A Framework for Verifying Depth-First Search Algorithms

Peter Lammich and René Neumann

April 19, 2020
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 Paranthesis 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
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** \( \wedge \) more init. dflt-parametrization more init \( \equiv \) []
on-init = init,
on-new-root = \( \lambda \). RETURN o more,
on-discover = \( \lambda \cdot \). RETURN o more,
on-finish = \( \lambda \cdot \). RETURN o more,
on-back-edge = \( \lambda \cdot \). RETURN o more,
on-cross-edge = \( \lambda \cdot \). RETURN o more,
is-break = \( \lambda \cdot \). False []

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

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

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

**1.1.2 DFS operations**

**Node predicates**

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

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

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

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

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

\[
\begin{align*}
\text{discover } u v s & \equiv \text{let } \\
& \quad d = (\text{discovered } s)(v \mapsto \text{counter } s); \ c = \text{counter } s + 1; \\
& \quad st = v \# \text{stack } s; \\
& \quad p = \text{pending } s \cup \{v\} \times E''\{v\}; \\
& \quad t = \text{insert } (u,v) \ (\text{tree-edges } s) \\
& \text{in } s(|\text{discovered} := d, \text{counter} := c, \text{stack} := st, \text{pending} := p, \text{tree-edges} := t|)
\end{align*}
\]

\textbf{lemma} \ discover-simps[simp]:

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

\textbf{definition} \ finish

\[
\begin{align*}
\text{finish } u s & \equiv \text{let } \\
& \quad f = (\text{finished } s)(u \mapsto \text{counter } s); \ c = \text{counter } s + 1; \\
& \quad st = \text{tl } (\text{stack } s) \\
& \text{in } s(|\text{finished} := f, \text{counter} := c, \text{stack} := st|)
\end{align*}
\]

\textbf{lemma} \ finish-simps[simp]:

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

\textbf{definition} \ back-edge

\[
\begin{align*}
\text{back-edge } u v s & \equiv \text{let }
\end{align*}
\]
\[ b = \text{insert } (u,v) (\text{back-edges } s) \]
\[ \text{in } s(\text{back-edges } := b) \]

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

by (simp-all add: back-edge-def)

**Definition** cross-edge ::
\[ 'v \Rightarrow ('v,'es) \text{ state-scheme } \Rightarrow ('v,'es) \text{ state-scheme} \]

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

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

by (simp-all add: cross-edge-def)

**Definition** new-root ::
\[ 'v \Rightarrow ('v,'es) \text{ state-scheme } \Rightarrow ('v,'es) \text{ state-scheme} \]

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

**Lemma** new-root-simps[simp]:
- \( \text{counter (new-root } v0 s) = \text{Suc (counter } s) \)
- \( \text{discovered (new-root } v0 s) = (\text{discovered } s)(v0 \mapsto \text{counter } s) \)
- \( \text{finished (new-root } v0 s) = \text{finished } s \)
stack \( (\text{new-root } v_0 s) \) = \([v_0]\)

pending \( (\text{new-root } v_0 s) \) = \((\{v_0\} \times E''\{v_0\})\)

tree-edges \( (\text{new-root } v_0 s) \) = tree-edges \( s \)

cross-edges \( (\text{new-root } v_0 s) \) = cross-edges \( s \)
=back-edges \( (\text{new-root } v_0 s) \) = back-edges \( s \)

state.more \( (\text{new-root } v_0 s) \) = state.more \( s \)

by \((\text{simp-all add: new-root-def})\)

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

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

**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.

\[
\text{definition do-cross-edge} \\
\quad :: \text{'}v \Rightarrow \text{'}v \Rightarrow (\text{'}v,\text{'}es) \text{ state-scheme } \Rightarrow (\text{'}v,\text{'}es) \text{ state-scheme nres}
\]

where

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

\[
\text{definition do-back-edge} \\
\quad :: \text{'}v \Rightarrow \text{'}v \Rightarrow (\text{'}v,\text{'}es) \text{ state-scheme } \Rightarrow (\text{'}v,\text{'}es) \text{ state-scheme nres}
\]

where

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

\begin{align*}
e & \leftarrow \text{on-back-edge param } u \ v \ s; \\
\text{return } & (s[\text{state}\text{.more} := e]) \\
\end{align*}

\textbf{definition} \ do-known-edge \\
\textbf{:} \ 'v \Rightarrow \ 'v \Rightarrow (\ 'v, 'es) \text{ state-scheme } \Rightarrow (\ 'v, 'es) \text{ state-scheme } \text{nres}

\textbf{where}
\begin{align*}
do-known-edge \ u \ v \ s & \equiv \\
\text{if } & \text{is-finished } v \ s \text{ then} \\
\text{do-cross-edge } & u \ v \ s \\
\text{else} \\
\text{do-back-edge } & u \ v \ s
\end{align*}

\textbf{definition} \ do-discover \\
\textbf{:} \ 'v \Rightarrow \ 'v \Rightarrow (\ 'v, 'es) \text{ state-scheme } \Rightarrow (\ 'v, 'es) \text{ state-scheme } \text{nres}

\textbf{where}
\begin{align*}
do-discover \ u \ v \ s & \equiv \text{do} \\
\text{let } & s = \text{discover } u \ v \ s; \\
e & \leftarrow \text{on-discover param } u \ v \ s; \\
\text{return } & (s[\text{state}\text{.more} := e])
\end{align*}

\textbf{definition} \ do-finish \\
\textbf{:} \ '(v, 'es) \text{ state-scheme } \Rightarrow (\ 'v, 'es) \text{ state-scheme } \text{nres}

\textbf{where}
\begin{align*}
do-finish \ u \ s & \equiv \text{do} \\
\text{let } & s = \text{finish } u \ s; \\
e & \leftarrow \text{on-finish param } u \ s; \\
\text{return } & (s[\text{state}\text{.more} := e])
\end{align*}

\textbf{definition} \ get-new-root \ \textbf{where}
\begin{align*}
\text{get-new-root } \ s & \equiv \text{SPEC} (\lambda v. \ v \in V0 \land \neg \text{is-discovered } v \ s)
\end{align*}

\textbf{definition} \ do-new-root \ \textbf{where}
\begin{align*}
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}\text{.more} := e])
\end{align*}

\textbf{lemmas} \ op-defs = \text{discover-def} \text{ finish-def} \text{ back-edge-def} \text{ cross-edge-def} \text{ new-root-def}

\textbf{lemmas} \ do-defs = \ do-discover-def \ do-finish-def \ do-known-edge-def
\text{ do-cross-edge-def} \text{ do-back-edge-def} \text{ do-new-root-def}

\textbf{lemmas} \ pred-defs = \text{is-discovered-def} \text{ is-finished-def} \text{ is-empty-stack-def}

\textbf{definition} \ init \ \equiv \text{ 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

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

\[ \land \neg \text{is-break param } s \]

**lemma** cond-alt:

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

**apply** (rule ext)

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

**by** auto

**definition** get-pending ::

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

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

**where** get-pending s ≡ do {
let u = hd (stack s);
let Vs = pending s ''. {u};

if Vs = {} then
RETURN (u,None,s)
else do {
  v ← RES Vs;
  let s = s | pending := pending s - {(u,v)};
  RETURN (u,Some v,s)
}
}

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

**where**

\[ \text{step } s \equiv \]

if is-empty-stack s then do {
  v0 ← get-new-root s;
  do-new-root v0 s
} else do {
  (u,Vs,s) ← get-pending s;
  case Vs of
    None ⇒ do-finish u s
  | Some v ⇒ do {
    if is-discovered v s then
      ...
\texttt{do-known-edge}\;u\;v\;s
else
\texttt{do-discover}\;u\;v\;s
\}
}

\textbf{definition}\;\texttt{it-dfs} \equiv \texttt{init} \Rightarrow \texttt{WHILE}\;\texttt{cond}\;\texttt{step}
\textbf{definition}\;\texttt{it-dfsT} \equiv \texttt{init} \Rightarrow \texttt{WHILET}\;\texttt{cond}\;\texttt{step}
\end{verbatim}

\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 \texttt{param-DFS} locale.

