{apply (rule find-path1-restr-spec-rule, intro conjI)} \\
\text{apply (rule red-DFS-precond, simp all add: ne)} \\
\text{apply (simp, rule red-DFS-precond, simp all add: ne)} \\
\text{using red-stack-disjoint ne}
\end{align*} \]

\[ \begin{align*}
\text{apply clarsimp} \\
\text{apply rprems} \\
\text{apply (simp all add: find-path1-restr-pred-def restr-invar-def)} \\
\text{apply (fastforce intro: path-restrict-tl rel-restrictI)} \\
\text{done}
\end{align*} \]

\text{note aux1 = this[rule format, THEN leaf-lift]}

\text{show ?case}
\[ \begin{align*}
\text{apply simp} \\
\text{unfolding run-red-dfs-def mk-blue-witness-def cyan-def}
\end{align*} \]

\[ \begin{align*}
\text{apply (simp add: run-red-dfs-def mk-blue-witness-def cyan-def)} \\
\text{apply (intro conjI impI)} \\
\text{apply (refine vcg leaf-trans[OF aux1])} \\
\text{using finish} \\
\text{apply (auto simp add: lasso-inv-def split: sum.split)} \\
\text{done}
\end{align*} \]

\text{next}

\text{case (back-edge } s s' u v \text{) then interpret BlueDFS-invar where } s=s \text{ by simp}

\[ \begin{align*}
\text{from back-edge se-back-edge-lasso-inv[THEN leaf-lift] show ?case} \\
\text{by auto}
\end{align*} \]

\text{qed simp all add: lasso-inv-def empty-state-def}

\text{end}

\text{context BlueDFS-invar}

\text{begin}

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

\text{lemma}

\text{assumes lasso s = Some (pr.pl)}
\text{shows loop-nonempty: pl \neq []} \\
\text{and accpt-loop: accpt (hd pl)} \\
\text{and loop-is-path: path } E \text{ (hd pl) pl (hd pl)} \\
\text{and loop-reachable: } \exists v0 \in V0. \text{ path } E \text{ v0 pr (hd pl)}
using assms s-lasso-inv
by (simp-all add: lasso-inv-def)

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

proof (cases lasso s)
  case None
  moreover
  \{ fix \(v v0\)
  assume \(v0 \in V0. (v0,v) \in E^* \land\) accept v \land (v,v) \in E^+
  moreover
  hence \(v \in\) reachable by (auto)
  with nc-finished-eq-reachable NC None have \(v \in\) dom (finished s)
  by simp
  moreover note no-accept-cycle-in-finish None
  ultimately have False by blast
  \}
  ultimately show ?thesis by auto
next
  case (Some prpl) with s-lasso-inv show ?thesis
  by (cases prpl)
    (auto intro: path-is-rtrancl path-is-trancl simp: lasso-inv-def)
qed

end

2.5.4 Interface

interpretation BlueDFS-defs for G accept.

definition nested-dfs-spec G accept \(\equiv\) \(\lambda r. \) case r of
  None \(\Rightarrow\) \(\neg\) (\(\exists v0 \in g-V0 G. \exists v. (v0,v) \in (g-E G)^* \land\) accept v \land (v,v) \in (g-E G)^+)
| Some (pr,pl) \(\Rightarrow\) (\(\exists v0 \in g-V0 G. \exists v.\) path (g-E G) v0 pr v \land\) accept v \land pl\(\neq\) [] \land path (g-E G) v pl v)

definition nested-dfs G accept \(\equiv\) do {\n  ASSERT (fb-graph G);
  s \(\leftarrow\) it-dfs TYPE('a) G accept;
  RETURN (lasso s)
  }

theorem nested-dfs-correct:
  assumes fb-graph G
  shows nested-dfs G accept \(\leq\) SPEC (nested-dfs-spec G accept)

177
proof
interpret fb-graph G by fact
interpret BlueDFS G accept by unfold-locales

show ?thesis
unfolding nested-dfs-def
apply (refine-reg refine-veg)
apply fact
apply (rule weaken-SPEC[OF it-dfs-correct])
apply clarsimp
proof
fix s
assume BlueDFS-invar G accept s
then interpret BlueDFS-invar G accept s.
assume ¬cond TYPE(’b) G accept s
from blue-dfs-correct[OF this] show nested-dfs-spec G accept (lasso s)
unfolding nested-dfs-spec-def by simp
qed
qed

2.5.5 Implementation
record ’v bdfs-state-impl = ’v simple-state +
lasso-impl :: (’v list × ’v list) option
red-impl :: ’v set

definition bdfs-erel ≡ {((lasso-impl=li,red-impl=ri),[lasso=l, red=r])
| li ri l r, li=lsi ∧ ri=rsi}
abbreviation bdfs-rel ≡ ⟨bdfs-erel⟩ simple-state-rel

definition mk-blue-witness-impl :: ’v bdfs-state-impl ⇒ ’v fpr-result ⇒ (’v,unit) bdfs-state-impl-ext
where
mk-blue-witness-impl s redS ≡
case redS of
    Inl R' ⇒ ⟨lasso-impl=li,red-impl=ri⟩,[lasso=l, red=r]
| Inr (vs, v) ⇒ let
rs = rev (map fst (CAST (ss-stack s)))
in ⟨
lasso-impl = Some (rs, vs@dropWhileNot v rs),
red-impl = red-impl s⟩

lemma mk-blue-witness-impl[refine]:
[(si,s)∈bdfs-rel; (ri,r)∈(Id, (Id)list-rel ×, Id)sum-rel ]
→ (mk-blue-witness-impl si ri, mk-blue-witness s r)∈bdfs-erel
unfolding mk-blue-witness-impl-def mk-blue-witness-def
apply parametricity
apply (cases si, cases s)
apply (auto simp: bdfs-erel-def simple-state-rel-def) []
apply (rule introR[where R=(Id)list-rel])
apply (cases si, cases s)
apply (auto simp: bdfs-erel-def simple-state-rel-def comp-def) []
apply (cases si, cases s)
apply (auto simp: bdfs-erel-def simple-state-rel-def) []
done
definition cyan-impl s ≡ on-stack s
lemma cyan-impl : [(si,s)∈bdfs-rel] ⇒ (cyan-impl si, cyan s)∈Id
  unfolding cyan-impl-def cyan-def
  by (auto simp: bdfs-erel-def simple-state-rel-def)
definition run-red-dfs-impl :: ('v, 'more) graph-rec-scheme ⇒ 'v ⇒ 'v bdfs-state-impl ⇒ ('v,unit) bdfs-state-impl-ext
where
  run-red-dfs-impl G u s ≡ case lasso-impl s of None ⇒ do {
    redS ← red-dfs TYPE('more) G (red-impl s) (lx. x = u ∨ x ∈ cyan-impl)
  } u;
    RETURN (mk-blue-witness-impl s redS)
  | - ⇒ RETURN (simple-state.more s)
lemma run-red-dfs-impl : [(Gi,G)∈Id; (ui,u)∈Id; (si,s)∈bdfs-rel] ⇒ run-red-dfs-impl Gi ui si ≤⇓ bdfs-erel (run-red-dfs TYPE('a) G u s)
  unfolding run-red-dfs-impl-def run-red-dfs-def
  apply refine-rcg
  apply refine-dref-type
  apply (cases si, cases s, auto simp: bdfs-erel-def simple-state-rel-def) []
  apply (cases si, cases s, auto simp: bdfs-erel-def simple-state-rel-def cyan-impl-def cyan-def) []
  apply (auto simp: bdfs-erel-def simple-state-rel-def) []
done
definition se-back-edge-impl accept u v s ≡ case lasso-impl s of None ⇒
if accept u then
  let rs = rev (map fst (tl (CAST (ss-stack s))));
  ur = rs;
  ul = u#dropWhileNot v rs
  in RETURN (lasso-impl = Some (ur,ul), red-impl = red-impl s)
else if accept v then
  let rs = rev (map fst (CAST (ss-stack s)));
  vr = takeWhileNot v rs;
  vl = dropWhileNot v rs
  in RETURN (lasso-impl = Some (vr,vl), red-impl = red-impl s)
else RETURN (simple-state.more s)
lemma se-back-edge-impl[refine]: \((\text{accept}, \text{accept}) \in \text{Id}; (\text{ui}, \text{u}) \in \text{Id}; (\text{vi}, \text{v}) \in \text{Id}; (\text{si}, \text{s}) \in \text{bdfs-rel}) \Rightarrow \text{se-back-edge-impl accept ui vi si} \leq \downarrow \text{bdfs-erel (se-back-edge accept u v s)}

unfolding se-back-edge-impl-def se-back-edge-def
apply refine-rcg
apply refine-dref-type
apply simp-all
apply (simp-all add: bdfs-erel-def simple-state-rel-def)
apply (cases si, cases s, (auto []))
done

lemma NOOP-impl: \((\text{si}, \text{s}) \in \text{bdfs-rel}) \Rightarrow \text{RETURN (simple-state.more si)} \leq \downarrow \text{bdfs-erel (NOOP s)}
apply (simp add: pw-le-iff refine-pw-simps)
apply (auto simp: simple-state-rel-def)
done

definition bdfs-params-impl :: \((\text{v}', \text{more}) \gamma \text{graph-rec-scheme} \Rightarrow (\text{v}' \Rightarrow \text{bool}) \Rightarrow (\text{v}', \text{v} \text{bdfs-state-impl},(\text{v}', \text{unit}) \text{bdfs-state-impl-ext})\gamma\text{gen-parameterization}
where bdfs-params-impl G accpt \equiv \{
  on-init = \text{RETURN } (\text{lasso-impl }= \text{None}, \text{red-impl }= \{\}),
  on-new-root = \lambda \text{v0 s}. \text{RETURN (simple-state.more s)},
  on-discover = \lambda \text{u v s}. \text{RETURN (simple-state.more s)},
  on-finish = \lambda \text{s}. 
  if \text{accept u} then \text{run-red-dfs-impl G u s} else \text{RETURN (simple-state.more s)},
  on-back-edge = \text{se-back-edge-impl accept},
  on-cross-edge = \lambda \text{u v s}. \text{RETURN (simple-state.more s)},
  is-break = \lambda \text{s}. \text{lasso-impl s }\neq \text{None }\}
lemmas bdfs-params-impl-simps[simp, DFS-code-unfold] =
gen-parameterization.simps[mk-record-simp, OF bdfs-params-impl-def]

interpretation impl: simple-impl-defs G bdfs-params-impl G accept blue-dfs-params
TYPE(\text{a}) G accpt
for G accpt .

context BlueDFS begin

sublocale impl: simple-impl G blue-dfs-params bdfs-params-impl G accept bdfs-erel
apply unfold-locales
apply (simp-all
   add: bdfs-params-impl-def run-red-dfs-impl se-back-edge-impl NOOP-impl)
apply parametricity
apply (clarsimp-all simp:
   pw-le-iff refine-pw-simps bdfs-erel-def simple-state-rel-def)
apply (rename-tac si s x y,
   case-tac si, case-tac s)
apply (auto simp add: bdfs-erel-def simple-state-rel-def) []
done

lemmas impl = impl.simple-tailrec-refine

end

definition nested-dfs-impl G accpt ≡ do
  ASSERT (fb-graph G);
  s ← Impl.tailrec-impl TYPE(ʻa) G accpt;
  RETURN (lasso-impl s)

lemma nested-dfs-impl[refine]:
  assumes (Gi,G)∈Id
  assumes (accepti,accept)∈Id
  shows nested-dfs-impl Gi accepti ≤⇓(⟨⟨Id⟩ list-rel × r ⟨Id⟩ list-rel⟩ option-rel)
    (nested-dfs G accept)
  using assms
  unfolding nested-dfs-impl-def nested-dfs-def
  apply refine-reg
  apply simp-all
  apply (rule intro-pryR[where R=bdfs-rel])
  defer
  apply (rename-tac si s)
  apply (case-tac si, case-tac s)
  apply (auto simp: bdfs-erel-def simple-state-rel-def) []
proof –
  assume fb-graph G
  then interpret fb-graph G .
  interpret BlueDFS G by unfold-locales

from Impl show Impl.tailrec-impl TYPE(ʻb) G accept ≤⇓ bdfs-rel (it-dfs TYPE(ʻb)
  G accept) .
qed