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

\texttt{locale}\;\texttt{param-DFS} =
\texttt{fb-graph}\;G + \texttt{param-DFS-defs}\;G\;\texttt{param}
\texttt{for}\;G::(v,\;more)\;\texttt{graph-rec-scheme}
\texttt{and}\;\texttt{param}::(v,es)\;\texttt{parameterization}
\texttt{begin}

\textbf{definition}\;\texttt{is-invar}::((v,\;es)\;\texttt{state-scheme} \Rightarrow \texttt{bool}) \Rightarrow \texttt{bool}
\texttt{—}\;\texttt{Predicate}\;\texttt{that}\;\texttt{states}\;\texttt{that}\;\texttt{I}\;\texttt{is}\;\texttt{an}\;\texttt{invariant}.
\texttt{where}\;\texttt{is-invar}\;I \equiv \texttt{is-rwof-invar}\;\texttt{init}\;\texttt{cond}\;\texttt{step}\;I

\texttt{end}

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

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

\texttt{lemma}\;\texttt{make-invar-thm:}\;\texttt{is-invar}\;I \Rightarrow I\;s

10
— Lemma to transfer an invariant into this locale  
**using** *rwof-cons[OF - rwof, folded is-invar-def]*.

**Establishing Invariants**

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

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

**lemmas** indep-invar-rules =
- leof-True-rule[where \( m = \text{on-init} \) param]
- leof-True-rule[where \( m = \text{on-new-root} \) param \( v_0 \) \( s' \) for \( v_0 \) \( s' \)]
- leof-True-rule[where \( m = \text{on-discover} \) param \( u \) \( v \) \( s' \) for \( u \) \( v \) \( s' \)]
- leof-True-rule[where \( m = \text{on-finish} \) param \( v \) \( s' \) for \( v \) \( s' \)]
- leof-True-rule[where \( m = \text{on-cross-edge} \) param \( u \) \( v \) \( s' \) for \( u \) \( v \) \( s' \)]

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

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

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

We define symbolic names for the preconditions of certain operations

**definition** pre-is-break s \( \equiv \) DFS-invar G param s

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

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

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

— Establish a DFS invariant (explicit preconditions).

**assumes** init: on-init param \( \leq_n \) SPEC (λx. I (empty-state x))

**assumes** new-root: \( \forall s \ s' v0 \).

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

\quad \text{SPEC } (\lambda x. DFS-invar G param (s'\{\text{state}\_\text{more } := x\})) 
\implies I (s'\{\text{state}\_\text{more } := x\}))
\]

**assumes** finish: \( \forall s \ s' u \).

\[
[DFS-invar G param s; I s; cond s; \neg is-break param s; 
\text{stack } s \neq []; u = \text{hd } (\text{stack } s); 
\text{pending } s' \{ u \} = \{\}; 
\quad s' = \text{finish } u s] 
\implies \quad \text{on-finish } param u s' \leq_n 

\quad \text{SPEC } (\lambda x. DFS-invar G param (s'\{\text{state}\_\text{more } := x\})) 
\implies I (s'\{\text{state}\_\text{more } := x\}))
\]

**assumes** cross-edge: \( \forall s \ s' u v \).

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

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

assumes discover: \( \forall s s' u v \cdot \)
\[ \begin{align*}
\text{DFS-invar } G \text{ param } s; & I s; \quad \text{cond } s; \quad \neg \text{is-break } \text{param } s; \\
& \text{stack } s \neq \text{}; \quad (u, v) \in \text{pending } s; \quad u = \text{hd } (\text{stack } s); \\
& v \notin \text{dom } (\text{discovered } s); \\
& s' = \text{discover } u v (s[\text{pending} := \text{pending } s - \{(u,v)\}]]) \\
\implies & \text{on-discover } \text{param } u v s' \leq n \\
\end{align*} \]

shows is-invar I

unfolding is-invar-def

proof
show \( \text{init} \leq_n \text{SPEC } I \)

unfolding init-def
by (refine-rcg refine-vcg) (simp add: init)

next
fix \( s \)
assume rwoff init cond step s and IC: \( I s \text{ cond } s \)

hence DI: DFS-invar G param s by unfold-locales

then interpret DFS-invar G param s.

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

have B: \( \text{step } s \leq_n \text{SPEC } (\text{DFS-invar } G \text{ param}) \)

by rule (metis DFS-invar-step' DI (cond s))

note rule-assms = DI IC IB

show \( \text{step } s \leq_n \text{SPEC } I \)
apply (rule leof-use-spec-rule[OF B])

unfolding step-def do-defs pred-defs get-pending-def get-new-root-def
apply (refine-rcg refine-vcg)
apply (simp-all)

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

qed
\textbf{lemma} \textit{establish-invarI} \textit{[case-names init new-root finish cross-edge back-edge discover]}:

— Establish a DFS invariant (symbolic preconditions).

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

\textbf{assumes} \textit{new-root}: \(\lambda s' v0.\) pre-on-new-root \(v0\ s'\)

\(\Rightarrow\) on-new-root param \(v0\ s' \leq_n\)

SPEC (\(\lambda x. DFS\-invar\ G\ param\ (s'[state\more := x])\))

\(\Rightarrow I (s'[state\more := x]))\)

\textbf{assumes} \textit{finish}: \(\lambda s' u.\) pre-on-finish \(u\ s'\)

\(\Rightarrow\) on-finish param \(u\ s' \leq_n\)

SPEC (\(\lambda x. DFS\-invar\ G\ param\ (s'[state\more := x])\))

\(\Rightarrow I (s'[state\more := x]))\)

\textbf{assumes} \textit{cross-edge}: \(\lambda s' u v.\) pre-on-cross-edge \(u\ v\ s'\)

\(\Rightarrow\) on-cross-edge param \(u\ v\ s' \leq_n\)

SPEC (\(\lambda x. DFS\-invar\ G\ param\ (s'[state\more := x])\))

\(\Rightarrow I (s'[state\more := x]))\)

\textbf{assumes} \textit{back-edge}: \(\lambda s' u v.\) pre-on-back-edge \(u\ v\ s'\)

\(\Rightarrow\) on-back-edge param \(u\ v\ s' \leq_n\)

SPEC (\(\lambda x. DFS\-invar\ G\ param\ (s'[state\more := x])\))

\(\Rightarrow I (s'[state\more := x]))\)

\textbf{assumes} \textit{discover}: \(\lambda s' u v.\) pre-on-discover \(u\ v\ s'\)

\(\Rightarrow\) on-discover param \(u\ v\ s' \leq_n\)

SPEC (\(\lambda x. DFS\-invar\ G\ param\ (s'[state\more := x])\))

\(\Rightarrow I (s'[state\more := x]))\)

\textbf{shows} \textit{is-invar I}

\textbf{apply} (rule \textit{establish-invarI})

\textbf{using} \textit{assms}

\textbf{unfolding} \textit{pre-on-defs}

\textbf{apply} -

\textbf{apply} blast

\textbf{apply} (rprems,blast)+

\textbf{done}

\textbf{lemma} \textit{establish-invarI-ND} \textit{[case-names prereq init new-discover finish cross-edge back-edge]}:

— Establish a DFS invariant (new-root and discover cases are combined).

\textbf{assumes} \textit{prereq}: \(\lambda u v s.\) on-discover param \(u\ v\ s =\) on-new-root param \(v\ s\)

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

\textbf{assumes} \textit{new-discover}: \(\lambda s s' v.\)

\([[DFS\-invar\ G\ param\ s; I\ s;\ cond\ s;\ ¬\ is\-break\ param\ s;\ v\ \notin\ \text{dom}\ (\text{discovered}\ s);\ discovered\ s' = (\text{discovered}\ s)(v\rightarrow\text{counter}\ s);\ finished\ s' = \text{finished}\ s;\ counter\ s' = \text{Suc\ (counter}\ s);\ stack\ s' = v\#\text{stack}\ s; back\-edges\ s' = \text{back\-edges}\ s; cross\-edges\ s' = \text{cross\-edges}\ s; tree\-edges\ s' \supseteq \text{tree\-edges}\ s; state\more\ s' = state\more\ s]]\)

\(\Rightarrow\) on-new-root param \(v\ s' \leq_n\)

SPEC (\(\lambda x. DFS\-invar\ G\ param\ (s'[state\more := x])\))

\(\Rightarrow I (s'[state\more := x]))\)

14
establish-invarI-lemma

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

assumes back-edge: \( \forall s s' u v. \)
\[
\text{DFS-invar } G \text{ param } s; I s; \text{ cond } s; \neg \text{ is-break } \text{ param } s;
\]
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)\);
\( s' = \text{back-edge } u v (s[\text{pending := pending } s - \{(u, v)\}]) \)
\( \implies \text{on-back-edge } \text{param } u v s' \leq_n \)
\( \text{SPEC } (\lambda x. \text{DFS-invar } G \text{ param } (s'[\text{state.more := } x])) \)
\( \implies I (s'[\text{state.more := } x]) \)

shows is-invar I

proof (induct rule: establish-invarI)
  case (new-root s) thus \( \forall \text{case by (auto intro!: new-discover)} \)
next
  case (discover s s' u v) hence
  on-new-root param v s' \leq_n \)
  \( \text{SPEC } (\lambda x. \text{DFS-invar } G \text{ param } (s'[\text{state.more := } x])) \)
  \( \implies I (s'[\text{state.more := } x]) \)
  by (auto intro!: new-discover)
with prereq show \( \forall \text{case by simp} \)

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

15
assumes finish: ∃s s' u.
\[\text{DFS-invar } G \text{ param } s; I; s; \text{ cond } s; \neg \text{-break param } s; \]
stack s ≠ []; u = hd (stack s);
pending s ‘\{u\} = {};

\[s' = \text{finish } u \text{ s} \]

\[\Rightarrow \text{on-finish param u s' ≤n} \]
SPEC (λx. DFS-invar G param (s’[state.more := x]))

\[\Rightarrow I (s'[state.more := x]) \]
assumes cross-back-edge: ∃s s' u v.
\[\text{DFS-invar } G \text{ param } s; I; s; \text{ cond } s; \neg \text{-break param } s; \]
stack s ≠ []; (u, v) ∈ pending s; u = hd (stack s);
v ∈ dom (discovered s);
discovered s’ = discovered s; finished s’ = finished s;
stack s’ = stack s; tree-edges s’ = tree-edges s; counter s’ = counter s;
pending s’ = pending s – {(u,v)};
cross-edges s’ ∪ back-edges s’ = cross-edges s ∪ back-edges s ∪ {(u,v)};
state.more s’ = state.more s

\[\Rightarrow \text{on-cross-edge param u v s' ≤n} \]
SPEC (λx. DFS-invar G param (s’[state.more := x]))

\[\Rightarrow I (s'[state.more := x]) \]
assumes discover: ∃s s' u v.
\[\text{DFS-invar } G \text{ param } s; I; s; \text{ cond } s; \neg \text{-break param } s; \]
stack s ≠ []; (u, v) ∈ pending s; u = hd (stack s);
v ∉ dom (discovered s);
s’ = discover u v (s[pending := pending s – {(u,v)}])

\[\Rightarrow \text{on-discover param u v s' ≤n} \]
SPEC (λx. DFS-invar G param (s’[state.more := x]))

\[\Rightarrow I (s'[state.more := x]) \]
shows is-invar I

proof (induct rule: establish-invarI)
next
\[\text{case cross-edge thus } ?\text{case by (auto intro!: cross-back-edge)}\]

\[\text{case (back-edge s s' u v) hence} \]

\[\text{on-cross-edge param u v s' ≤n} \]
SPEC (λx. DFS-invar G param (s’[state.more := x]))

\[\Rightarrow I (s'[state.more := x]) \]

by (auto intro!: cross-back-edge)

with prereq show ?case by simp

qed fact+

lemma establish-invarI-ND-CB [case-names prereq-ND prereq-CB init new-discover finish cross-back-edge]:
— Establish a DFS invariant (new-root/discover and cross/back-edge cases are combined).

assumes prereq:

\[\land \land u v s. \text{on-discover param u v s = on-new-root param v s} \]
\[\land \land u v s. \text{on-back-edge param u v s = on-cross-edge param u v s} \]
assumes init: on-init param ≤n SPEC (λx. I (empty-state x))

16
assumes new-discover: ∃s s' v.
[DFS-invar G param s; I s; cond s; ¬ is-break param s;
v ∉ dom (discovered s);
   discovered s' = (discovered s)(v→counter s); finished s' = finished s;
counter s' = Suc (counter s); stack s' = v#stack s;
back-edges s' = back-edges s; cross-edges s' = cross-edges s;
tree-edges s' ⊇ tree-edges s;
state.more s' = state.more s]
⇒ on-new-root param v s' ≤_n
   SPEC (λx. DFS-invar G param (s'[state.more := x]))
   → I (s'[state.more := x]))
assumes finish: ∃s s' u.
[DFS-invar G param s; I s; cond s; ¬ is-break param s;
stack s ≠ []; u = hd (stack s);
pending s " {u} = {};
s' = finish u s]
⇒ on-finish param u s' ≤_n
   SPEC (λx. DFS-invar G param (s'[state.more := x]))
   → I (s'[state.more := x]))
assumes cross-back-edge: ∃s s' u v.
[DFS-invar G param s; I s; cond s; ¬ is-break param s;
stack s ≠ []; (u, v) ∈ pending s; u = hd (stack s);
v ∈ dom (discovered s);
   discovered s' = discovered s; finished s' = finished s;
stack s' = stack s; tree-edges s' = tree-edges s; counter s' = counter s;
pending s' = pending s - {(u,v)};
cross-edges s' ∪ back-edges s' = cross-edges s ∪ back-edges s ∪ {(u,v)};
state.more s' = state.more s
⇒ on-cross-edge param u v s' ≤_n
   SPEC (λx. DFS-invar G param (s'[state.more := x]))
   → I (s'[state.more := x]))
shows is-invar I
proof (induct rule: establish-invar-I-ND)
case cross-edge thus ?case by (auto intro!: cross-back-edge)
next
case (back-edge s s' u v) hence
   on-cross-edge param u v s' ≤_n
   SPEC (λx. DFS-invar G param (s'[state.more := x]))
   → I (s'[state.more := x]))
by (auto intro!: cross-back-edge)
with prereq show ?case by simp
qed fact+

lemma is-invarI-full [case-names init new-root finish cross-edge back-edge discover]:
— Establish a DFS invariant not taking into account the parameterization.
assumes init: ∀e. I (empty-state e)
assumes new-root: ∃s s' v0 e.
\[ \{ I \ s; \ cond \ s; \ DFS-invar \ G \ param \ s; \ DFS-invar \ G \ param \ s' \; \\
\text{stack} \ s = []; \ v0 \notin \text{dom} \ (\text{discovered} \ s); \ v0 \in V0; \\
\; s' = \text{new-root} \ v0 \ s[[\text{state.more} := e]] \}
\implies I \ s'
\] 

**and** \text{finish}: \( \land s \ s' \ u \ e. \\
\{ I \ s; \ cond \ s; \ DFS-invar \ G \ param \ s; \ DFS-invar \ G \ param \ s' \; \\
\text{stack} \ s \neq []; \ \text{pending} \ s' = \{ u \}; \ u \in \text{dom} \ (\text{discovered} \ s); \\
\; s' = \text{finish} \ u \ s[[\text{state.more} := e]] \}
\implies I \ s'
\]

**and** \text{cross-edge}: \( \land s \ s' \ u \ v e. \\
\{ I \ s; \ cond \ s; \ DFS-invar \ G \ param \ s; \ DFS-invar \ G \ 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); \\
\; s' = \text{(cross-edge} \ u \ v \ (s[[\text{pending} := \text{pending} \ s - \{(u,v)\}]]))(\text{state.more} := e)]
\implies I \ s'
\]

**and** \text{back-edge}: \( \land s \ s' \ u \ v e. \\
\{ I \ s; \ cond \ s; \ DFS-invar \ G \ param \ s; \ DFS-invar \ G \ 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{(back-edge} \ u \ v \ (s[[\text{pending} := \text{pending} \ s - \{(u,v)\}]]))(\text{state.more} := e)]
\implies I \ s'
\]

**and** \text{discover}: \( \land s \ s' \ u \ v e. \\
\{ I \ s; \ cond \ s; \ DFS-invar \ G \ param \ s; \ DFS-invar \ G \ 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.more} := e)]
\implies I \ s'
\]

shows \text{is-invar I} 

apply (rule establish-invar1) 
apply (blast intro: indep-invar-rules assms)+ 
done

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

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

**and** \text{finish}': \( \land s \ s' \ u \ e. \\
\{ I \ s; \ cond \ s; \ DFS-invar \ G \ param \ s; \ DFS-invar \ G \ param \ s' \; \\
\text{stack} \ s \neq []; \ \text{pending} \ s' = \{ u \}; \\
\; u = \text{hd} \ (\text{stack} \ s); \\
\; s' = \text{finish} \ u \ s[[\text{state.more} := e]] \}
\implies I \ s'
\]
and visited': ∃ s s' u v e b.
[1 s; cond s; DFS-invar G param s; DFS-invar G param s'];
stack s = ]; v ∈ pending s {u}; v ∈ dom (discovered s);
stack s ≠ ]; v ∈ pending s {u}; v ∈ dom (discovered s);
u = hd (stack s);
cross-edges s ⊆ c; back-edges s ⊆ b;
s' = s;
pending := pending s - {(u,v)},
state.more := e,
cross-edges := c,
back-edges := b]]
⇒ I s'
and discover': ∃ s s' u v e.
[1 s; cond s; DFS-invar G param s; DFS-invar G param s'];
stack s ≠ ]; v ∈ pending s {u}; v ∈ dom (discovered s);
u = hd (stack s);
s' = (discover u v (s[pending := pending s - {(u,v)}])(state.more := e)]
⇒ I s'
shows is-invar I
proof (induct rule: is-invarI-full)
case (cross-edge s s' u v e) thus ?case
   apply -
      apply (rule visited"of s s' v u insert (u,v) (cross-edges s) back-edges s e]
      apply clarsimp-all
   done
next
   case (back-edge s s' u v e) thus ?case
      apply -
      apply (rule visited"of s s' v u cross-edges s insert (u,v) (back-edges s) e]
      apply clarsimp-all
   done
qed fact+
end