2.5.6 Synthesis of Executable Code

record (ʻv,ʼsi,ʼnsi)bdfs-state-impl' = (ʼsi,ʼnsi)simple-state-impl +
  lasso-impl' :: (ʼv list × ʼv list) option
  red-impl' :: ʼnsi

definition [to-relAPP]: bdfs-state-erel' Vi ≡ {
  (lasso-impl' = li, red-impl' = ri), (lasso-impl = l, red-impl = r) | li ri l r.
  (li,l)∈(⟨Vi⟩ list-rel × r ⟨Vi⟩ list-rel) option-rel ∧ (ri,r)∈⟨Vi⟩ dfl-R-ahs-rel}
consts
  i-bdfs-state-ext :: interface ⇒ interface

lemmas [autoref-rel-intf] = REL-INTFI[of bdfs-state-erel' i-bdfs-state-ext]

lemma [autoref-rules]:
  fixes ns-rel vis-rel Vi
  defines R ≡ ⟨ns-rel, vis-rel, ⟨Vi⟩ bdfs-state-erel'⟩ ss-impl-rel
  shows
  (bdfs-state-impl'-ext, bdfs-state-impl-ext)
  ∈ ⟨⟨Vi⟩ list-rel ×, ⟨Vi⟩ list-rel⟩ option-rel → ⟨Vi⟩ dflt-ahs-rel → unit-rel →
  ⟨Vi⟩ bdfs-state-erel'
  (lasso-impl', lasso-impl) ∈ R → ⟨⟨Vi⟩ list-rel ×, ⟨Vi⟩ list-rel⟩ option-rel
  (red-impl', red-impl) ∈ R → ⟨Vi⟩ dflt-ahs-rel
  unfolding bdfs-state-erel'-def ss-impl-rel-def R-def
  by auto

schematic-goal nested-dfs-code:
  assumes Vid: V = (Id :: ('v::hashable × 'v) set)
  assumes [unfolded Vid, autoref-rules]:
    (Ḡ, G) ∈ ⟨Rm, V⟩ g-impl-rel-ext
    (accept, accept) ∈ (V → bool-rel)
  notes [unfolded Vid, autoref-tyrel] =
    TYRELI[where R=⟨V⟩ dflt-ahs-rel]
    TYRELI[where R=⟨V⟩ ras-rel]
  shows (nres-of ?c, nested-dfs-impl G accept)
  ∈ ⟨⟨V⟩ list-rel ×, ⟨V⟩ list-rel⟩ option-rel)nres-rel
  unfolding nested-dfs-impl-def[abs-def] Vid
  se-back-edge-impl-def run-red-dfs-impl-def mk-blue-witness-impl-def
cyan-impl-def
  DFS-code-unfold

  using [[autoref-trace-failed-id]]
  apply (autoref-monadic (trac))
  done

concrete-definition nested-dfs-code uses nested-dfs-code

export-code nested-dfs-code checking SML

2.5.7 Conclusion

We have implemented an efficiently executable nested DFS algorithm. The
following theorem declares this implementation to the Autoref tool, such
that it uses it to synthesize efficient code for nested-dfs. Note that you will
need the lemma nested-dfs-correct to link nested-dfs to an abstract specifi-
cation, which is usually done in a previous refinement step.
theorem nested-dfs-autoref[autoref-rules]:
assumes PREFER-id V
shows (λ G accpt. nres-of (nested-dfs-code G accpt),nested-dfs) ∈
(Rm, V)g-impl-rel-ext → (V → bool-rel) →
⟨⟨(V)list-rel × r (V)list-rel)option-rel⟩nres-rel
proof –
from assms have Vid: V=Id by simp
note nested-dfs-code.refine[OF Vid,param-fo, THEN nres-relD]
also note nested-dfs-impl
finally show \text{thesis} by (fastforce intro: nres-relI)
qed

2.6 Invariants for Tarjan’s Algorithm

theory Tarjan-LowLink
imports
../DFS-Framework
../Invars/DFS-Invars-SCC
begin
context begin interpretation timing-syntax.
abbreviation LowLink where
LowLink s v ≡ Min (δ s ' lowlink-set s v)
end
context DFS-invar begin
lemma lowlink-setI:
assumes lowlink-path s v p w

183
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 (cases $v = w$
  case True thus $?\text{thesis} \ by \ (\text{simp add: \text{lowlink-set-def} \ assms})$
next
  case False with assms have $(v,w) \in E^+ \ (w,v) \in E^+$ by (metis rtrancl-eq-or-trancl)+
    with assms show $?\text{thesis} \ by \ (\text{auto \ simp add: \text{lowlink-set-def}})$
qed

lemma lowlink-set-discovered:
  lowlink-set $s \ v \subseteq \text{dom} \ (\text{discovered} \ s)$
unfolding lowlink-set-def
by blast

lemma lowlink-set-finite[simp, intro!]:
  finite (lowlink-set $s \ v$)
using lowlink-set-discovered discovered-finite
by (metis finite-subset)

lemma lowlink-set-not-empty:
  assumes $v \in \text{dom} \ (\text{discovered} \ s)$
shows lowlink-set $s \ v \neq \{}$
unfolding lowlink-set-def
using assms by auto

lemma lowlink-path-single:
  assumes $(v,w) \in \text{cross-edges} \ s \cup \text{back-edges} \ s$
shows lowlink-path $s \ v \ [v] \ w$
unfolding lowlink-path-def
using assms cross-edges-ssE back-edges-ssE
by (auto simp add: path-simps)

lemma lowlink-path-Cons:
  assumes lowlink-path $s \ v \ (x\#xs) \ w$
and $xs \neq []$
shows $\exists u. \ \text{lowlink-path} \ s \ u \ xs \ w$
proof
  from assms have path: $\text{path} \ E \ v \ (x\#xs) \ w$
    and cb: $(\text{last} \ xs, \ w) \in \text{cross-edges} \ s \cup \text{back-edges} \ s$
    and f: $(x\#xs)!1 \in \text{dom} \ (\text{finished} \ s)$
    and t: $(\forall k < \text{length} \ xs. \ ((x\#xs)!k, \ (x\#xs)!\text{Suc} \ k) \in \text{tree-edges} \ s)$
    unfolding lowlink-path-def
    by auto

  from path obtain u where path $E \ u \ xs \ w$ by (auto simp add: path-simps)
moreover note cb $[xs \neq []$
moreover { fix k define $k' \ where \ k' = \text{Suc} \ k$
  assume $k < \text{length} \ xs - 1$
with \( k' \)-def have \( k' < \text{length} \, xs \) by simp

with \( t \) have \((x\#xs)!k', (x\#xs)!\text{Suc} \, k') \in \text{tree-edges} \, s \) by simp

hence \((xs!k,xs)!\text{Suc} \, k) \in \text{tree-edges} \, s \) by (simp add: \( k' \)-def nth-Cons)

} note \( t' = \text{this} \)

moreover {
  assume \( \ast: \text{length} \, xs > 1 \)
  from \( f \) have \( xs!0 \in \text{dom} \, (\text{finished} \, s) \) by auto

  ultimately have \( \ast: (xs!0,xs!1) \in \text{tree-edges} \, s \) by (simp add: lowlink-path-def)
}

ultimately have \( \text{lowlink-path} \, s \, u \, xs \, w \) by (auto simp add: lowlink-path-def)

thus \( \text{thesis} \) ..

qed

lemma lowlink-path-in-tree:
assumes \( p: \text{lowlink-path} \, s \, v \, p \, w \)

and \( j: j < \text{length} \, p \)

and \( k: k < j \)

shows \((p!k, p!j) \in (\text{tree-edges} \, s)^+ \)

proof 
  from \( p \) have \( p \neq [] \) by (auto simp add: lowlink-path-def)
  thus \( \text{thesis} \) using \( p \, j \, k \)

proof (induction arbitrary: \( v \, j \, k \) rule: list-nonempty-induct)
  case single
  thus \( \text{case} \) by auto

next
  case (cons \( x \) \( xs \))
  define \( j' \) where \( j' = j - 1 \)

  with \( \text{cons} \) have \( j'\text{-le: } j' < \text{length} \, xs \ \text{and} \ k \leq j' \ \text{and} \ j = \text{Suc} \, j' \) by auto

  from \( \text{cons} \) \( \text{lowlink-path-Cons} \) obtain \( u \) where \( \text{where} \, p: \text{lowlink-path} \, s \, u \, xs \, w \) by blast

  show \( \text{?case} \)

  proof (cases \( k = 0 \))
    case True
    from \( \text{cons} \) have \( \land k. k < \text{length} \, (x\#xs) - 1 \implies ((x\#xs)!k,(x\#xs)!\text{Suc} \, k) \in \text{tree-edges} \, s \)
      unfolding lowlink-path-def
      by auto

    moreover from \( \text{True cons} \) have \( k < \text{length} \, (x\#xs) - 1 \) by auto

    ultimately have \( \ast: ((x\#xs)!k,(x\#xs)!\text{Suc} \, k) \in \text{tree-edges} \, s \) by metis

  show \( \text{?thesis} \)

  proof (cases \( j' = 0 \))
    case True with \( \ast \, j \, (k = 0) \): show \( \text{?thesis} \) by simp

next
  case False with \( \text{True have} \, j' \, > \, k \) by simp
with \( \text{cons.IH}[OF p \ j'-le] \) have \((xs!k, xs!j')\) \(\in\) \((\text{tree-edges} s)^+\).
with \( j \) have \((x#xs)!\text{Suc} k, (x#xs)!j)\) \(\in\) \((\text{tree-edges} s)^+\) by simp
with \( * \) show \(?thesis\) by (metis trancl-into-trancl2)
qed

next

\text{case \textit{False}}

\text{define} \( k' \) \text{where} \( k' = k - 1 \)
with \textit{False} \( \langle k \leq j' \rangle \) have \( k' < j' \) \text{and} \( k = \text{Suc} k' \) by simp-all
with \text{cons.IH}[OF p \ j'-le] have \((xs!k', xs!j')\) \(\in\) \((\text{tree-edges} s)^+\) by metis

\text{hence} \((x#xs)!\text{Suc} k', (x#xs)!j') \(\in\) \((\text{tree-edges} s)^+\) by simp

\text{with} \( k \ j \) show \(?thesis\) by simp
qed

\text{lemma \textit{lowlink-path-finished}}:
\text{assumes} \( p \colon \text{lowlink-path} s \ v \ p \ w \)
and \( j \colon j < \text{length} p \ j > 0 \)
\text{shows} \( p!j \in \text{dom} \ (\text{finished} s) \)
\text{proof –}
from \( j \) have \( \text{length} p > 1 \) by simp
with \( p \) have \( f \colon p!1 \in \text{dom} \ (\text{finished} s) \) by (simp add: \text{lowlink-path-def})
thus \(?thesis\)
proof (cases \( j=1 \))
\text{case \textit{False}} with \( j \) have \( j > 1 \) by simp
with \text{assms} lowlink-path-in-tree[\text{where} \( k=1 \)] have \((p!1, p!j)\) \(\in\) \((\text{tree-edges} s)^+\)
by simp
with \( f \) \text{tree-path-impl-parenthesis show} \(?thesis\) by simp
qed simp
qed

\text{lemma \textit{lowlink-path-tree-prepend}}:
\text{assumes} \( p \colon \text{lowlink-path} s \ v \ p \ w \)
and \((\text{tree-edges}) : (u, v) \in (\text{tree-edges} s)^+\)
and \( \text{fin} : u \in \text{dom} \ (\text{finished} s) \lor (\text{stack} s \neq [] \land u = \text{hd} \ (\text{stack} s)) \)
\text{shows} \( \exists p. \ \text{lowlink-path} s u p w \)
\text{proof –}
\text{note} \text{lowlink-path-def[simp]}
from \text{tree-edges trancl-is-path} obtain \( tp \) \text{where}
\( tp : \text{path} \ (\text{tree-edges} s) u v \end{array} \)
\text{by metis}
from \text{tree-path-impl-parenthesis \text{assms} hd-stack-tree-path-finished have}
\( v\text{-fin} : v \in \text{dom} \ (\text{finished} s) \) by blast
from \( p \) have \( p!0 = \text{hd} p \) by (simp add: \text{hd-conv-nth})
with \( p \) have \( p!0 = v \) by (auto simp add: \text{path-hd})
let \(?p = tp \# p\)

\[
\{
\text{from } tp \text{ path-mono[OF tree-edges-ssE]} \text{ have } \text{path } E u \ tp \ v \text{ by simp}
\text{also from } p \text{ have } \text{path } E \ v \ p \ w \text{ by simp}
\text{finally have } \text{path } E \ u \ ?p \ w .
\}
\]

moreover from \(p\) have \(?p \neq []\) by \(\text{simp}\)

moreover from \(p\) have \((\text{last } ?p, w) \in \text{cross-edges } s \cup \text{back-edges } s\) by \(\text{simp}\)

moreover \{
\text{assume } length \ ?p > 1
\text{have } ?p ! 1 \in \text{dom } (\text{finished } s)
\text{proof } (\text{cases } \text{length } tp > 1)
\text{case } \text{True hence } tp1: ?p ! 1 = tp ! 1 \text{ by } (\text{simp add: nth-append})
\text{from } tp \text{ True have } (tp ! 0, \ tp ! 1) \in (\text{tree-edges } s)^+
\text{by } (\text{auto simp add: path-nth-conv nth-append elim: allE[where } x=0])
\text{also from } \text{True have } tp ! 0 = hd \ tp \text{ by } (\text{simp add: hd-conv-nth})
\text{also from } tp \text{ have } hd \ tp = u \text{ by } (\text{simp add: path-hd})
\text{finally have } tp ! 1 \in \text{dom } (\text{finished } s)
\text{using } \text{tree-path-impl-parenthesis fin hd-stack-tree-path-finished by blast}
\text{thus } \#\text{thesis by } (\text{subst } tp1)
\text{next}
\text{case } \text{False with } tp \text{ have } \text{length } tp = 1 \text{ by } (\text{cases } tp) \text{ auto}
\text{with } p-0 \text{ have } ?p ! 1 = v \text{ by } (\text{simp add: nth-append})
\text{thus } \#\text{thesis by } (\text{simp add: v-fin})
\text{qed}
\]