1.1.5 Basic Invariants
We establish some basic invariants

context param-DFS begin

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

lemma i-basic-invar: is-invar basic-invar
unfolding basic-invar-def[abs-def]
apply (induction rule: is-invarI)
apply (clarsimp-all simp: neq-Nil-conv last-tl)
apply blast+
done
end

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

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

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

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

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

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

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

lemma stack-not-finished:
  x ∈ set (stack s) ⇒ x \notin dom (finished s)
  using stack-set-def
  by auto

lemma discovered-not-stack-imp-finished:
  x ∈ dom (discovered s) ⇒ x \notin set (stack s) ⇒ x ∈ dom (finished s)
  using stack-set-def
  by auto

lemma finished-discovered:
\[ \text{dom} \, (\text{finished} \, s) \subseteq \text{dom} \, (\text{discovered} \, s) \]

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

**Lemma finished-no-pending:**
\[ v \in \text{dom} \, (\text{finished} \, s) \implies \text{pending} \, s = \{v\} = \{} \]
using basic-invar
by (auto simp add: basic-invar-def)

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

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

**Lemma empty-stack-imp-empty-pending:**
\[ \text{stack} \, s = [] = \{} \implies \text{pending} \, s = \{\} \]
by auto

end

**Context param-DFS begin**

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

**Definition discovered-closed s ≡**
\[ E\cdot \text{dom} \, (\text{finished} \, s) \subseteq \text{dom} \, (\text{discovered} \, s) \]
\[ \land (E - \text{pending} \, s) = \text{set} \, (\text{stack} \, s) \subseteq \text{dom} \, (\text{discovered} \, s) \]

**Lemma i-discovered-closed: is-invar discovered-closed**
**Proof** (induct rule: is-invarI)
\[ \text{case (finish} \, s \, s') \]
\[ \text{hence (finish} \, s \, s') \]
by (simp add: discovered-closed-def)
moreover from finish have \text{set} \, (\text{stack} \, s') \subseteq \text{set} \, (\text{stack} \, s)
by (auto simp add: neq-Nil-conv cond-def)
ultimately have \((E - pending s')^\sim set (stack s') \subseteq dom (discovered s')\)

using \texttt{finish} 
by \texttt{simp blast}

moreover 
from \texttt{\(\langle stack s \neq [] \rangle\) finish have \(E^\sim dom (finished s') \subseteq dom (discovered s')\)}
apply (cases \texttt{stack s}) apply \texttt{simp}
apply (simp add: discovered-closed-def)
apply (blast)
done
ultimately show \(?case by (simp add: discovered-closed-def)\)

\texttt{qed (auto simp add: discovered-closed-def cond-def)}

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

\texttt{end}

context DFS-invar
begin

\texttt{lemmas discovered-reachable = i-discovered-reachable [THEN make-invar-thm]}

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

\texttt{lemmas discovered-closed = i-discovered-closed [THEN make-invar-thm]}

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

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

\texttt{lemma finished-closed: \(E^\sim dom (finished s) \subseteq dom (discovered s)\)}

\texttt{using discovered-closed [unfolded discovered-closed-def] by auto}

\texttt{lemma finished-imp-succ-discovered:}
\(v \in dom (finished s) \Rightarrow w \in succ v \Rightarrow w \in dom (discovered s)\)

\texttt{using discovered-closed [unfolded discovered-closed-def] by auto}

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

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

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

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

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

qed

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

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

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

end

1.1.6 Total Correctness

We can show termination of the DFS algorithm, independently of the pa-
rameterization

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

end

context param-DFS begin
lemma it-dfsT_eq_it-dfs:
  assumes [simp, intro!]: finite reachable
  shows it-dfsT = it-dfs
proof –
  have it-dfs \leq 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 \leq it-dfs
proof (cases nofail it-dfs)
  case False thus ?thesis by (simp add: not-nofail-iff)
next
case True

  show ?thesis
    unfolding it-dfsT-def it-dfs-def
apply (rule bind-mono)
apply simp
apply (subst WHILE-eq-WHILE-tproof[
  where I=DFS-invar G param
  and V=param-dfs-variant
])
apply auto []
apply (subst rwof-eq-DFS-invar [symmetric])
using rwof-init [OF True [unfolded it-dfs-def]]
apply (fastforce dest: order-trans) []
apply (rule SPEC-rule-conjI)
apply (rule DFS-invar-step [OF True], assumption+) []
apply (rule param-dfs-variant-step, (assumption|rule True)+) []
apply simp
done
qed
finally show ?thesis by simp
qed
end

1.1.7 Non-Failing Parameterization

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

locale DFS =
  param-DFS G param
for G :: ('v,'more) graph-rec-scheme
  and param :: ('v,'es) parameterization
+
assumes nofail-on-init:
  nofail (on-init param)
assumes nofail-on-new-root:
  pre-on-new-root v0 s ⇒ nofail (on-new-root param v0 s)
assumes nofail-on-finish:
  pre-on-finish u s ⇒ nofail (on-finish param u s)
assumes nofail-on-cross-edge:
  pre-on-cross-edge u v s ⇒ nofail (on-cross-edge param u v s)
assumes nofail-on-back-edge:
  pre-on-back-edge u v s ⇒ nofail (on-back-edge param u v s)
assumes nofail-on-discover:
pre-on-discover \( u \) \( v \) \( s \) \( \implies \) nofail (on-discover param \( u \) \( v \) \( s \))

begin

lemmas nofails = nofail-on-init nofail-on-new-root nofail-on-finish
\hspace{1em} nofail-on-cross-edge nofail-on-back-edge nofail-on-discover

lemma init-leaf-invar: \( \text{init} \leq_n \text{SPEC (DFS-invar G param)} \)
unfolding rwof-eq-DFS-invar[symmetric]
by (rule rwof-leaf-init)

lemma it-dfs-eq-spec: \( \text{it-dfs} = \text{SPEC (\( \lambda s. \text{DFS-invar G param s} \wedge \neg \text{cond s} \}} \)
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 \( \Phi=\lambda-. \text{True} \])
apply (refine-vcg leofD
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
\hspace{1em} get-pending-def get-new-root-def pred-defs
\hspace{1em} split: option_split)
apply (intro allI conjI impI nofails)
apply (auto simp add: pre-on-defs)
done

lemma it-dfs-correct: \( \text{it-dfs} \leq \text{SPEC (\( \lambda s. \text{DFS-invar G param s} \wedge \neg \text{cond s} \}} \)
by (simp add: it-dfs-eq-spec)

lemma it-dfs-SPEC:
assumes \( \forall s. \text{DFS-invar G param s} \wedge \neg \text{cond s} \implies P s \)
shows \( \text{it-dfs} \leq \text{SPEC P} \)
using weaken-SPEC[OF it-dfs-correct]
using assms
by blast

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

lemma it-dfsT-SPEC:
assumes finite reachable
assumes \( \forall s. \text{DFS-invar G param s} \wedge \neg \text{cond s} \implies P s \)
shows \( \text{it-dfsT} \leq \text{SPEC P} \)
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 ≡ the (discovered s v)
abbreviation the-finished s v ≡ the (finished s v)

locale timing-syntax
begin

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

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

definition timing-common-inv s ≡
— δ s v < ϕ s v
(∀ v ∈ dom (finished s). δ s v < ϕ s v)
— v ≠ w → δ s v ≠ δ s w ∧ ϕ s v ≠ ϕ s w
— Can’t use card dom = card ran as the maps may be infinite ... ∧ (∀ v ∈ dom (discovered s). ∀ w ∈ dom (discovered s). v ≠ w → δ s v ≠ δ s w)
∧ (∀ v ∈ dom (finished s). ∀ w ∈ dom (finished s). v ≠ w → ϕ s v ≠ ϕ s w)
— δ s v < counter ∧ ϕ s v < counter
∧ (∀ v ∈ dom (discovered s). δ s v < counter s)
∧ (∀ v ∈ dom (finished s). ϕ s v < counter s)
∧ (∀ v ∈ dom (finished s). ∀ w ∈ succ v. δ s w < ϕ 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-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

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 ∈ dom (discovered s) ⇒ δ s v < counter s
v ∈ dom (finished s) ⇒ ϕ s v < counter s
using s-timing-common-inv
by (auto simp add: timing-common-inv-def)

lemma disc-lt-fin:
v ∈ dom (finished s) ⇒ δ s v < ϕ s v
using s-timing-common-inv
by (auto simp add: timing-common-inv-def)

lemma disc-unequal:
  assumes v ∈ dom (discovered s) w ∈ dom (discovered s)
  and v ≠ w
  shows δ s v ≠ δ s w
using s-timing-common-inv assms
by (auto simp add: timing-common-inv-def)

lemma fin-unequal:
  assumes v ∈ dom (finished s) w ∈ dom (finished s)
  and v ≠ w
  shows ϕ s v ≠ ϕ s w
using s-timing-common-inv assms
by (auto simp add: timing-common-inv-def)

lemma finished-succ-fin:
  assumes v ∈ dom (finished s)
  and w ∈ succ v
  shows δ s w < ϕ s v
using assms s-timing-common-inv
by (simp add: timing-common-inv-def)
end

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

lemma i-prev-stack-discover-all:
is-invar (λs. ∀ n < length (stack s). ∀ v ∈ set (drop (Suc n) (stack s)).
  δ s (stack s ! n) > δ s v)
proof (induct rule: is-invarI)
case (finish s) thus ?case
by (cases stack s) auto 
next 
case (discover s s' u v)
hence EQ[simp]: discovered s' = (discovered s)(v ∈ counter s)  
    stack s' = v#stack s  
    by simp-all 

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

from stack-discovered timing-less-counter have
    \forall w. w ∈ set (stack s) ⇒ δ s w < counter s
    by blast

with v: v ∈ set (stack s) ⇒ δ s' w < δ s' v by auto

hence \forall w. w ∈ set (drop (Suc 0) (stack s')) ⇒ δ s' w < δ s' (stack s' ! 0)
    by auto

moreover
from v: v ∈ set (stack s) have
    \forall n. \[ n < (length (stack s')); n > 0 \]
    ⇒ δ s' (stack s' ! n) = δ s (stack s' ! n)
    by auto

with discover(1) v

have \forall n. \[ n < (length (stack s')) - 1; n > 0 \]
    ⇒ \forall w ∈ set (drop (Suc n) (stack s')). δ s' (stack s' ! n) > δ s' w
    by (auto dest: in-set-dropD)

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

qed simp-all

end end

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

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

lemma prev-stack-discover:
    \[ n < length (stack s); v ∈ set (drop (Suc n) (stack s)) \]
    ⇒ δ s (stack s ! n) > δ s v
    by (metis prev-stack-discover-all)

lemma Suc-stack-discover:
    assumes n: n < (length (stack s)) - 1
    shows δ s (stack s ! n) > δ s (stack s ! Suc n)

proof -
    from prev-stack-discover assms have
        \[ \forall v. v ∈ set (drop (Suc n) (stack s)) ⇒ δ s (stack s ! n) > δ s v \]
    by fastforce
moreover from \( n \) have stack \( s \) ! Suc \( n \in set (\text{drop} (\text{Suc} n) (\text{stack} s)) \)
using in-set-conv-nth by fastforce
ultimately show \(?thesis\).
qed

lemma tl-ht-stack-hd-discover:
assumes 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 notempty obtain \( y \ ys \) where stack \( s \) = \( y \# ys \) by (metis list.exhaust)
with assms show \(?thesis\)
using prev-stack-discover
by (cases \( ys \)) force+
qed

lemma stack-nth-order:
assumes \( l \): \( i < \text{length} (\text{stack} s) \) \( j < \text{length} (\text{stack} s) \)
shows \( \delta s \ (\text{stack} s ! i) < \delta s \ (\text{stack} s ! j) \iff i > j \) (is \( \delta s \ ?i < \delta s \ ?j \iff - \))
proof
assume \( \delta \): \( \delta s \ ?i < \delta s \ ?j \)
from \( l \) stack-set-def
\text{disc}: \( \delta s \ ?i \in \text{dom} (\text{discovered} s) \) \( \delta s \ ?j \in \text{dom} (\text{discovered} s) \)
by auto
with disc-unequal[OF disc] \( \delta \) have \( i \neq j \) by auto
moreover
\{ 
assume \( i < j \)
\}
with \( l \) have stack \( s \) ! \( j \in set (\text{drop} (\text{Suc} i) (\text{stack} s)) \)
using in-set-drop-conv-nth[of stack \( s \) ! \( j \) Suc \( i \) stack \( s \)]
by fastforce
with prev-stack-discover \( l \) have \( \delta s \ (\text{stack} s ! j) < \delta s \ (\text{stack} s ! i) \)
by simp
with \( \delta \) have \( False \) by simp
\}
ultimately show \( i > j \) by force
next
assume \( i > j \)
with \( l \) have stack \( s \) ! \( i \in set (\text{drop} (\text{Suc} j) (\text{stack} s)) \)
using in-set-drop-conv-nth[of stack \( s \) ! \( i \) Suc \( j \) stack \( s \)]
by fastforce
with prev-stack-discover \( l \) show \( \delta s \ ?i < \delta s \ ?j \) by simp
qed
end end
1.2.2 Paranthesis Theorem

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

definition parenthesis s ≡ ∀ v ∈ dom (discovered s). ∀ w ∈ dom (discovered s).
  δ s v < δ s w ∧ v ∈ dom (finished s) →
  \( \varphi s v < \delta s w \) — disjoint
  \( \vee (\varphi s v \geq \delta s w \land v \in \text{dom} (\text{finished s}) \land \varphi s w < \varphi s v) \)

lemma i-parenthesis: is-invar parenthesis

proof (induct rule: is-invarI)
  case (finish s s' )
  hence EQ [simp]: discovered s' = discovered s
  counter s' = Suc (counter s)
  finished s' = finished s(hd (stack s) → counter 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)

{ fix x y
  assume dom: x ∈ dom (discovered s) y ∈ dom (discovered s')
  and δ : δ s' x < δ s' y
  and f: x ∈ dom (finished s')
  hence neq: x ≠ y by force

  note assms = dom δ f EQ

  let ?DISJ = \( \varphi s' x < \delta s' y \)
  let ?IN = \( \delta s' y < \varphi s' x \land y \in \text{dom} (\text{finished s'}) \land \varphi s' y < \varphi s' x \)

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

    have δ s y < \( \varphi s' x \land y \in \text{dom} (\text{finished s}) \land \varphi s y < \varphi s' x \)
    proof (cases y ∈ set (stack s))
      — y on stack is not possible: According to
      δ s' x < \( \delta s' y \)
      it is discovered after x (= hd (stack s))
    case True with y-not-hd have y ∈ set (tl (stack s))
      by (cases stack s simp-all)
    with tl-tl-stack-hd-discover[OF NE] δ x-is-hd have δ s y < δ s x
      by simp
    with δ have False by simp

  33
thus \( \text{thesis} \).

next

case False — y must be a successor of \( x \) (= (hd (stack s)))

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

with False discovered-not-stack-imp-finished have *:

\( y \in \text{dom} \) (finished s)

by simp

moreover with timing-less-counter \( \varphi x \) have \( \varphi s \ y < \varphi s' \ x \) by simp

moreover with * disc-lt-fin \( \varphi x \) have \( \delta s \ y < \varphi s' \ x \)

by (metis less-trans)

ultimately show \( \text{thesis} \) by simp

qed

with y-not-hd show \( \text{thesis} \) by simp

next

case False note [simp] = this

show \( \text{thesis} \)

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

case False with finish assms show \( \text{thesis} \)

by (simp add: parenthesis-def)

next

case True with stack-not-finished have \( y \notin \text{dom} \) (finished s)

using hd-in-set[OF NE]

by auto

with finish assms have \( \varphi s \ x < \delta s \ y \)

unfolding parenthesis-def

by auto

hence \( \text{DISJ} \) by simp

thus \( \text{thesis} \) ..

qed

qed

}

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

next

case (discover s s' u v)

hence EQ[simp]: discovered \( s' = \) (discovered s)(v \( \mapsto \) counter s)

\( \text{finished} \ s' = \text{finished} \ s \)

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

by simp-all

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

from discover finished-discovered have

\( V' : v \notin \text{dom} \) (discovered s) \( v \notin \text{dom} \) (finished s)

by auto

{

fix \( x y \)

assume dom: \( x \in \text{dom} \) (discovered s') \( y \in \text{dom} \) (discovered s')

and \( \delta: \delta s' x < \delta s' y \)

and \( f: x \in \text{dom} \) (finished s')