also have \(\forall k < \text{length } ?p - 1. \ (\#p!k, \ #p!\text{Suc } k) \in \text{tree-edges } s\)
\text{proof } (\text{safe})
\text{fix } k
\text{assume } A: k < \text{length } ?p - 1
\text{show } (\#p!k, \ #p!\text{Suc } k) \in \text{tree-edges } s
\text{proof } (\text{cases } k < \text{length } tp)
\text{case } \text{True hence } k: ?p ! k = tp ! k \text{ by } (\text{simp add: nth-append})
\text{show } \#\text{thesis}
\text{proof } (\text{cases } \text{Suc } k < \text{length } tp)
\text{case } \text{True hence } ?p ! \text{Suc } k = tp ! \text{Suc } k \text{ by } (\text{simp add: nth-append})
\text{moreover from } \text{True } tp \text{ have } (tp ! k, \ tp ! \text{Suc } k) \in \text{tree-edges } s
\text{by } (\text{auto simp add: path-nth-conv nth-append elim: allE[where } x=k])
\text{ultimately show } \#\text{thesis using } k \text{ by } \text{simp}
\text{next}
\text{case } \text{False with } \text{True have } *: \text{Suc } k = \text{length } tp \text{ by } \text{simp}
\text{with } tp \text{ True have } (tp ! k, \ v) \in \text{tree-edges } s
by (auto simp add: path-nth-conv nth-append
  elim: allE[where x=k])
also from * p-0 have v = ?p ! Suc k by (simp add: nth-append)
finally show ?thesis by (simp add: k)
qed
next
case False hence *: Suc k - length tp = Suc (k - length tp) by simp
define k' where k' = k - length tp
with False * have k': ?p ! k = p ! k' ?p ! Suc k = p ! Suc k'
  by (simp-all add: nth-append)
from k'-def False A have k' < length p - 1 by simp
with p have (p!k', p!Suc k') ∈ tree-edges s by simp
with k' show ?thesis by simp
qed
qed

also (conjI) note calculation

ultimately have lowlink-path s u ?p w by simp
thus ?thesis ..
qed

lemma lowlink-path-complex:
  assumes (u,v) ∈ (tree-edges s)+
  and u ∈ dom (finished s) ∨ (stack s ≠ [] ∧ u = hd (stack s))
  and (v,w) ∈ cross-edges s ∪ back-edges s
  shows ∃ p. lowlink-path s u p w
proof -
  from assms lowlink-path-single have lowlink-path s v [v] w by simp
  with assms lowlink-path-tree-prepend show ?thesis by simp
qed

lemma no-path-imp-no-lowlink-path:
  assumes edges s " {v} = {}"
  shows ¬lowlink-path s v p w
proof
  assume p: lowlink-path s v p w
  hence [simp]: p≠[] by (simp add: lowlink-path-def)
from p have hd p = v by (auto simp add: lowlink-path-def path-hd)
with hd-conv-nth[OF p≠[]] have v: p!0 = v by simp
show False
proof (cases length p > 1)
case True with p have (p!0,p!1) ∈ tree-edges s by (simp add: lowlink-path-def)
  with v assms show False by auto
next

188
case False with (p ≠ []) have length p = 1 by (cases p) auto

hence last p = v by (simp add: last-conv-nth v)

with p have (v, w) ∈ edges s by (simp add: lowlink-path-def)

with assms show False by auto

qed

qed

context begin interpretation timing-syntax .

lemma LowLink-le-disc:
  assumes v ∈ dom (discovered s)
  shows LowLink s v ≤ δ s v
  using assms
  unfolding lowlink-set-def
  by clarsimp

lemma LowLink-lessE:
  assumes LowLink s v < x
  and v ∈ dom (discovered s)
  obtains w where δ s w < x w ∈ lowlink-set s v
  proof
    let ?L = δ s ' lowlink-set s v
    note assms
    moreover from lowlink-set-finite have finite ?L by auto
    moreover from lowlink-set-not-empty assms have ?L ≠ {} by auto
    ultimately show ?thesis using that by (auto simp: Min-less-iff)
  qed

lemma LowLink-lessI:
  assumes y ∈ lowlink-set s v
  and δ s y < δ s v
  shows LowLink s v < δ s v
  proof
    let ?L = δ s ' lowlink-set s v
    from assms have δ s y ∈ ?L by simp
    moreover hence ?L ≠ {} by auto
    moreover from lowlink-set-finite have finite ?L by auto
    ultimately show ?thesis
      by (metis Min-less-iff assms(2))
  qed

lemma LowLink-eqI:
  assumes DFS-invar G param s'
  assumes sub-m: discovered s ⊆ m discovered s'
  assumes sub: lowlink-set s w ⊆ lowlink-set s' w
  and rev-sub: lowlink-set s' w ⊆ lowlink-set s w ∪ X
  and w-disc: w ∈ dom (discovered s)
and $X: \forall x. [x \in X; x \in \text{lowlink-set } s' w] \implies \delta s' x \geq \text{LowLink } s w$

shows $\text{LowLink } s w = \text{LowLink } s' w$

proof (rule ccontr)

interpret $s'$: DFS-invar where $s=s'$ by fact

assume $A$: $\text{LowLink } s w \neq \text{LowLink } s' w$

from lowlink-set-discovered sub sub-m w-disc have

$\text{sub}': (\delta s \cdot \text{lowlink-set } s w \subseteq \delta s' \cdot \text{lowlink-set } s' w)$

and $w\text{-disc}'$: $w \in \text{dom (discovered } s')$

and $eq: \forall ll. ll \in \text{lowlink-set } s w \implies \delta s' ll = \delta s ll$

by (force simp: map-le-def+)

from lowlink-set-not-empty[OF w-disc] A Min-antimono[OF sub'] $s', \text{lowlink-set-finite}$

have $\text{LowLink } s' w < \text{LowLink } s w$ by fastforce

then obtain $ll$ where $ll\in \text{lowlink-set } s' w$ and $ll\text{-le}': \delta s' ll < \text{LowLink } s w$

by (metis $s'$,LowLink-lessE w-disc')

with rev-sub have $ll \in \text{lowlink-set } s w \lor ll \in X$ by auto

hence $\text{LowLink } s w \leq \delta s' ll$

proof

assume $ll \in \text{lowlink-set } s w$ with $\text{lowlink-set-finite}$ eq show $\text{thesis}$ by force

next

assume $ll \in X$ with $ll$ show $\text{thesis}$ by (metis X)

qed

with $ll\text{-le}$ show $\text{False}$ by simp

qed

lemma LowLink-eq-disc-iff-scc-root:

assumes $v \in \text{dom (finished } s) \lor (\text{stack } s \neq [] \land v = \text{hd (stack } s) \land \text{pending } s$:: $\{v\} = \{\})$

shows $\text{LowLink } s v = \delta s v \iff \text{scc-root } s v (\text{scc-of } E v)$

proof

let $?scc = \text{scc-of } E v$

assume $scc$: $\text{scc-root } s v$ $?scc$

show $\text{LowLink } s v = \delta s v$

proof (rule ccontr)

assume $A$: $\text{LowLink } s v \neq \delta s v$

from assms finished-discovered stack-discovered hd-in-set have disc: $v \in \text{dom (discovered } s)$ by blast

with assms $\text{LowLink-le-disc}$ $A$ have $\text{LowLink } s v < \delta s v$ by force

with disc obtain $w$ where

$w$: $\delta s w < \delta s v w \in \text{lowlink-set } s v$

by (metis LowLink-lessE)

with lowlink-set-discovered have wdisc: $w \in \text{dom (discovered } s)$ by auto

from $w$ have $(v,w) \in E^* (w,v) \in E^*$ by (auto simp add: lowlink-set-def)

moreover have is-scc $E$ $?scc v \in $?scc by simp-all

ultimately have $w \in $?scc by (metis is-scc-closed)
with wdisc scc-root-disc-le[OF scc] have δ s v ≤ δ s w by simp
with w show False by auto
qed
next
assume LL: LowLink s v = δ s v
from assms finished-discovered stack-discovered hd-in-set have v-disc: v ∈ dom (discovered s) by blast
from assms finished-no-pending have v-no-p: pending s "{v} = {}" by blast
let ?scc = scc-of E v
have is-scc: is-scc E ?scc by simp
{
  fix r
  assume r ≠ v
  and r ∈ ?scc r ∈ dom (discovered s)
  have v ∈ ?scc by simp
  with (r ∈ ?scc) is-scc have (v,r) ∈ (Restr E ?scc)∗
  by (simp add: is-scc-connected*)
  hence (v,r) ∈ (tree-edges s)⁺ using (r≠v)
proof (induction rule: rtrancl-induct)
case (step y z) hence (v,z) ∈ (Restr E ?scc)∗
  by (metis rtrancl-into-rtrancl)
hence (v,z) ∈ E∗ by (metis Restr-rtrancl-mono)
from step have (z,v) ∈ E∗ by (simp add: is-scc-connected[OF is-scc])
{
  assume z-disc: z ∈ dom (discovered s)
  and ∃ p. lowlink-path s v p z
  with (z,v)∈E⁺ (v,z)∈E⁺ have ll: z ∈ lowlink-set s v
  by (metis lowlink-setI)
  have δ s v < δ s z
  proof (rule ccontr)
    presume δ s v ≥ δ s z with (z≠v) v-disc z-disc disc-unequal have δ s z < δ s v by fastforce
    with ll have LowLink s v < δ s v by (metis LowLink-lessI)
    with LL show False by simp
  qed simp
} note δz = this
show ?case
proof (cases y=v)
case True note [simp] = this
}
with step v-no-p v-disc no-pending-impl-succ-discovered have
z-disc: \( z \in \text{dom}(\text{discovered } s) \) by blast

from step edges-covered v-no-p v-disc have \((v,z) \in \text{edges } s\) by auto
thus \(?\text{thesis}\)
proof (rule edgesE-CB)
  assume \((v,z) \in \text{tree-edges } s\) thus \(?\text{thesis}\)
next
  assume CB: \((v,z) \in \text{cross-edges } s \cup \text{back-edges } s\)
hence lowlink-path s v \([v] z\)
  by (simp add: lowlink-path-single)
  with \(\delta z[\text{OF } z\text{-disc}]\) no-pending-succ-impl-path-in-tree v-disc v-no-p step
show \(?\text{thesis}\)
  by auto
qed

next
  case False with step.IH have \(T: (v,y) \in (\text{tree-edges } s)^+\),
  with \(\text{tree-path-impl-parenthesis} \text{ assms hd-stack-tree-path-finished} \text{ tree-path-disc}\)
  have
    y-fin: \( y \in \text{dom}(\text{finished } s) \)
    and y-\(\delta\) : \( \delta s v < \delta s y \) by blast+
  with step have \( z\text{-disc: } z \in \text{dom}(\text{discovered } s) \)
    using finished-imp-succ-discovered
    by auto
  from step edges-covered finished-no-pending[of y] y-fin finished-discovered
  have
    \((y,z) \in \text{edges } s\)
    by fast
  thus \(?\text{thesis}\)
proof (rule edgesE-CB)
  assume \((y,z) \in \text{tree-edges } s\) with \(T\) show \(?\text{thesis}\)
next
  assume CB: \((y,z) \in \text{cross-edges } s \cup \text{back-edges } s\)
  with lowlink-path-complex[of \(T\)] assms have
    \(\exists p. \text{lowlink-path } s v p z\) by blast
  with \(\delta z\) \(z\text{-disc have } \delta z: \delta s v < \delta s z\) by simp
show \(?\text{thesis}\)
proof (cases \(v \in \text{dom}(\text{finished } s))\)
  case True with \(\text{tree-path-impl-parenthesis} \text{ \(T\) have } y\text{-f}: \varphi s y < \varphi s v\)
  by blast
  from \(C\) show \(?\text{thesis}\)
proof
  assume C: \((y,z) \in \text{cross-edges } s\)
  with \(\text{cross-edges-finished-decr}\ y\text{-f} \text{ have } \varphi s z < \varphi s v\)
    by force
  moreover note \(\delta z\)
moreover from $C$ cross-edges-target-finished have
$z \in \text{dom}(\text{finished} \ s)$ by simp
ultimately show $\text{thesis}$
using parenthesis-impl-tree-path[OF True] by metis
next
assume $B$: $(y, z) \in \text{back-edges} \ s$
with back-edge-disc-lt-fin $y$-fin $y\cdot f$ have $\delta \ s \ z < \varphi \ s \ v$
by force
moreover note $\delta z$-disc
ultimately have $z \in \text{dom}(\text{finished} \ s)$ $\varphi \ s \ z < \varphi \ s \ v$
using parenthesis-contained[OF True] by simp-all
with $\delta z$ show $\text{thesis}$
using parenthesis-impl-tree-path[OF True] by metis
qed
next
case False — $v \notin \text{dom}(\text{finished} \ s)$
with assms have $st$: stack $s \neq [] \ v = \text{hd}(\text{stack} \ s)$ pending $s$ " {} =
{ } by blast+
have $z \in \text{dom}(\text{finished} \ s)$
proof (rule econtr)
assume $z \notin \text{dom}(\text{finished} \ s)$
with $z$-disc have $z \in \text{set}(\text{stack} \ s)$ by (simp add: stack-set-def)
with ($z \neq v$ $st$ have $z \in \text{set}(\text{tl}(\text{stack} \ s))$ by (cases stack $s$) auto
with $st$ tl-lt-stack-hd-discover $\delta z$ show False by force
qed
with $\delta z$ parenthesis-impl-tree-path-not-finished $v$-disc False show $\text{thesis}$
by simp
qed
qed
qed simp
hence $r \in (\text{tree-edges} \ s)^* $ " {} by auto
}
hence $\text{scc} \cap \text{dom}(\text{discovered} \ s) \subseteq (\text{tree-edges} \ s)^* $ " {} by fastforce
thus $\text{scc-root} \ s \ v \ ?\text{scc}$ by (auto intro!: scc-rootI $v$-disc)
qed
end end

2.7 Tarjan’s Algorithm

theory Tarjan
imports
  Tarjan-LowLink
begin

We use the DFS Framework to implement Tarjan’s algorithm. Note that,
currently, we only provide an abstract version, and no refinement to efficient
code.