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

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

lemma parenthesis-disjoint:
  assumes \( v \in \text{dom} \ (\text{finished} \ s) \) \( w \in \text{dom} \ (\text{discovered} \ s) \)
and $\delta s v < \delta s w \quad \varphi s w > \varphi s v$
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 $\text{edges } s \equiv \text{tree-edges } s \cup \text{cross-edges } s \cup \text{back-edges } s$

lemma is-invar $(\lambda s. \text{finite } (\text{edges } s))$
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 $x \in \text{tree-edges } s \Longrightarrow P$
and $x \in \text{cross-edges } s \cup \text{back-edges } s \Longrightarrow P$
shows $P$
using assms by auto

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

lemma $i$-edges-basic:
is-invar edges-basic
unfolding edges-basic-def[abs-def]
proof (induct rule: is-invarI-full)
\begin{enumerate}
\item 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:
  \(\text{i-invar} (\lambda s. (E \cap \text{dom} (\text{discovered} s) \times \text{UNIV}) \setminus \text{pending} s = \text{edges} s)\)
proof (induction rule: i-invarI-full)
case (new-root s s' v0)
interpret DFS-invar G param s by fact
from new-root empty-stack-imp-empty-pending
have [simp]: pending s = {} by simp
from \(v0 \notin \text{dom} (\text{discovered} s)\):
have [simp]: \(E \cap \text{insert} v0 (\text{dom} (\text{discovered} s)) \times \text{UNIV} \setminus \{v0\} \times \text{succ} v0 = E \cap \text{dom} (\text{discovered} s) \times \text{UNIV}\) 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
  \(E \cap \text{dom} (\text{discovered} s) \times \text{UNIV} \setminus (\text{pending} s \setminus \{(\text{hd} (\text{stack} s), v)\}) = \text{insert} (\text{hd} (\text{stack} s), v) (E \cap \text{dom} (\text{discovered} s) \times \text{UNIV} \setminus \text{pending} s)\)
  by auto
thus ?case using cross-edge by simp
next
case (back-edge s s' u v)
interpret DFS-invar G param s by fact

from back-edge stack-discovered have u ∈ dom (discovered s) by (cases stack s) auto

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

thus ?case using back-edge by simp

next
  case (discover s s' u v)
  interpret DFS-invar G param s by fact

from discover stack-discovered have u ∈ dom (discovered s) by (cases stack s) auto

with discover(2−) pending-ssE have
  E ∩ insert v (dom (discovered s)) × UNIV
  − (pending s − {(hd (stack s), v)} ∪ {v} × succ v)
  = insert (hd (stack s), v) (E ∩ dom (discovered s) × UNIV − pending s)
by auto

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

context DFS-invar begin

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

lemma edges-ss-reachable-edges:
  edges s ⊆ E ∩ reachable × UNIV
using edges-covered discovered-reachable
by (fast intro: rtrancl-image-advance-rtrancl)

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

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

tree-edges-ssE: tree-edges s ⊆ E and
tree-edges-not-pending: tree-edges s ⊆ − pending s and
tree-edge-is-succ: (v, w) ∈ tree-edges s ⇒ w ∈ succ v and
tree-edges-discovered: Field (tree-edges s) ⊆ dom (discovered s) and
cross-edges-ssE: cross-edges s ⊆ E and
cross-edges-not-pending: cross-edges s ⊆ − pending s and
cross-edge-is-succ: (v, w) ∈ cross-edges s ⇒ w ∈ succ v and
cross-edges-discovered: Field (cross-edges s) ⊆ dom (discovered s) and
back-edges-ssE: back-edges s ⊆ E and
back-edges-not-pending: back-edges s ⊆ − pending s and
back-edge-is-succ: (v, w) ∈ back-edges s ⇒ w ∈ succ v and
back-edges-discovered: Field (back-edges s) ⊆ dom (discovered s) and
using edges-basic
unfolding edges-basic-def
by auto

lemma edges-disjoint:
back-edges s ∩ cross-edges s = {}
back-edges s ∩ tree-edges s = {}
cross-edges s ∩ tree-edges s = {}
using edges-basic
unfolding edges-basic-def
by auto

lemma tree-edge-imp-discovered:
(v, w) ∈ tree-edges s ⇒ v ∈ dom (discovered s)
(v, w) ∈ tree-edges s ⇒ w ∈ dom (discovered s)
using tree-edges-discovered
by (auto simp add: Field-def)

lemma back-edge-imp-discovered:
(v, w) ∈ back-edges s ⇒ v ∈ dom (discovered s)
(v, w) ∈ back-edges s ⇒ w ∈ dom (discovered s)
using back-edges-discovered
by (auto simp add: Field-def)

lemma cross-edge-imp-discovered:
(v, w) ∈ cross-edges s ⇒ v ∈ dom (discovered s)
(v, w) ∈ cross-edges s ⇒ w ∈ dom (discovered s)
using cross-edges-discovered
by (auto simp add: Field-def)

lemma edge-imp-discovered:
(v, w) ∈ edges s ⇒ v ∈ dom (discovered s)
\((v, w) \in \text{edges } s \implies w \in \text{dom (discovered } s)\)

**using** tree-edge-imp-discovered cross-edge-imp-discovered back-edge-imp-discovered

**by** blast+

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

**using** finite-subset[OF tree-edges-discovered discovered-finite] **by** simp

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

**using** finite-subset[OF cross-edges-discovered discovered-finite] **by** simp

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

**using** finite-subset[OF back-edges-discovered discovered-finite] **by** simp

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

**by** auto

end

**Properties of the DFS Tree**

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

**lemma** tree-edge-disc-lt-fin:

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

**by** (metis finished-succ-fin tree-edge-is-succ)

**lemma** back-edge-disc-lt-fin:

\((v, w) \in \text{back-edges } s \implies v \in \text{dom } \text{(finished } s) \implies \delta s w < \varphi s v\)

**by** (metis finished-succ-fin back-edge-is-succ)

**lemma** cross-edge-disc-lt-fin:

\((v, w) \in \text{cross-edges } s \implies v \in \text{dom } \text{(finished } s) \implies \delta s w < \varphi s v\)

**by** (metis finished-succ-fin cross-edge-is-succ)

end

context **param-DFS begin**

**lemma** i-stack-is-tree-path:

is-invar \((\lambda s. \text{stack } s \not= [] \implies (\exists v0 \in V0. \text{path (tree-edges } s) v0 (\text{rev (tl (stack } s))) (\text{hd (stack } s))))\)

**proof** (induct rule: is-invarI)

**case** (discover s s' u v)

**hence** EQ[simp]: stack s' = v \# stack s

**by** simp-all

**from** discover have NE[simp]: stack s \not= [] **by** simp

**from** discover obtain v0 \text{where}
\( v_0 \in V_0 \)

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

by blast

with path-mono[OF - this(2)] EQ have

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

by blast

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

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

next

case (finish \( s \ \_)')

hence EQ[simp]: \( \text{stack } s' = \text{tl } (\text{stack } s) \)

\[
\text{tree-edges } s' = \text{tree-edges } s
\]

by simp-all

from finish obtain \( v_0 \) where

\( v_0 \in V_0 \)

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

by blast

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

show ?case

proof

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

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

by (auto simp: neq-Nil-conv path-simps)

moreover from \( P \ A \) have

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

by (simp)

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

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

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

qed

qed simp-all

end

context DFS-invar begin

lemmas stack-is-tree-path =

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

lemma stack-is-path:

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

using stack-is-tree-path path-mono[OF tree-edges-ssE]

by blast

lemma hd-succ-stack-is-path:

assumes ne: \( \text{stack } s \neq [] \)
and succ: $v \in \text{succ}(\text{hd}(\text{stack} s))$
shows $\exists v_0 \in V_0. \text{path} E v_0 (\text{rev}(\text{stack} s)) v$
proof 
from stack-is-path[OF ne] succ obtain $v_0$ where $v_0 \in V_0$
path $E v_0 (\text{rev}(\text{tl}(\text{stack} s))) @ [\text{hd}(\text{stack} s)]) v$
by (auto simp add: path-append-cone)
thus ?thesis using ne
by (cases stack s) auto
qed

lemma tl-stack-hd-tree-path:
assumes 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 $(\text{tree-edges} s) v_0 (\text{rev}(\text{tl}(\text{stack} s))) (\text{hd}(\text{stack} s))$
by auto
from assms path-member-reach-end[OF this] show ?thesis by simp
qed

end
context param-DFS begin

definition tree-discovered-inv $s \equiv$
$(\text{tree-edges} s = \{} \longrightarrow \text{dom}(\text{discovered} s) \subseteq V_0 \land (\text{stack} s = [])$
$\lor (\exists v_0 \in V_0, \text{stack} s = [v_0] ))$
$\land (\text{tree-edges} s \neq \{} \longrightarrow (\text{tree-edges} s)^+ \land \text{V} \cup \text{V} = \text{dom} (\text{discovered} s) \cup V_0)$

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 $s' = v \neq \text{stack} s$

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 NE[simp]: stack $s \neq []$ by simp
note TDI = (tree-discovered-inv $s$ [unfolded tree-discovered-inv-def])

have tree-edges $s' = \{} \longrightarrow \text{dom}(\text{discovered} s') \subseteq V_0 \land (\text{stack} s' = [] \lor (\exists v_0 \in V_0, \text{stack} s' = [v_0] ))$
by simp -- tree-edges $s' \neq \{}$
moreover 
\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
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')\+ y \notin V0
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 \neq 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

\note t-d = this

\{ 
\fix x
\assume x ∈ dom (discovered s') \∪ V0 x \notin V0
\hence A: x ∈ dom (discovered s') by simp

have x ∈ (tree-edges s')\+ \̸∈ V0 \∪ V0
proof (cases tree-edges s = {})
case True with trancl-single have (tree-edges s')\+ = {(hd (stack s), v)}
by simp
moreover from True TDI have hd (stack s) ∈ V0 dom (discovered s) \subseteq V0 by auto
ultimately show thesis using A \notin V0 by auto
next
case False note t-ne = this

show thesis
proof (cases x=v)
case False with A have x ∈ dom (discovered s) by simp
with TDI t-ne \( x \notin V0 \) have \( x \in (\text{tree-edges } s)^+ \) \( V0 \) by auto

with trancl-sub-insert-trancl show \(?thesis\) by simp blast

next

case True

from t-ne TDI have \( \text{dom } (\text{discovered } s) \cup V0 = (\text{tree-edges } s)^+ \) \( V0 \)

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 \( (v0, \text{hd } (\text{stack } s)) \in (\text{tree-edges } s')^* \) by (auto intro: rtrancl-mono-mp)

ultimately show \(?thesis\) using \( (v0 \in V0) \) True by (auto elim: rtrancl-into-trancl1)

qed

ultimately show \(?case\) by (simp add: tree-discovered-inv-def)

qed (auto simp add: tree-discovered-inv-def)

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

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

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

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

using tree-discovered-inv

by (auto simp add: tree-discovered-inv-def)

lemma i-tree-one-predecessor:

is-invar \((\lambda s. \forall (v,v') \in \text{tree-edges } s. \forall y. y \neq v \implies (y,v') \notin \text{tree-edges } s)\)

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 \( v-hd: \text{hd } (\text{stack } s) \neq v \)

using hd-in-set[of NE] by blast

from discover tree-edges-discovered have \( v-notin-tree: \forall (x,x') \in \text{tree-edges } s. x \neq v \land x' \neq v \)

\}

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

thus ?case by blast
qed simp-all

lemma (in DFS-invar) tree-one-predecessor:
  assumes \((v,w) \in \text{tree-edges } s\)
  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 (in DFS-invar) 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 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 (hd (stack } s), v) \text{ (tree-edges } 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 ≠ [] by (simp add: cond-alt)

from discover tree-edges-discovered have
v-notin-tree: ∀(x,x') ∈ tree-edges s. x ≠ v ∧ x' ≠ v
by (blast intro: Field-not-elem)
from discover stack-discovered have
v-hd: hd (stack s) ≠ v
using hd-in-set[OF NE]
by blast

{ fix a b
  assume T; (a,b) ∈ tree-edges s'
  have δ s' a < δ s' b
  proof (cases b = v)
    case True with T v-notin-tree have [simp]: a = hd (stack s) by auto
    with stack-discovered have a ∈ dom (discovered s)
    by (metis hd-in-set NE subsetD)
    with v-hd True timing-less-counter show ?thesis by simp
  next
    case False with v-notin-tree T have (a,b) ∈ tree-edges s a ≠ v by auto
    with discover have δ s a < δ s b by auto
    with False ⟨a≠v⟩ show ?thesis by simp
  qed
} thus ?case by blast
next
  case (new-root s s' v0)
  then interpret DFS-invar where s=s by simp
  from new-root have tree-edges s' = tree-edges s by simp
  moreover from tree-edge-imp-discovered new-root have ∀(v,v') ∈ tree-edges
    s. v ≠ v0 ∧ v' ≠ v0 by blast
  ultimately show ?case using new-root by auto
  qed simp-all
end
end

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

lemma tree-edge-disc:
(v,w) ∈ tree-edges s ⟹ δ s v < δ s w
using i-tree-edge-disc[THEN make-invar-thm]
by blast

lemma tree-path-disc:
(v,w) ∈ (tree-edges s)⁺ ⟹ δ s v < δ s w
by (auto elim!: trancl-induct dest: tree-edge-disc)

lemma no-loop-in-tree:
(v,v) /∈ (tree-edges s)⁺
using tree-path-disc by auto

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

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

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

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

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

from y obtain py where py: path (tree-edges s) y py v
  using trancl-is-path by metis

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

47
using tree-eq-rule
by auto
with path-is-trancl \( as \neq [] \): show ?case by metis
next
case (snoclr a as b bs) hence
path (tree-edges s) x as a (a,v) \( \in \) tree-edges s
path (tree-edges s) y bs b (b,v) \( \in \) tree-edges s
by (simp-all add: path-simps)
moreover hence a=b using tree-eq-rule by simp
ultimately show ?thesis using snoclr.IH by metis
qed
qed

lemma tree-subpath:
assumes \((x,v) \in (\text{tree-edges } s)^+\)
and \((y,v) \in (\text{tree-edges } s)^+\)
and \(\delta; \delta s x < \delta s y\)
shows \((x,y) \in (\text{tree-edges } s)^+\)
proof –
from \(\delta\) have \(x \neq y\) by auto
with assms tree-subpath have \((x,y) \in (\text{tree-edges } s)^+ \lor (y,x) \in (\text{tree-edges } s)^+\)
by simp
moreover from \(\delta\) tree-path-disc have \((y,x) \notin (\text{tree-edges } s)^+\) by force
ultimately show ?thesis by simp
qed

lemma on-stack-is-tree-path:
assumes \(x; x \in \text{set } (\text{stack } s)\)
and \(y; y \in \text{set } (\text{stack } s)\)
and \(\delta; \delta s x < \delta s y\)
shows \((x,y) \in (\text{tree-edges } s)^+\)
proof –
from \(x\) obtain \(i\) where \(i; \text{stack } s ! i = x i < \text{length } (\text{stack } s)\)
by (metis in-set-conv-nth)
from \(y\) obtain \(j\) where \(j; \text{stack } s ! j = y j < \text{length } (\text{stack } s)\)
by (metis in-set-conv-nth)
with \(i \delta\) stack-nth-order have \(j < i\) by force
from \(x\) have ne[simp]: \(\text{stack } s \neq []\) by auto
from \(j<i\) have \(x \in \text{set } (\text{tl } (\text{stack } s))\)
using nth-mem nth-tl[OF ne, of \(i-1\)] \(i\)
by auto
with il-stack-hd-tree-path have \(x\)-path: \((x, \text{hd } (\text{stack } s)) \in (\text{tree-edges } s)^+\)
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 \(y \in set \ (tl \ (stack \ s))\)
using \(nth\)-mem \(nth\)-tl[\(OF \ ne, \ of \ j - 1\)] \(j\)
by auto
with \(tl\)-stack-hd-tree-path have \((y, \ hd \ (stack \ s)) \in (tree\edges \ s)^+\)
by simp
with \(x\)-path \(\delta\) show \(?thesis\)
using \(tree\)-subpath
by metis
qed

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

lemma tree-edge-impl-parenthesis:
assumes \(t: (v, w) \in tree\edges \ s\)
and \(f: v \in dom \ (finished \ s)\)
shows \(w \in dom \ (finished \ s)\)
\(\land \delta \ s \ v < \delta \ s \ w\)
\(\land \varphi \ s \ w < \varphi \ s \ v\)
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