2.7.1 Preliminaries

lemma tjs-union:
  fixes tjs u
defines dw ≡ dropWhile ((≠) u) tjs
defines tw ≡ takeWhile ((≠) u) tjs
assumes u ∈ set tjs
shows set tjs = set (tl dw) ∪ insert u (set tw)
proof –
  from takeWhile-dropWhile-id have set tjs = set (tw@dw) by (auto simp: dw-def tw-def)
    hence set tjs = set tw ∪ set dw by (metis set-append)
  moreover from ⟨u ∈ set tjs⟩ dropWhile-nil-conv have dw ≠ [] by (auto simp: dw-def)
    from hd-dropWhile[OF this[unfolded dw-def]] have hd dw = u by (simp add: dw-def)
      with ⟨dw ≠ []⟩ have set dw = insert u (set (tl dw)) by (cases dw) auto
  ultimately show ?thesis by blast
qed

2.7.2 Instantiation of the DFS-Framework

record ′v tarjan-state = ′v state +
sccs :: ′v set set
lowlink :: ′v ⇒ nat
tj-stack :: ′v list
type-synonym ′v tarjan-param = (′v, (′v, unit) tarjan-state-ext) parameterization
abbreviation the-lowlink s v ≡ the (lowlink s v)

context timing-syntax
begin
  notation the-lowlink (ζ)
end
locale Tarjan-def = graph-defs G
  for G :: (′v, ′more) graph-rec-scheme
begin
  context begin interpretation timing-syntax.
  definition tarjan-disc :: ′v ⇒ ′v tarjan-state ⇒ (′v, unit) tarjan-state-ext nres
    where
      tarjan-disc v s = RETURN ( sccs = sccs s,
                               lowlink = (lowlink s)(v ⇒ δ s v),
                               tj-stack = v#tj-stack s)
definition tj-stack-pop :: 'v list ⇒ 'v ⇒ ('v list × 'v set) nres where
tj-stack-pop tjs u = RETURN (tl (dropWhile ((≠) u) tjs), insert u (set (takeWhile ((≠) u) tjs)))

lemma tj-stack-pop-set:
tj-stack-pop tjs u ≤ SPEC (λ(tjs',scc). u ∈ set tjs → set tjs = set tjs' ∪ scc ∧ u ∈ scc)
proof
  from tjs-union[of u tjs] show ?thesis
unfolding tj-stack-pop-def by (refine-vcg) auto
qed

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

definition tarjan-fin :: 'v ⇒ 'v tarjan-state ⇒ ('v,unit) tarjan-state-ext nres where
tarjan-fin v s = do {
  let ll = (if stack s = [] then lowlink s
               else let u = hd (stack s) in
                   (lowlink s)(u ↦→ min (ζ s u) (ζ s v)));
  let s' = s[ lowlink := ll ];
  ASSERT (v ∈ set (tj-stack s));
  ASSERT (distinct (tj-stack s));
  if ζ s v = δ s v then do {
    ASSERT (scc-root' E s v (scc-of E v));
    (tjs,scc) ← tj-stack-pop (tj-stack s) v;
    RETURN (state.more (s![ ] tj-stack := tjs, sccs := insert scc (sccs s))));
  } else do {
    ASSERT (∼ scc-root' E s v (scc-of E v));
    RETURN (state.more s')
  }
}

definition tarjan-back :: 'v ⇒ 'v ⇒ 'v tarjan-state ⇒ ('v,unit) tarjan-state-ext nres where
tarjan-back u v s = (if δ s v < δ s u ∧ v ∈ set (tj-stack s) then
  let ul' = min (ζ s u) (δ s v)
  in RETURN (state.more (s[ ] lowlink := (lowlink s)(v→ul') [ ]))
  else NOOP s)
end

definition tarjan-params :: 'v tarjan-param where
  tarjan-params = [on-init = RETURN (s[ ] sccs = {}, lowlink = Map.empty, tj-stack = [] )]
on-new-root = tarjan-disc,
on-discover = λu. tarjan-disc,
on-finish = tarjan-fin,
on-back-edge = tarjan-back,
on-cross-edge = tarjan-back,
is-break = λs. False ⊥

schematic-goal tarjan-params-simps[simp]:
on-init tarjan-params = ?OI
on-new-root tarjan-params = ?ONR
on-discover tarjan-params = ?OD
on-finish tarjan-params = ?OF
on-back-edge tarjan-params = ?OBE
on-cross-edge tarjan-params = ?OCE
is-break tarjan-params = ?IB
unfolding tarjan-params-def gen-parameterization.simps
by (rule refl)+

sublocale param-DFS-defs G tarjan-params .
end
locale Tarjan = Tarjan-def G +
param-DFS G tarjan-params
for G :: (’v, ’more) graph-rec-scheme
begin

lemma [simp]:
  sccs (empty-state (| sccs = s, lowlink = l, tj-stack = t|)) = s
  lowlink (empty-state (| sccs = s, lowlink = l, tj-stack = t|)) = l
  tj-stack (empty-state (| sccs = s, lowlink = l, tj-stack = t|)) = t
  by (simp-all add: empty-state-def)

lemma sccs-more-cong[cong]: state.more s = state.more s' ==> sccs s = sccs s'
  by (cases s, cases s') simp
lemma lowlink-more-cong[cong]: state.more s = state.more s' ==> lowlink s =
  lowlink s'
  by (cases s, cases s') simp
lemma tj-stack-more-cong[cong]: state.more s = state.more s' ==> tj-stack s =
  tj-stack s'
  by (cases s, cases s') simp

lemma [simp]:
  s⟦ state.more := (| sccs = sc, lowlink = l, tj-stack = t|)⟧
  = s⟧ sccs := sc, lowlink := l, tj-stack := t⟧
  by (cases s) simp
end
locale Tarjan-invar = Tarjan +
DFS-invar where param = tarjan-params
2.7.3 Correctness Proof

color context Tarjan-def begin
lemma Tarjan-invar-eq[simp]:
  DFS-invar G tarjan-params s \iff Tarjan-invar G s (is \ ?D \iff \ ?T)
proof
  assume \ ?D then interpret DFS-invar where param=\text{tarjan-params}.
  show \ ?T ..
next
  assume \ ?T then interpret Tarjan-invar .
  show \ ?D ..
qed
end

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

lemma i-tj-stack-distinct:
  is-invar (\ls. distinct \ (tj-stack s))
proof (induct rule: establish-invarI-ND)
case (new-discover s s' v)
  then interpret Tarjan-invar where s'=s by simp
from new-discover tj-stack-discovered have v \notin set (tj-stack s) by auto
with new-discover show \ ?case by (simp add: tarjan-disc-def)
next
case (finish s) thus \ ?case
  apply simp
  unfolding tarjan-fin-def tj-stack-pop-def
  apply (refine-vcg)
  apply (auto intro: distinct-tl)
  done
qed (simp-all add: tarjan-back-def)

lemmas (in Tarjan-invar) tj-stack-distinct =
i-tj-stack-distinct [THEN make-invar-thm]
context begin interpretation timing-syntax .

lemma i-tj-stack-incr-disc:
  is-invar (\( \lambda s. \forall k < \text{length} \ (tj-stack \ s). \forall j < k. \delta \ s \ (tj-stack \ s \ ! \ j) > \delta \ s \ (tj-stack \ s \ ! \ k) \))
  proof (induct rule: establish-invar1-ND)
    case (new-discover \( s \ s' \ v \)) then interpret Tarjan-invar where \( s=s \) by simp
    from new-discover tj-stack-discovered have \( v \notin \text{set} \ (tj-stack \ s) \) by auto
    moreover {
      fix \( k \ j \)
      assume \( k < \text{Suc} \ (\text{length} \ (tj-stack \ s)) \ j < k \)
      hence \( k - \text{Suc} \ 0 < \text{length} \ (tj-stack \ s) \) by simp
      hence \( tj-stack \ s \ ! \ (k - \text{Suc} \ 0) \in \text{set} \ (tj-stack \ s) \) using nth-mem by metis
      with tj-stack-discovered timing-less-counter have \( \delta \ s \ (tj-stack \ s \ ! \ (k - \text{Suc} \ 0)) < \text{counter} \ s \) by blast
    }
    moreover {
      fix \( k \ j \)
      define \( k' \) where \( k' = k - \text{Suc} \ 0 \)
      define \( j' \) where \( j' = j - \text{Suc} \ 0 \)
      assume A: \( k < \text{Suc} \ (\text{length} \ (tj-stack \ s)) \ j < k \ (v\#tj-stack \ s) \ ! \ j \neq v \)
      hence gt-0: \( j > 0 \land k > 0 \) by (cases \( j=0 \)) simp-all
      moreover with \( j < k \) have \( j' < k' \) by (simp add: j'-def k'-def)
      moreover from A have \( k' < \text{length} \ (tj-stack \ s) \) by (simp add: k'-def)
      ultimately have \( \delta \ s \ (tj-stack \ s \ ! \ j') > \delta \ s \ (tj-stack \ s \ ! \ k') \)
      using new-discover by blast
      with gt-0 have \( \delta \ s \ ((v\#tj-stack \ s) \ ! \ j) > \delta \ s \ (tj-stack \ s \ ! \ k') \)
      unfolding j'-def
      by (simp add: nth-Cons')
    }
    ultimately show ?case
    using new-discover
    by (auto simp add: tarjan-disc-def)
  next
  case (finish \( s \ s' \ u \))
    
    
    
    let \( ?dw = \text{dropWhile} \ ((\neq) \ u) \ (tj-stack \ s) \)
    let \( ?tw = \text{takeWhile} \ ((\neq) \ u) \ (tj-stack \ s) \)
    fix \( a \ k \ j \)
    assume A: \( a = \text{tl} \ ?dw \ k < \text{length} \ a \ j < k \)
    and \( u \in \text{set} \ (tj-stack \ s) \)
    hence \( ?dw \neq [] \) by auto
    define \( j' \ k' \) where \( j' = \text{Suc} \ j + \text{length} \ ?tw \) and \( k' = \text{Suc} \ k + \text{length} \ ?tw \)
    with \( j < k \) have \( j' < k' \) by simp

198
have \( \text{length} (\text{tj-stack } s) = \text{length } tw + \text{length } dw \)
by (simp add: length-append[symmetric])

moreover from \( A \) have ~∗: \( \text{Suc } k < \text{length } dw \) and ~∗∗: \( \text{Suc } j < \text{length } dw \)
by auto

ultimately have \( k' < \text{length} (\text{tj-stack } s) \)
by (simp add: \( k'-\text{def} \))

with finish \( j' < k' \)
have \( \delta s (\text{tj-stack } s ! k') < \delta s (\text{tj-stack } s ! j') \)
by simp

also from \( \text{dropWhile-nth}[OF \ ] \) have \( \text{tj-stack } s ! k' = ?dw ! \text{Suc } k \)
by (simp add: \( k'-\text{def} \))

also from \( \text{dropWhile-nth}[OF \ ] \) have \( \text{tj-stack } s ! j' = ?dw ! \text{Suc } j \)
by (simp add: \( j'-\text{def} \))

also from \( \text{nth-tl}[OF \ ] \) have \( ?dw ! \text{Suc } k = a ! k \)
by (simp add: \( A \))

also from \( \text{nth-tl}[OF \ ] \) have \( ?dw ! \text{Suc } j = a ! j \)
by (simp add: \( A \))

finally have \( \delta s (a ! k) < \delta s (a ! j) \).

} note aux = this

from \( \text{finish} \) show \( \text{?case} \)
apply simp
unfolding \( \text{tarjan-fin-def} \) \( \text{tj-stack-pop-def} \)
apply refine-vcg
apply (auto intro: aux)
done

qed (simp-all add: \( \text{tarjan-back-def} \))

end

end

context \( \text{Tarjan-invar} \) begin context begin interpretation \( \text{timing-syntax} \).

lemma \( \text{tj-stack-incr-disc} \):
assumes \( k < \text{length} (\text{tj-stack } s) \)
and \( j < k \)
shows \( \delta s (\text{tj-stack } s ! j) < \delta s (\text{tj-stack } s ! k) \)
using assms \( \text{i-tj-stack-incr-disc[THEN make-invar-thm]} \)
by blast

lemma \( \text{tjs-disc-dw-tw} \):
fixes \( u \)
defines \( dw \equiv \text{dropWhile } ((\neq) u) (\text{tj-stack } s) \)
defines \( tw \equiv \text{takeWhile } ((\neq) u) (\text{tj-stack } s) \)
assumes \( x \in \text{set} \) \( dw \) \( y \in \text{set} \) \( tw \)
shows \( \delta s x < \delta s y \)
proof ~
from assms obtain \( k \)
where \( k: dw ! k = x \) \( k < \text{length} \) \( dw \) by (metis in-set-conv-nth)
from assms obtain \( j \)
where \( j: tw ! j = y \) \( j < \text{length} \) \( tw \) by (metis in-set-conv-nth)

have \( \text{length} (\text{tj-stack } s) = \text{length} tw + \text{length} dw \)
by (simp add: length-append[symmetric] \( tw\)-def \( dw\)-def)

with \( k \) \( j \)
have \( \delta s (\text{tj-stack } s ! (k + \text{length} tw)) < \delta s (\text{tj-stack } s ! j) \)
by (simp add: \( \text{tj-stack-incr-disc} \))
also from $j$ takeWhile-nth have tj-stack $s! j = y$ by (metis tw-def)
also from dropWhile-nth $k$ have tj-stack $s! (k + \text{length } tw) = x$ by (metis tw-def dw-def)

finally show ?thesis.

qed

end end

class Tarjan begin context begin interpretation timing-syntax.

lemma i-sccs-finished-stack-ss-tj-stack: is-invar $(\lambda s. \bigcup sccs s \subseteq \text{dom } (\text{finished } s)) \land \text{set } (\text{stack } s) \subseteq \text{set } (\text{tj-stack } s))$

proof (induct rule: establish-invarI)

case (finish $s$ $s'$ $u$) then interpret Tarjan-invar where $s = s$ by simp

let $?tw = \text{takeWhile } (\neq u) (\text{tj-stack } s)$
let $?dw = \text{dropWhile } (\neq u) (\text{tj-stack } s)$

{ 
fix $x$
assume $A: x \neq u \in \text{set } ?tw$ $u \in \text{set } (\text{tj-stack } s)$

proof (rule ccontr)

assume $x \notin \text{dom } (\text{finished } s)$

with $x$-tj tj-stack-discovered discovered-eq-finished-un-stack have $x \in \text{set } (\text{tj-stack } s)$ by (auto dest: set-takeWhileD)

have $x \in \text{dom } (\text{finished } s)$

proof (rule simp)

assume $x \notin \text{dom } (\text{finished } s)$

with $x$-tj tj-stack-discovered discovered-eq-finished-un-stack have $x \in \text{set } (\text{tj-stack } s)$ by (auto dest: set-takeWhileD)

have $x \in \text{set } (\text{tj-stack } s)$ by blast

with $x$-tj tj-stack-discovered discovered-eq-finished-un-stack have $x \in \text{set } (\text{tj-stack } s)$ by (auto dest: set-takeWhileD)

from $A$ have $?dw \neq []$ by simp

with $\text{hd-dropWhile } (\text{OF } this)$ $\text{hd-in-set } u \in \text{set } ?dw$ by metis

with $\text{tjs-disc-dw-tw } (x \in \text{set } ?tw)$ have $\delta s u < \delta s x$ by simp

with $*$ show False by force

qed

hence $\exists y. \text{finished } s x = \text{Some } y$ by blast

} note aux-scc = this

{ 
fix $x$

assume $A: x \in \text{set } (\text{tl } (\text{stack } s))$ $u \in \text{set } (\text{tj-stack } s)$

with finish stack-distinct have $x \neq u$ by (cases stack s) auto

moreover

from $A$ have $x \in \text{set } (\text{stack } s)$ by (metis in-set-tlD)

with $\text{stack-not-finished } x \notin \text{dom } (\text{finished } s)$ by simp

with $A$ aux-scc (OF $x \neq w$) have $x \notin \text{set } ?tw$ by blast

moreover
from finish \( x \in \text{set} (\text{stack } s) \) have \( x \in \text{set} (\text{tj-stack } s) \) by auto

moreover note \( \text{tjs-union[OF } u \in \text{set} (\text{tj-stack } s)\] \)

ultimately have \( x \in \text{set} (\text{tl } ?d\)w) by blast
} note aux-tj = this

from finish show \( ?\text{case} \)
  apply simp
  unfolding tarjan-fin-def tj-stack-pop-def
  apply (refine-vcg)
  using aux-scc aux-tj apply (auto dest: in-set-tlD)
  done
qed (auto simp add: tarjan-disc-def tarjan-back-def)

lemma i-tj-stack-ss-stack-finished:
  is-invar \( \lambda s. \text{set} (\text{tj-stack } s) \subseteq \text{set} (\text{stack } s) \cup \text{dom} (\text{finished } s) \)
proof (induct rule: establish-invarI)
  case (finish s) thus \( ?\text{case} \)
    apply simp
    unfolding tarjan-fin-def
    apply (refine-vcg tj-stack-pop-set-leaf-rule)
    apply ((simp, cases stack s, simp-all))
    done
qed (auto simp add: tarjan-disc-def tarjan-back-def)

lemma i-finished-ss-sccs-tj-stack:
  is-invar \( \lambda s. \text{dom} (\text{finished } s) \subseteq \bigcup \text{sccs } s \cup \text{set} (\text{tj-stack } s) \)
proof (induction rule: establish-invarI-ND)
  case (new-discover s s' v)
    then interpret Tarjan-invar where \( s=s \) by simp
    from new-discover finished-discovered have \( v \notin \text{dom} (\text{finished } s) \) by auto
    with new-discover show \( ?\text{case} \)
      by (auto simp add: tarjan-disc-def)
  next
  case (finish s s' u)
    then interpret Tarjan-invar where \( s=s \) by simp
    from finish show \( ?\text{case} \)
      apply simp
      unfolding tarjan-fin-def
      apply (refine-vcg tj-stack-pop-set-leaf-rule)
      apply auto
      done
qed (simp-all add: tarjan-back-def)
end end

context Tarjan-invar begin
lemmas finished-ss-sccs-tj-stack =
  i-finished-ss-sccs-tj-stack[THEN make-invar-thm]
lemmas tj-stack-ss-stack-finished =
lemma sccs-finished:
\[ \bigcup \text{sccs } s \subseteq \text{dom } (\text{finished } s) \]
using i-sccs-finished-stack-ss-tj-stack[THEN make-invar-thm]
by blast

lemma stack-ss-tj-stack:
set (stack s) \subseteq set (tj-stack s)
using i-sccs-finished-stack-ss-tj-stack[THEN make-invar-thm]
by blast

lemma hd-stack-in-tj-stack:
stack s \neq [] \implies hd (stack s) \in set (tj-stack s)
using stack-ss-tj-stack hd-in-set
by auto
end

default

context Tarjan begin context begin interpretation timing-syntax .

lemma i-no-finished-root:
\( \lambda s. \text{sec-root } s \ r \ \text{scc} \land r \in \text{dom } (\text{finished } s) \implies (\forall x \in \text{scc} . \ x \notin \text{set } (tj-stack s)) \)
proof (induct rule: establish-invarI-ND-CB)
case (new-discover s s' v) then interpret Tarjan-invar where s=s by simp
  { fix x
    let ?s = s'
    assume TRANS: \( \forall \Psi . \text{tarjan-disc } v \ s' \leq_n \text{SPEC } \Psi \implies \Psi \ x \)
    and inv': DFS-invar G tarjan-params \( (s'(\text{state.more } := x)) \)
    and r: sec-root ?s r \text{scc} r \in \text{dom } (\text{finished } s')
    from inv' interpret s': Tarjan-invar where s=?s by simp
    have tj-stack ?s = v#tj-stack s
      by (rule TRANS) (simp add: new-discover tarjan-disc-def)
    moreover from r s'.sec-root-finished-impl-scc-finished have sec \subseteq \text{dom } (\text{finished } ?s)
      by auto
      with new-discover finished-discovered have v \notin sec by force
    moreover
      from r finished-discovered new-discover have r \in \text{dom } (\text{discovered } s)
      by auto
      with r inv' new-discover have sec-root s r \text{scc}
      apply (intro sec-root-transfer[where s'=?s, THEN iffD2])
      apply clarsimp
      done
      with new-discover r have \( \forall x \in \text{scc} . \ x \notin \text{set } (tj-stack s') \)
      by simp

202
ultimately have \(\forall x \in \text{scc}.\ x \notin \text{set}\ (\text{tj-stack } ?s)\) by (auto simp: new-discover)

with new-discover show ?case by (simp add: pw-leof-iff)

next
case (cross-back-edge s s' u v) then interpret Tarjan-invar where \(s=s\) by simp

\{
fix \(x\)
let \(?s = s'[\text{state.more := } x]\)
assume TRANS: \(\forall \Psi.\ \text{tarjan-back } u v s' \leq_n \text{SPEC } \Psi \Rightarrow \Psi x\)
and \(r: \text{scc-root } ?s\ r\ \text{sec } r \in \text{dom}\ (\text{finished } s')\)
with cross-back-edge have scc-root \(s\ r\ \text{scc}\)
by (simp add: scc-root-transfer? where \(s'=?s\))
\}

moreover
have tj-stack \(?s = \text{tj-stack } s\) by (rule TRANS) (simp add: cross-back-edge)

tarjan-back-def)

ultimately have \(\forall x \in \text{scc}.\ x \notin \text{set}\ (\text{tj-stack } ?s)\)

using cross-back-edge \(r\) by simp

\}

with cross-back-edge show ?case by (simp add: pw-leof-iff)

next
case (finish s s' u) then interpret Tarjan-invar where \(s=s\) by simp

\{
fix \(x\)
let \(?s = s'[\text{state.more := } x]\)
assume TRANS: \(\forall \Psi.\ \text{tarjan-fin } u\ s' \leq_n \text{SPEC } \Psi \Rightarrow \Psi x\)
and inv': DFS-invar \(G\) tarjan-params \((s'[\text{state.more := } x])\)
and \(r: \text{scc-root } ?s\ r\ \text{sec } r \in \text{dom}\ (\text{finished } s')\)

from inv' interpret s': Tarjan-invar where \(s'=?s\) by simp

have \(\forall x \in \text{scc}.\ x \notin \text{set}\ (\text{tj-stack } ?s)\)
proof (cases \(r = u\))
case False with finish \(r\) have \(\forall x \in \text{scc}.\ x \notin \text{set}\ (\text{tj-stack } s)\)
using sec-root-transfer? where \(s'=?s\)
by simp

moreover have set \((\text{tj-stack } ?s) \subseteq \text{set}\ (\text{tj-stack } s)\)
apply (rule TRANS)
unfolding tarjan-fin-def
apply (refine-vcg tj-stack-pop-set-leof-rule)
apply (simp-all add: finish)
done
ultimately show ?thesis by blast
next
case True with \(r\ s'.\ \text{scc-root-unique-is-scc}\) have scc-root \(?s\ u\) (scc-of \(E\ u))
by simp
  with s’.scc-root-transfer[where s’=s’] finish have scc-root s’ u (scc-of E u) by simp

moreover
hence [simp]: tj-stack ?s = tl (dropWhile ((≠) u) (tj-stack s))
  apply (rule-tac TRANS)
unfolding tarjan-fin-def tj-stack-pop-def
  apply (refine-vcg)
apply (simp-all add: finish)
done

{  
let ?dw = dropWhile ((≠) u) (tj-stack s)
let ?tw = takeWhile ((≠) u) (tj-stack s)
fix x
define j :: nat where j = 0

assume x: x ∈ set (tj-stack ?s)
then obtain i where: i < length (tj-stack ?s) tj-stack ?s ! i = x
  by (metis in-set-conv-nth)

have length (tj-stack s) = length ?tw + length ?dw
  by (simp add: length-append[ symmetric])

with i have δ s (tj-stack s ! (Suc i + length ?tw)) < δ s (tj-stack s ! length ?tw)
  by (simp add: tj-stack-incr-disc)

also from hd-stack-in-tj-stack finish have: ?dw ≠ [] and length ?dw > 0 by simp-all
  from hd-dropWhile[OF ne] hd-conv-nth[OF ne] have ?dw ! 0 = u by simp
  with dropWhile-nth[OF length ?dw > 0] have tj-stack s ! length ?tw = u by simp

also from i have ?dw ! Suc i = x Suc i < length ?dw by (simp-all add: nth-tl[OF ne])
  with dropWhile-nth[OF this(2)] have tj-stack s ! (Suc i + length ?tw) = x by simp

finally have δ s x < δ ?s u by (simp add: finish)

moreover from x s’.tj-stack-discovered have x ∈ dom (discovered ?s) by auto
ultimately have x ∉ scc using s’.scc-root-disc-le r True by force
} thus ?thesis by metis
qed

with finish show ?case by (simp add: pw-leaf-iff)
context Tarjan-invar begin

lemma no-finished-root: assumes scc-root s r scc and r ∈ dom (finished s) and x ∈ scc shows x /∈ set (tj-stack s) using assms using i-no-finished-root[THEN make-invar-thm] by blast

context begin interpretation timing-syntax .

lemma tj-stack-reach-stack: assumes u ∈ set (tj-stack s) shows ∃ v ∈ set (stack s). (u,v) ∈ E* ∧ δ s v ≤ δ s u proof −
  have u-scc: u ∈ scc-of E u by simp
  from assms tj-stack-discovered have u-disc: u ∈ dom (discovered s) by auto with scc-root-of-node-exists obtain r where r: scc-root s r (scc-of E u) by blast
  have r ∈ set (stack s) proof (rule ccontr)
    assume r /∈ set (stack s)
    with r[unfolded scc-root-def] stack-set-def have r ∈ dom (finished s) by simp
    with u-scc have u /∈ set (tj-stack s) using no-finished-root r by blast
    with assms show False by contradiction
  qed
  moreover from r scc-reach-scc-root u-scc u-disc have (u,r) ∈ E* by blast
  moreover from r scc-root-disc-le u-scc u-disc have δ s r ≤ δ s u by blast
  ultimately show ?thesis by metis
  qed

lemma tj-stack-reach-hd-stack: assumes v ∈ set (tj-stack s) shows (v, hd (stack s)) ∈ E* proof −
  from tj-stack-reach-stack assms obtain r where r: r ∈ set (stack s) (v,r) ∈ E* by blast
  hence r = hd (stack s) ∨ r ∈ set (tl (stack s)) by (cases stack s) auto
  thus ?thesis proof
    assume r = hd (stack s) with r show ?thesis by simp
  next
    from r have ne :stack s ≠ [] by auto
  qed
assume \( r \in \text{set} \ (\text{tl} \ (\text{stack} \ s)) \)

with \( \text{tl-stack-hd-tree-path} \ne \) have \((r, \text{hd} \ (\text{stack} \ s)) \in (\text{tree-edges} \ s)^+ \) by simp

with \( \text{trancl-mono-mp} \ \text{tree-edges-ssE} \) have \((r, \text{hd} \ (\text{stack} \ s)) \in E^+ \) by (metis \( \text{rtrancl-eq-or-trancl} \))

with \langle \( v, r \rangle \in E^* \rangle \) show \(?\text{thesis}\) by (metis \( \text{rtrancl-trans} \))

qed

lemma empty-stack-imp-empty-tj-stack:
assumes \( \text{stack} \ s = [] \)
shows \( \text{tj-stack} \ s = [] \)
proof (rule ccontr)
assume \( \neg \) then obtain \( x \) where \( x \in \text{set} \ (\text{tj-stack} \ s) \) by auto
with \( \text{tj-stack-reach-stack} \) obtain \( r \) where \( r \in \text{set} \ (\text{stack} \ s) \) by auto
with \( \text{assms} \) show \text{False} by simp
qed

lemma stacks-eq-iff: \( \text{stack} \ s = [] \iff \text{tj-stack} \ s = [] \)
using empty-stack-imp-empty-tj-stack stack-ss-tj-stack by auto

end end

context Tarjan begin context begin interpretation timing-syntax .

lemma i-sccs-are-sccs:
is-invar \( (\lambda s. \forall scc \in \text{sccs} \ s. \text{is-scc} \ E \ scc) \)
proof (induction rule: establish-invarI)
case (finish \( s s' u \)) then interpret Tarjan-invar where \( s=s' \) by simp
from \( \text{finish} \) have \( \text{EQ[simp]}: \)
\( \text{finished} \ s' = (\text{finished} \ s)(u \mapsto \text{counter} \ s) \)
\( \text{discovered} \ s' = \text{discovered} \ s \)
\( \text{tree-edges} \ s' = \text{tree-edges} \ s \)
\( \text{sccs} \ s' = \text{sccs} \ s \)
\( \text{tj-stack} \ s' = \text{tj-stack} \ s \)
by simp-all

fix \( x \)

let \( s = s'[(\text{state}.more := x)] \)
assume \( \text{TRANS}: \\land \Psi. \ \text{tarjan-fin} \ u \ s' \subseteq_n \ \text{SPEC} \ \Psi \Longrightarrow \Psi \ x \)
and \( \text{inv'}: \ \text{DFS-invar} \ G \ \text{tarjan-params} \ (s'[(\text{state}.more := x)]) \)
then interpret \( s': \ \text{Tarjan-invar} \ where \ s=s' \) by simp

from \( \text{finish} \) have \( \text{hd-in-set stack-set-def} \)
\( u\text{-disc}: u \in \text{dom} \ (\text{discovered} \ s) \)
and \( u\text{-n-fin}: u \notin \text{dom} \ (\text{finished} \ s) \) by blast+

have \( \forall scc \in \text{sccs} \ s. \ \text{is-scc} \ E \ scc \)

end
proof (cases scc-root \( s' \) \( u \) (scc-of \( E \) \( u \)))

  case False
  have \( \text{sccs} \ ?s = \text{sccs} s \)
    apply (rule TRANS)
    unfolding tarjan-fin-def tj-stack-pop-def
    by (refine-vcg) (simp-all add: False)
  thus \(?thesis\) by (simp add: finish)

next
  case True
  let \(?dw = \text{dropWhile} \ ((\neq) \ u) \ (\text{tj-stack} \ s)\)
  let \(?tw = \text{takeWhile} \ ((\neq) \ u) \ (\text{tj-stack} \ s)\)
  let \(?tw' = \text{insert} \ u \ (\text{set} \ ?tw)\)

  have \[\text{simp}]: \( \text{sccs} \ ?s = \text{insert} \ ?tw' \ (\text{sccs} s)\)
    apply (rule TRANS)
    unfolding tarjan-fin-def tj-stack-pop-def
    by (refine-vcg) (simp-all add: True)

  have \[\text{simp}]: \( \text{tj-stack} \ ?s = \text{tl} \ ?dw\)
    apply (rule TRANS)
    unfolding tarjan-fin-def tj-stack-pop-def
    by (refine-vcg) (simp-all add: True)

  from True scc-root-transfer[\text{where} \( s'=\text{s} \)]
  have \( \text{scc-root} \ s \ u \ (\text{scc-of} \ E \ u)\)
    by simp

  with inv' scc-root-transfer[\text{where} \( s'=\text{s} \)]
  u-disc have u-root: \( \text{scc-root} \ ?s \ u \ (\text{scc-of} \ E \ u)\)
    by simp

  have \(?tw' \subseteq \text{scc-of} \ E \ u\)
  proof
    fix \( v \) assume \( v \in \ ?tw' \)
    show \( v \in \ \text{scc-of} \ E \ u\)
    proof cases
      assume \( v \neq u \) with \( v \) have \( v \in \ \text{set} \ ?tw \) by auto
      hence v-tj: \( v \in \ \text{set} \ (\text{tj-stack} \ s)\) by (auto dest: set-takeWhileD)
      with tj-stack-discovered have v-disc: \( v \in \ \text{dom} \ (\text{discovered} \ s)\) by auto
      from hd-stack-in-tj-stack finish have \( ?dw \neq [] \) by simp
      with hd-dropWhile[\text{OF this}] hd-in-set have \( u \in \ \text{set} \ ?dw \) by metis
      with \( v \) have \( \delta \ s \ v > \delta \ s \ u \) using tjs-disc-dw-tw by blast
    moreover have \( v \in \ \text{dom} \ (\text{finished} \ s)\)
    proof (rule ccontr)
      assume \( v \notin \ \text{dom} \ (\text{finished} \ s)\)
      with v-disc stack-set-def have \( v \in \ \text{set} \ (\text{stack} \ s)\) by auto
      with \( v \neq u \) finish have \( v \in \ \text{set} \ (\text{tl} \ (\text{stack} \ s))\) by (cases stack \( s\)) auto
      with tl-ll-stack-hd-discover finish have \( \delta \ s \ v < \delta \ s \ u \) by simp
      with \( \delta \ s \ v > \delta \ s \ u \) show False by force
    qed

  qed
ultimately have \((u,v) \in (\text{tree-edges } s)^+\)
   using parenthesis-impl-tree-path-not-finished[\text{OF } u\text{-disc}] u-n-fin
   by force
with trancl-mono-mp tree-edges-ssE have \((u,v) \in E^*\) by (metis rtrancl-eq-or-trancl)
moreover
from tj-stack-reach-hd-stack v-tj finish have \((v,u) \in E^*\) by simp
moreover have is-scc E (scc-of E u) u \in scc-of E u by simp-all
ultimately show ?thesis using is-scc-closed by metis
qed simp
qed
moreover have scc-of E u \subseteq ?tw'
proof
fix v assume v: v \in scc-of E u
moreover note u-root
moreover have u \in dom (finished ?s) by simp
ultimately have v \in dom (finished ?s) v \notin (tj-stack ?s)
   using s'.scc-root-finished-impl-scc-finished s'.no-finished-root
by auto
with s'.finished-ss-sccs-tj-stack have v \in \bigcup sccs ?s by blast
hence v \in \bigcup sccs s \lor v \in ?tw' by auto
thus v \in ?tw'
proof
assume v \in \bigcup sccs s
then obtain scc where scc: v \in scc scc \in sccs s by auto
moreover with finish have is-scc E scc by simp
moreover have is-scc E (scc-of E u) by simp
moreover note v
ultimately have scc = scc-of E u using is-scc-unique by metis
hence u \in scc by simp
with scc sccs-finished have u \in dom (finished s) by auto
with u-n-fin show ?thesis by contradiction
qed simp
qed
ultimately have ?tw' = scc-of E u by auto
hence is-scc E ?tw' by simp
with finish show ?thesis by auto
qed
}
thus ?case by (auto simp: pw-leof-iff finish)
qed (simp-all: tarjan-back-def tarjan-disc-def)
end

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

context begin interpretation timing-syntax .
lemma i-lowlink-eq-LowLink:
   is-invar (λs. ∀x ∈ dom (discovered s), ζ s x = LowLink s x)

proof -
{
  fix s s′ :: ′v tarjan-state
  fix v w
  fix x

  let ?s = s′[|state.more := x|]

  assume pre-ll-sub-rev: ∀w. [Tarjan-invar G ?s; w ∈ dom (discovered ?s); w ≠ v] ⇒ lowlink-set ?s w ⊆ lowlink-set s w ∪ {v}
  assume tree-sub : tree-edges s′ = tree-edges s ∪ (∃u. u ≠ v ∧ tree-edges s′ = tree-edges s ∪ {(u,v)})

  assume Tarjan-invar G s
  assume [simp]: discovered s′ = (discovered s)(v → counter s)
  finished s′ = finished s
  lowlink s′ = lowlink s
  cross-edges s′ = cross-edges s back-edges s′ = back-edges s

  assume v-n-disc: v /∈ dom (discovered s)
  assume IH: ∀w. w ∈ dom (discovered s) ⇒ ζ s w = LowLink s w

  assume TRANS: ∀Ψ. tarjan-disc v s′ ≤n SPEC Ψ ⇒ Ψ x
  and INV: DFS-invar G tarjan-params ?s
  and w-disc: w ∈ dom (discovered ?s)

  interpret Tarjan-invar where s=s by fact
  from INV interpret s′:Tarjan-invar where s=?s by simp

  have [simp]: lowlink ?s = (lowlink s)(v → counter s)
    by (rule TRANS) (auto simp: tarjan-disc-def)

  from v-n-disc edge-imp-discovered have edges s “ v = {} by auto
  with tree-sub tree-edge-imp-discovered have edges ?s “ v = {} by auto
  with s′.no-path-imp-no-lowlink-path have \w. ¬(∃p. lowlink-path ?s v p w)

  by metis

  hence ll-v: lowlink-set ?s v = {v}

  unfolding lowlink-set-def by auto

  have ζ ?s w = LowLink ?s w
  proof (cases w=v)
    case True with ll-v show ?thesis by simp
  next
    case False hence ζ ?s w = ζ s w by simp

  also from IH have ζ s w = LowLink s w using w-disc False by simp
  also have LowLink s w = LowLink ?s w

  proof (rule LowLink-eqI[OF INV])
from v-n-disc show discovered s ⊆ \text{discovered } ?s by (simp add: \text{map-le-def})

from tree-sub show lowlink-set s w ⊆ lowlink-set ?s w
  unfolding lowlink-set-def lowlink-path-def
  by auto

show lowlink-set ?s w ⊆ lowlink-set s w ∪ \{v\}
proof (cases w = v)
  case True with ll-v show \?thesis by auto
next
  case False thus \?thesis
    using pre-ll-sub-rev w-disc INV
    by simp
qed

show w ∈ dom (discovered s) using w-disc False by simp

fix ll assume ll ∈ \{v\} with timing-less-counter lowlink-set-discovered
have \(\forall x. x \in \delta \ s'\text{lowlink-set } s w \implies x < \delta \ ?s ll\) by simp force
moreover from Min-in lowlink-set-finite lowlink-set-not-empty w-disc
False have LowLink s w ∈ \delta \ s'\text{lowlink-set } s w by auto
ultimately show LowLink s w ≤ \delta \ ?s ll by force
qed
finally show ?thesis.
qed
} note tarjan-disc-aux = this

show ?thesis
proof (induct rule: establish-invarI-CB)
  case (new-root s s' v0)
  { fix w x
    let ?s = new-root v0 s[state.more := x] |
    have lowlink-set ?s w ⊆ lowlink-set s w ∪ \{v0\}
      unfolding lowlink-set-def lowlink-path-def
      by auto
  } note * = this

from new-root show ?case using tarjan-disc-aux[OF *]
  by (auto simp add: pw-leaf-iff)
next
  case (discover s s' u v) then interpret Tarjan-invar where s=s by simp
    let \?s' = discover (hd (stack s)) u (s[pending := pending s - \{(hd (stack s),v)\}])
  { 210
fix \( w \) \( x \)

let \( ?s = ?s' [\text{state.more := } x] \)

assume \( \text{INV: Tarjan-invar } G ?s \)

and \( d : w \in \text{dom} \{\text{discovered } ?s'\} \)

and \( w \neq v \)

interpret \( ?s' \): \( \text{Tarjan-invar } \) where \( s = ?s \) by \qquad \text{fact}

have \( \text{lowlink-set } ?s \ w \subseteq \text{lowlink-set } s \ w \cup \{v\} \)

proof

fix \( ll \)

assume \( ll : ll \in \text{lowlink-set } ?s \ w \)

hence \( ll = w \lor (\exists p. \text{lowlink-path } ?s \ w p ll) \) by (auto simp add: lowlink-set-def)

thus \( ll \in \text{lowlink-set } s \ w \cup \{v\} \) (is \( ll \in ?L \))

proof

assume \( ll = w \) with \( d \) show \( \text{thesis by} \) (auto simp add: lowlink-set-def)

next

assume \( \exists p. \text{lowlink-path } ?s \ w p ll \)

then guess \( p \) .. note \( p = this \)

hence [simp]: \( p \neq [] \) by (simp add: lowlink-path-def)

from \( p \) have \( \text{hd } p = w \) by (auto simp add: lowlink-path-def path-hd)

show \( \text{thesis} \)

proof (rule tri-caseE)

assume \( v \neq ll \ v \notin \text{set } p \) hence \( \text{lowlink-path } s \ w \ p \ ll \)

using \( p \) by (auto simp add: lowlink-path-def)

with \( ll \) show \( \text{thesis by} \) (auto simp add: lowlink-set-def)

next

assume \( v = ll \) thus \( \text{thesis by} \) simp

next

assume \( v \in \text{set } p \) \( v \neq ll \)

then obtain \( i \) where \( i : i < \text{length } p \ p!i = v \)

by (metis in-set-conv-nth)

have False

proof (cases \( i \))

case 0 with \( i \) have \( \text{hd } p = v \) by (simp add: hd-conv-nth)

with \( \text{hd } p = w \) \( (w \neq v) \) show False by simp

next

case (Suc \( n \)) with \( i \) \( s' \).lowlink-path-finished[\( OF p, \text{where } j=i \)] have \( v \in \text{dom} \) (finished ?s) by simp

with \( \text{finished-discovered } \text{discover} \) show False by auto

qed

thus \( \text{thesis ..} \)

qed

qed

qed

211
\[
\text{from discover hd-in-set stack-set-def have } v \neq u \text{ by auto}
\]
\[
\text{with discover have } *: \text{tree-edges } ?s' = \text{tree-edges } s \lor (\exists u. \ u \neq v \land \text{tree-edges } ?s' = \text{tree-edges } s \cup \{(u,v)\}) \text{ by auto}
\]
\[
\text{from discover show } \text{?case using tarjan-disc-aux[OF } * *]\]
proof (rule LowLink-eqI[OF inv' - ll-sub ll-sub-rev w-disc])
  show discovered s \subseteq_m discovered ?s by simp

fix ll assume ll \in \{v\} ll \in lowlink-set ?s w
with \* show LowLink s w \leq \delta ?s ll by simp
qed

} note LL-eqI = this

have \(\zeta ?s w = \text{LowLink} ?s w\)
proof (cases w=\u)
  case True show \(?thesis
proof (cases (\delta s v < \delta s w \land v \in \text{set} (\text{tj-stack s}) \land \delta s v < \zeta s w))
  case False
  from v have \(v \in \text{stack s}\) by (auto simp add: stack-set-def)
  with finished-ss-sccs-tj-stack v-n-tj have \(v \notin \text{set} (\text{stack s})\) by simp
  unfolding lowlink-set-def by (auto simp add: stack-set-def)

from v-n-tj have \(v \notin \text{set} (\text{stack s})\) using stack-ss-tj-stack by auto
with cross-back-edge have \(v \in \text{dom} (\text{finished s})\) by (auto simp add: stack-set-def)
with \(\text{finished-ss-sccs-tj-stack v-n-tj sccs-are-sccs}\) obtain scc
where scc: \(v \in \text{scc}\ \text{scc} \in \text{sccs s is-scc E scc}\) by blast
with is-scc-closed e have \(u \in \text{scc}\) by metis
with scc sccs-finished u-n-fin have False by blast
thus \(?thesis ..
qed
qed

finally show \(?thesis ..
next
  case True note all-True = this
  with \(w=\u\) have \(\zeta ?s w = \delta s v\)
  by (rule-tac TRANS) (simp add: tarjan-back-def cross-back-edge)

also from True cross-back-edge w-disc have \(\delta s v < \text{LowLink} s w\) by simp
with lowlink-set-finite lowlink-set-not-empty w-disc have \(\delta s v = \text{Min} (\text{\?L} \cup \{\delta s v\})\) by simp
also have \(v \in \text{lowlink-set} ?s w\)
proof –

213
have \( cb: (u,v) \in \text{cross-edges} \ ?s \cup \text{back-edges} \ ?s \) by (simp add: cross-back-edge)

with \( s'.\text{lowlink-path-single} \) have \( \text{lowlink-path} \ ?s \ u \ [u] \ v \) by auto
moreover from \( cb \ s'.\text{cross-edges-ssE} \ s'.\text{back-edges-ssE} \) have \( (u,v) \in E \) by blast

hence \((u,v) \in E^* \) ..
moreover from \( \text{all-True} \ tj\text{-stack-reach-hd-stack} \) have \((v,u) \in E^* \) by (simp add: cross-back-edge)
moreover note \( \langle v \in \text{dom} \ (\text{discovered} \ s) \rangle \)
ultimately show ?thesis by (auto intro: \( s'.\text{lowlink-setI} \) simp: \( \langle w=u \rangle \))

qed

next


case False — \( w \neq u \)
hence \( \zeta \ ?s \ w = \zeta \ s \ w \) by (rule-tac TRANS) (simp add: \text{tarjan-back-def} cross-back-edge)
also have \( \zeta \ s \ w = \text{LowLink} \ s \ w \) using \( \text{w-disc} \ False \) by (simp add: cross-back-edge)
also have \( \text{LowLink} \ s \ w = \text{LowLink} \ ?s \ w \)
proof (rule LL-eqI)
assume \( v: v \in \text{lowlink-set} \ ?s \)
thus \( \text{LowLink} \ s \ w \leq \delta \ ?s \ v \) using \( LLw \)
proof cases
assume \( v \neq w \)
with \( v \) obtain \( p \) where \( p: \text{lowlink-path} \ ?s \ w \ p \)
by (auto simp: \text{lowlink-set-def} \text{lowlink-path-def})
hence \( \text{hd} \ p = w \) by (auto simp: \text{lowlink-path-def} \text{path-hd})

show ?thesis
proof (cases \( u \in \text{set} \ p \))
case False with \( \text{last-in-set} \ p \) \text{cross-back-edge} have \( \text{last} \ p \neq \text{hd} \ (\text{stack} \ s) \) by force
with \( p \) have \( \text{lowlink-path} \ s \ w \ p \ v \)
by (auto simp: \text{cross-back-edge} \text{lowlink-path-def})
with \( v \) have \( v \in \text{lowlink-set} \ s \ w \)
by (auto intro: \text{lowlink-setI} simp: \text{lowlink-set-def} \text{cross-back-edge})
thus ?thesis by simp

next

case True then obtain \( i \) where \( i: i < \text{length} \ p \)
by (metis \text{in-set-conv-nth})
have \( \text{False} \)
proof (cases \( i \))
case 0 with \( i \) have \( \text{hd} \ p = u \) by (simp add: \text{hd-conv-nth})
with \( \langle \text{hd} \ p = w \rangle \) \( w \neq w \) show \( \text{False} \) by simp

next

214
case (Suc n) with i s'.lowlink-path-finished[OF p(I), where j=i]

have
  u ∈ dom (finished ?s) by simp
  with u-n-fin show ?thesis by simp
  thus ?thesis ..
  qed
  qed
  qed
  finally show ?thesis .
  qed

} note aux = this

with cross-back-edge show ?case by (auto simp: pw-leof-iff)

next

case (finish s s' u) then interpret Tarjan-invar where s=s by simp

from finish have [simp]:
  discovered s' = discovered s
  finished s' = (finished s)(u→counter s)
  tree-edges s' = tree-edges s
  back-edges s' = back-edges s
  cross-edges s' = cross-edges s
  lowlink s' = lowlink s tj-stack s' = tj-stack s

by simp-all

from finish hd-in-set stack-discovered have u-disc: u ∈ dom (discovered s)
by blast

{
  fix w :: 'v
  fix x

  let ?s = s'[(state.more := x)]
  let ?L = δ s' (lowlink-set s w
  let ?Lu = δ s (lowlink-set s u
  let ?L' = δ s (lowlink-set ?s w

  assume TRANS: \( \forall \Psi. \text{tarjan-fin} u s' \leq_n \text{SPEC} \Rightarrow \Psi x \)
  and inv': DFS-invar G tarjan-params ?s
  and w-disc: w ∈ dom (discovered ?s)

  from inv' interpret s':Tarjan-invar where s=?s by simp

  have ll-sub: lowlink-set s w ⊆ lowlink-set ?s w
    unfolding lowlink-set-def lowlink-path-def
    by auto

  have ll-sub-rev: lowlink-set ?s w ⊆ lowlink-set s w ∪ lowlink-set s u
  proof
fix \( ll \)
assume \( ll \): \( ll \in \text{lowlink-set} \ ?s \ w \)
  \( \quad \text{hence } ll = w \lor (\exists p. \ \text{lowlink-path} \ ?s \ w \ p \ ll) \) by (auto simp add: lowlink-set-def)
thus \( ll \in \text{lowlink-set} \ s \ w \cup \text{lowlink-set} \ s \ u \)
proof (rule disjE1)
  assume \( ll = w \) with \( w\text{-disc} \)
  show \( ?\text{thesis} \) by (auto simp add: lowlink-set-def)
next
  assume \( ll \neq w \)
  assume \( \exists p. \ \text{lowlink-path} \ ?s \ w \ p \ ll \)
  then guess \( p \) .. note \( p = \text{this} \)
  hence \( [\text{simp}]: p \neq [] \) by (simp add: lowlink-path-def)
from \( p \) have \( \text{hd} \ p = w \) by (auto simp add: lowlink-path-def path-hd)
show \( ?\text{thesis} \)
proof (cases \( u \in \text{set} \ p \))
  case False
  hence \( \text{lowlink-path} \ s \ w \ p \ ll \) using \( p \) by (auto simp add: lowlink-path-def)
  with \( ll \) show \( ?\text{thesis} \) by (auto simp add: lowlink-set-def)
next
  case True
  then obtain \( i \) where \( i: i < \text{length} \ p \ p!i = u \)
    by (metis in-set-conv-nth)
  moreover
  let \( ?dp = \text{drop} \ i \ p \)
from \( i \) have \( ?dp \neq [] \) by simp
from \( i \) have \( \text{hd} \ ?dp = u \) by (simp add: hd-drop-conv-nth)
moreover from \( i \) have \( \text{last} \ ?dp = \text{last} \ p \) by simp
moreover {
  fix \( k \)
  assume \( 1 < \text{length} \ ?dp \)
  and \( k < \text{length} \ ?dp - 1 \)
  hence \( l: 1 < \text{length} \ p \ k+i < \text{length} \ p - 1 \) by (auto)
  with \( p \) have \( p!(k+i), \ p!\text{Suc} \ (k+i) \in \text{tree-edges} \ s \) by (auto simp add: lowlink-path-def)
moreover from \( l \) have \( i+k \leq \text{length} \ p \ i+\text{Suc} \ k \leq \text{length} \ p \) by simp-all
ultimately have \( (\text{?dp}!k,\ ?dp!\text{Suc} \ k) \in \text{tree-edges} \ s \) by (simp add: add.commute)
} note \( \text{aux} = \text{this} \)
moreover {
  assume \( \ast: 1 < \text{length} \ ?dp \)
  hence \( l: 1 + i < \text{length} \ p \) by simp
with $s'.lowlink-path\text{-}finished[\text{OF } p]$ have $p ! (1+i) \in \text{dom } (\text{finished } ?s)$ by auto
moreover from $l$ have $i+1 \leq \text{length } p$ by simp
ultimately have $?dp!1 \in \text{dom } (\text{finished } ?s)$ by simp
moreover from aux[of 0] * have $(?dp!0, ?dp!\text{Suc 0}) \in \text{tree\text{-}edges } s$
by simp
with $\text{hd } ?dp = w$ hd\text{-}conv\text{-}nth[of $?dp] * have $(u, ?dp!\text{Suc 0}) \in \text{tree\text{-}edges } s$
by simp
with no\text{-}self\text{-}loop\text{-}in\text{-}tree have $?dp!1 \neq u$ by auto
ultimately have $?dp!1 \in \text{dom } (\text{finished } s)$ by simp
moreover from aux[of 0] * have $(?dp!0, ?dp!\text{Suc 0}) \in \text{tree\text{-}edges } s$
by simp
with $\langle \text{hd } ?dp = u \rangle$ hd\text{-}conv\text{-}nth[of $\text{Suc 0}$]
ultimately have $?dp!1 \in \text{dom } (\text{finished } s)$ by simp
moreover from $p$ have $P$: path $E$ w $p$ ll by (simp add: lowlink-path-def)
ultimately have lowlink-path $s$ u ?dp ll using $p$ by (simp add: lowlink-path-def)
moreover from u\text{-}path path\text{-}is\text{-}trancl (?dp \neq []) have $(u, ll) \in E^+$ by force
moreover { from ll (ll \neq w) have $(ll, w) \in E^+$ by (auto simp add: lowlink-set-def)
also from path-u path\text{-}is\text{-}rtrancl have $(w, u) \in E^+$ by metis
finally have $(ll, u) \in E^+$ .
}
moreover note ll u-disc
ultimately have ll \in lowlink-set $s$ u unfolding lowlink-set-def by auto
thus $\text{thesis}$ by auto
qed
qed
qed

hence ll-sub-rev': $?L' \subseteq ?L \cup ?Lu$ by auto

have ref-ne: stack $?s \neq [] \implies$
lowlink $?s = (\text{lowlink } s)(\text{hd } \text{stack } ?s) \mapsto \min (\zeta s (\text{hd } \text{stack } ?s)) (\zeta s u))
apply (rule TRANS)
unfolding tarjan-fin-def tj-stack-pop-def
by refine-vcg simp-all

have ref-e: stack $?s = [] \implies$ lowlink $?s = \text{lowlink } s$
apply (rule TRANS)
unfolding tarjan-fin-def tj-stack-pop-def

} 217
by refine-vcg simp-all

have ref-tj: \( \zeta s u \neq \delta s u \implies tj\text{-}stack ?s = tj\text{-}stack s \)
apply (rule TRANS)
unfolding tarp-refresh tj-stack-pop-def
by refine-vcg simp-all

have \( \zeta \) ?s w = LowLink ?s w
proof
(cases w = hd (stack ?s) \& stack ?s \neq [])
case True note all-True = th
with ref-ne have *: \( \zeta ?s w = \min (\zeta s w) (\zeta s u) \) by simp
show ?thesis
proof
(cases \( \zeta \) s u < \( \zeta \) s w)
case False with * finish w-disc have \( \zeta ?s w = \text{LowLink } s w \) by simp
also have \( \text{LowLink } s w = \text{LowLink } ?s w \)
proof (rule LowLink-eqI [OF inv' ll-sub ll-sub-rev])
from w-disc show w \in dom (discovered s) by simp
fix ll assume ll \in lowlink-set s u
hence \( \text{LowLink } s u \leq \delta s ll \) by simp
moreover from False finish w-disc u-disc have \( \text{LowLink } s w \leq \text{LowLink } s u \) by simp
ultimately have \( \text{LowLink } s w \leq \delta ?s ll \) by simp
qed simp
finally show ?thesis.
next
case True note \( \zeta \) rel = th
have \( \text{LowLink } s u \in \tilde{\mathcal{L}}' \)
proof
from all-True finish have w-tl: w \in set (tl (stack s)) by auto
obtain ll where ll: ll \in lowlink-set s u \( \delta s ll = \text{LowLink } s u \)
using Min-in[of \( ?L' u \)] llowlink-set-finite lowlink-set-not-empty w-disc
by fastforce
have ll \in lowlink-set ?s w
proof (cases \( \delta s u = \zeta s u \))
case True
moreover from w-tl finish tl-lt-stack-hd-discover have \( \delta s w \leq \delta s u \)
by simp
moreover from w-disc have \( \text{LowLink } s w \leq \delta s w \)
by (simp add: LowLink-le-disc)
with w-disc finish have \( \zeta s w \leq \delta s w \) by simp
moreover note \( \zeta \) rel
ultimately have False by force
thus ?thesis ..
next
case False with u-disc finish ll have u \neq ll by auto
with ll have
e: (ll, u) \in E^+ (u, ll) \in E^+ and
p: \exists p. lowlink-path s u p ll and
ll-disc: \( ll \in \text{dom} \) (discovered \( s \))
by (auto simp: lowlink-set-def)

from \( p \) have \( p' : \exists p. \text{lowlink-path} \ ?s u p \ ll \)
unfolding lowlink-path-def
by auto
from \( w-l \) tl-stack-hd-tree-path finish have \( T: (w,u) \in (\text{tree-edges} \ ?s) \)
\( ?s \) \( w \ p \ ll \)

moreover from \( T \) trancl-mono-mp[OF \( s' \).tree-edges-ssE] have \( (w,u) \in E^+ \)
by blast

moreover have \( (x,w) \in E^+ \)
proof (rule rtrancl-eq-or-trancl[THEN iffD2], safe)
assume \( x \neq w \) with all-True \( x \)
with \( \text{hd-in-set} \) finish stack-ss-tj-stack have \( u \in \text{set} \ (\text{tl-stack} \ ?s) \)
by auto
with \( s' \).tl-stack-hd-tree-path all-True have \( (x,w) \in (\text{tree-edges} s)^+ \)
by auto
with trancl-mono-mp[OF tree-edges-ssE] show \( (x,w) \in E^+ \)
by simp
qed
finally have \( (ll,w) \in E^+ \).
}

moreover note ll-disc
ultimately show \( ?\text{thesis} \) by (simp add: lowlink-set-def)
qed

hence \( \delta \ s \ ll \in \ ?L' \) by auto
with \( ll \) show \( ?\text{thesis} \) by simp
qed

hence \( \text{LowLink} \ ?s \ w \leq \text{LowLink} \ s \ u \)
using Min-le-iff[of \?L] s'.lowlink-set-not-empty w-disc s'.lowlink-set-finite
by fastforce
also from True \( u \)disc \( w \)disc finish have \( \text{LowLink} \ s \ u < \text{LowLink} \ s \ w \)
by simp

hence \( \text{Min} \ (\ ?L \cup \ ?L_u) = \text{LowLink} \ s \ u \)
using Min-Un[of \?L] \?L_u lowlink-set-finite lowlink-set-not-empty w-disc

w-disc
by simp
hence \( \text{LowLink} \ s \ u \leq \text{LowLink} \ ?s \ w \)
using Min-antimono[OF \( ll \)-sub-rev'] lowlink-set-finite s'.lowlink-set-not-empty
\[ w\text{-disc} \]

by auto
also from True \( u \)-disc finish * have \( \text{LowLink} \ s \ u = \zeta \ s \ w \) by simp
finally show \( \text{thesis} \)
qed
next
case False note all-False = this
proof (cases stack \( \tilde{s} \) = [])
case True with ref-e show \( \text{thesis} \) by simp
next
case False with all-False have \( w \neq \text{hd} \) (stack \( \tilde{s} \)) by simp
with False ref-ne show \( \text{thesis} \) by simp
qed
also from finish have \( \zeta \ s \ w = \text{LowLink} \ s \ w \) using \( w\text{-disc} \) by simp
also {
  fix \( v \)
  assume \( v \in \text{lowlink-set} \ s \ u \)
  and \( * : v \notin \text{lowlink-set} \ s \ w \)
  hence \( v \neq w \ w \neq u \) by (auto simp add: \( \text{lowlink-set-def} \))
  have \( v \notin \text{lowlink-set} \ s \ w \)
  proof (rule notI)
    assume \( v : v \in \text{lowlink-set} \ s \ w \)
    hence \( e : (v,w) \in E^* \ (w,v) \in E^* \)
    and \( v\text{-disc}: v \in \text{dom} \ (\text{discovered} \ s) \) by (auto simp add: \( \text{lowlink-set-def} \))

from \( v \ (w \neq w) \) obtain \( p \) where \( p : \text{lowlink-path} \ s \ w p v \) by (auto simp add: \( \text{lowlink-set-def} \))
  hence \( [\text{simp}] : p \neq [] \) by (simp add: \( \text{lowlink-path-def} \))

from \( p \) have \( \text{hd} \ p = w \) by (auto simp add: \( \text{lowlink-path-def} \) \text{path-hd})

show False
proof (cases \( u \in \text{set} \ p \))
case False hence \( \text{lowlink-path} \ s \ w p v \)
  using \( p \) by (auto simp add: \( \text{lowlink-path-def} \))
with \( e \ v\text{-disc} \) have \( v \in \text{lowlink-set} \ s \ w \) by (auto intro: \( \text{lowlink-setI} \))
with \( * \) show False ..
next
case True
then obtain \( i \) where \( i : i < \text{length} \ p \) \( p!i = u \)
  by (metis \( \text{in-set-conv-nth} \))
show False
proof (cases \( i \))
case 0 with \( i \) have \( \text{hd} \ p = u \) by (simp add: \( \text{hd-conv-nth} \))
  with \( \langle \text{hd} \ p = w \rangle \ (w \neq u) \) show False by simp
next
case \( \text{Suc} \ n \) with \( i \) \( p \) have \( * : (p!n,u) \in \text{tree-edges} \ s \ n < \text{length} \ p \)
  unfolding \( \text{lowlink-path-def} \)

220
by auto
with tree-edge-imp-discovered have \( p!n \in \text{dom} \) (discovered \( s \)) by auto

moreover from finish \( \text{hd-in-set} \) stack-not-finished have \( u \notin \text{dom} \) (finished \( s \)) by auto with * have \( pn-n-fin: p!n \notin \text{dom} \) (finished \( s \)) by (metis tree-edge-impl-parenthesis)
moreover from * \( \text{no-self-loop-in-tree} \) have \( p!n \neq u \) by blast
ultimately have \( p!n \in \text{set} \) (stack ?\( s \)) using stack-set-def finish by (cases stack ?\( s \)) auto

hence \( \text{s-ne} \): stack ?\( s \) \( \notin \) \( \text{[]} \) by auto
with all-False have \( w \neq \text{hd} \) (stack ?\( s \)) by simp
from stack-is-tree-path finish obtain \( v_0 \) where
path (tree-edges \( s \)) \( v_0 \) (rev (stack ?\( s \))) \( u \) by auto
with \( \text{s-ne} \) have \( \text{(hd (stack ?s), u) \in tree-edges s} \) by (auto simp: neq-Nil-conv path-simps)
with * \( \text{tree-eq-rule} \) have **: \( \text{hd (stack ?s)} = p!n \) by simp
show ?thesis
proof (cases \( n \))
case \( 0 \) with * have \( \text{hd p} = p!n \) by (simp add: hd-conv-nth)
with (\( \text{hd p} = w \)) ** have \( \text{w} = \text{hd (stack ?s)} \) by simp
with (\( \text{w} \neq \text{hd (stack ?s)} \)) show False ..
next
case (Suc \( m \)) with ** \( \text{s'.lowlink-path-finished[OF p, where} \)
\( j=n \) have \( \text{hd (stack ?s) \in dom (finished ?s) by simp} \)
with \( \text{hd-in-set[OF s-ne] s'.stack-not-finished} \) show ?thesis by blast
qed
qed
qed

with \( \text{finish} \) show ?case by (auto simp: pw-leaf-iff)
qed simp-all
qed
end end

context Tarjan-invar begin context begin interpretation timing-syntax .
lemmas lowlink-eq-LowLink = i-lowlink-eq-LowLink[THEN make-invar-thm, rule-format]

lemma lowlink-eq-disc-iff-scc-root:
assumes $v \in \text{dom} (\text{finished } s) \lor (\text{stack } s \neq [] \land v = \text{hd} (\text{stack } s) \land \text{pending } s$ $\{v\} = \{\})$

shows $\zeta s v = \delta s v \iff \text{scc-root } s v (\text{scc-of } E v)$

proof
from assms have $v \in \text{dom} (\text{discovered } s)$ using finished-discovered hd-in-set stack-discovered by blast

hence $\zeta s v = \text{LowLink } s v$ using lowlink-eq-LowLink by simp
with LowLink-eq-disc-iff-scc-root[of assms] show $\text{thesis}$ by simp

qed

lemma nc-sccs-eq-reachable:
assumes NC: $\neg \text{cond } s$
shows reachable $= \bigcup sccs s$

proof
from nc-finished-eq-reachable NC have [simp]: reachable $= \text{dom} (\text{finished } s)$ by simp
with sccs-finished show $\bigcup sccs s \subseteq \text{reachable}$ by simp

from NC have stack $s = []$ by (simp add: cond-alt)
with stacks-eq-iff have tj-stack $s = []$ by simp
with finished-ss-sccs-tj-stack show reachable $\subseteq \bigcup sccs s$ by simp

qed

end

class

context

begin
lemma tarjan-fin-nofail:
assumes pre-on-finish $u s'$
shows nofail (tarjan-fin $u s'$)

proof
from assms obtain $s$ where $s$: DFS-invar $G$ tarjan-params $s$ stack $s \neq []$ $u = \text{hd} (\text{stack } s) s' = \text{finish } u s \text{ cond } s \text{ pending } s$ $\{u\} = \{\}$

by (auto simp: pre-on-finish-def)
then interpret Tarjan-invar where $s=s$ by simp

from $s$ hd-stack-in-tj-stack have $u \in \text{set } (\text{tj-stack } s')$ by simp

moreover from $s$ tj-stack-distinct have distinct (tj-stack $s'$) by simp
moreover have the (lowlink $s' u) = \text{the } (\text{discovered } s' u) \iff \text{scc-root } s' u (\text{scc-of } E u)$

proof
from $s$ have the (lowlink $s' u) = \text{the } (\text{discovered } s' u) \iff \text{the } (\text{lowlink } s u)
= \text{the } (\text{discovered } s u)$ by simp
also from $s$ lowlink-eq-disc-iff-scc-root have ... $\iff \text{scc-root } s u (\text{scc-of } E u)$ by blast
also from $s$ scc-root-transfer'[where $s'=s']$ have ... $\iff \text{scc-root } s' u (\text{scc-of } E u)$ by simp
finally show $\text{thesis}$.

qed
ultimately show $\text{thesis}$

222
unfolding tarjan-fin-def tj-stack-pop-def
by simp
qed

sublocale DFS G tarjan-params
by unfold-locales (simp-all add: tarjan-disc-def tarjan-back-def tarjan-fin-nofail)
end

interpretation tarjan: Tarjan-def for G .

2.7.4 Interface

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

definition tarjan-spec G ≡ do {
  ASSERT (fb-graph G);
  SPEC (λseccs. (∀ scc ∈ seccs. is-scc (g-E G) scc)
               ∧ ∪ seccs = tarjan.reachable TYPE('a) G))

lemma tarjan-correct:
  tarjan G ≤ tarjan-spec G
unfolding tarjan-def tarjan-spec-def
proof (refine-vcg le-ASSERTI order-trans[OF DFS.it-dfs-correct])
  assume fb-graph G
  then interpret fb-graph G .
  interpret Tarjan ..
  show DFS G (tarjan.tarjan-params TYPE('b) G) ..
next
  fix s
  assume C: DFS-invar G (tarjan.tarjan-params TYPE('b) G) s ∧ ¬ tarjan.cond TYPE('b) G s
  then interpret Tarjan-invar G s by simp
from seccs-are-seccs show ∀ scc∈seccs s. is-scc (g-E G) scc .
from ne-seccs-eq-reachable C show ∪ seccs s = tarjan.reachable TYPE('b) G by simp
qed
end