49
lemma tree-path-impl-parenthesis:
assumes \((v, w) \in (\text{tree-edges } s)^+\)
and \(v \in \text{dom } (\text{finished } s)\)
shows \(w \in \text{dom } (\text{finished } s)\)
\(\land \delta s v < \delta s w\)
\(\land \phi s w < \phi s v\)
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 \(v0 \text{ where } v0 \in V0\)
\(\land \delta s v0 < \delta s v \land \phi s v < \phi s v0\)
proof –
from nc-discovered-eq-reachable[of \(C\)] discovered-iff-tree-path v
obtain v0 where \(v0 \in V0\) and
(v0,v) \((\text{tree-edges } s)^+\)
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 \(s \equiv \forall v \in \text{dom } (\text{discovered } s). \forall w \in \text{dom } (\text{finished } s).\)
\(\delta s v < \delta s w \land (v \notin \text{dom } (\text{finished } s) \lor \phi s v > \phi s w)\)
\(\rightarrow (v, w) \in (\text{tree-edges } s)^+\)

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]: \(\text{tree-edges } s’ = \text{insert } (hd (\text{stack } s), v) (\text{tree-edges } s)\)
  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]: \(\text{stack } s \neq []\) by (simp add: cond-alt)
show \(?case\)

50
proof (intro ballI impI)
  fix a b
  assume F: a ∈ dom (discovered s') b ∈ dom (finished s')
  and D: δ s' a < δ s' b ∧ (a ∉ dom (finished s') ∨ ϕ s' a > ϕ s' b)

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

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

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

show (a,b) ∈ (tree-edges s')+
proof (cases b = hd (stack s))
  case True hence ∃b: 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⟩ ⟨b 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

case False note b-not-hd = this

show \( ?\text{thesis} \)

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

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

next

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

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

by simp-all

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

by simp

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

qed

qed

qed

next

case (new-root \( \text{s} \) \( \text{s'} \) \( \text{v0} \)) then interpret DFS-invar where \( \text{s=}\text{s} \) by simp

from new-root finished-discovered have \( \text{v0} \notin \text{dom} \) (finished \( \text{s} \)) by auto

moreover note timing-less-counter finished-discovered

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

qed simp-all

end end

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

lemmas s-paren-imp-tree-reach =

paren-imp-tree-reach[THEN make-invar-thm]

lemma parenthesis-impl-tree-path-not-finished:

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

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

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

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

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

using s-paren-imp-tree-reach assms

by (auto simp add: paren-imp-tree-reach-def)

lemma parenthesis-impl-tree-path:

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

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

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

proof =

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

using finished-discovered by blast

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

using s-paren-imp-tree-reach assms

by (auto simp add: paren-imp-tree-reach-def)

qed

52
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 $\delta s w < \varphi s w \in \text{dom}(\text{discovered } s)$
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 $v$ $\delta$ 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

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

Properties of Cross Edges
context param-DFS begin context begin interpretation timing-syntax .

lemma i-cross-edges-finished: is-invar $(\lambda s. \forall (u, v) \in \text{cross-edges } s)$.

53
\( v \in \text{dom} (\text{finished } s) \land (u \in \text{dom} (\text{finished } s) \rightarrow \varphi s v < \varphi s u) \)

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

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

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

**qed** simp-all

end end

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

lemmas cross-edges-finished
  = i-cross-edges-finished[THEN make-invar-thm]

lemma cross-edges-target-finished:
  \((u,v)\in \text{cross-edges } s \Rightarrow v \in \text{dom} (\text{finished } s)\)
  using cross-edges-finished by auto

lemma cross-edges-finished-decr:
  \[((u,v)\in \text{cross-edges } s; u\in \text{dom} (\text{finished } s)\] \Rightarrow \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 \(?\text{thesis}\)
    proof (cases \(v \in \text{dom} (\text{finished } s)\))
      case True with cross-edges-finished-decr[OF cross]
      show \(?\text{thesis by force}\)
    next
      case False with w-fin show \(?\text{thesis by force}\)
    qed
  qed
end end

Properties of Back Edges

context \(\text{param-DFS}\) 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)

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

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

using \(\text{stack-set-def}\)
by auto

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

end end

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

**lemma** \(\text{back-edge-impl-tree-path}:\)

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

using \(i\text{-back-edge-impl-tree-path}[\text{THEN make-invar-thm}]\)
by blast

**lemma** \(\text{back-edge-disc}:\)

assumes \((v, w) \in \text{back-edges } s\)
shows \(\delta s w \leq \delta s v\)
proof cases
assume \(v \neq w\)
with \(\text{assms back-edge-impl-tree-path}\) have \((w, v) \in (\text{tree-edges } s)^+\) by simp
with \(\text{tree-path-disc}\) show \(?\text{thesis}\) by force
qed simp

**lemma** \(\text{back-edges-tree-disjoint}:\)

\(\text{back-edges } s \cap \text{tree-edges } s = \{\}\)
using \(\text{back-edge-disc}\) \(\text{tree-edge-disc}\)
by force

**lemma** \(\text{back-edges-tree-pathes-disjoint}:\)

\(\text{back-edges } s \cap (\text{tree-edges } s)^+ = \{\}\)
using \(\text{back-edge-disc}\) \(\text{tree-path-disc}\)
by force

**lemma** \(\text{back-edge-finished}:\)

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

context param-DFS begin context
interpretation timing-syntax
.

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

fix \( w \)
assume w: \( w \in \text{succ } (\text{hd } (\text{stack } s)) \) \( w \neq \text{hd } (\text{stack } s) \)
\( w \in \text{dom } (\text{discovered } s) \)
and $f: w \in \text{dom} (\text{finished } s) \rightarrow \text{counter } s \leq \varphi s w$
and $\delta: \delta s w \leq \delta s (\text{hd} (\text{stack } s))$

with timing-less-counter have $w \notin \text{dom} (\text{finished } s)$ by force
with finish $w \not\in \text{disc} (\text{hd} (\text{stack } s), w) \in \text{back-edges } s$ by blast

moreover

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

ultimately show ?case
using finish
by clarsimp auto

next

\text{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

\text{case} (\text{new-root } s s' v0) then interpret DFS-invar where $s=s$ by simp

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

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

**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** \(*: u \in \text{dom } (\text{finished } s) \) and \( (u,v) \in ?E^+ \)

from this(2) have \( \varphi s v < \varphi s u \land v \in \text{dom } (\text{finished } s) \)

**proof** induct

**case** base thus \(?case\)

by (metis Un-iff * cross-edges-finished-decr cross-edges-target-finished tree-edge-impl-parenthesis)

**next**

**case** (step \( v \) \( w \))

**hence** \( \varphi s w < \varphi s v \land w \in \text{dom } (\text{finished } s) \)

by (metis Un-iff cross-edges-finished-decr cross-edges-target-finished tree-edge-impl-parenthesis)

**with** step show \(?case\) **by** auto

**qed**

\} note aux = this

**assume** \( \neg \text{acyclic } ?E \)

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

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 .

59
lemma back-edge-closes-reachable-cycle:
back-edges s \neq \{\} \implies \neg \text{acyclic} (E \cap \text{reachable} \times \text{UNIV})
by (metis back-edge-closes-cycle edges-ss-reachable-edges cyclic-subset)

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

end

1.2.4 White Path Theorem
context DFS begin
context begin interpretation timing-syntax .
definition white-path where
white-path s x y \equiv x \neq y 
\rightarrow (\exists p. \text{path} E \ x \ p \ y \wedge
(\delta s x < \delta s y \wedge (\forall v \in \text{set} (tl p). \delta s x < \delta s v)))

lemma white-path:
it-dfs \leq \text{SPEC}(\lambda s. \forall x \in \text{reachable}. \forall y \in \text{reachable}. \neg \text{is-break} \ \text{param} \ s \implies
white-path s x y \iff (x,y) \in (\text{tree-edges} s)^*)
proof (rule it-dfs-SPEC, intro ballI impI)
fix s x y
assume DI: DFS-invar G \text{param} s
and C: \neg \text{cond} s \neg \text{is-break} \ \text{param} s
and reach: x \in \text{reachable} y \in \text{reachable}
from DI interpret DFS-invar where s=s .

note fin-eq-reach = nc-finished-eq-reachable[\text{OF} C]

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

show \:?thesis
proof
assume (x,y) \in (\text{tree-edges} s)^*
with (x\neq y) have T: (x,y) \in (\text{tree-edges} s)^+ by (metis \text{rtranclD})
then obtain p where P: \text{path} (\text{tree-edges} s) x p y by (metis \text{trancel-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 \( \delta \ s \ x < \delta \ s \ y \land (\forall \ v \in \text{set} \ tl \ p). \ \delta \ s \ x < \delta \ s \ v \)
using \( \langle x \neq y \rangle \)
proof (induct rule: path-\( tl \)-induct)
case (\underline{single} u) thus ?case by (fact tree-edge-disc)
next
case (\underline{step} u v) note \( \delta \ s \ x < \delta \ s \ w \)
also from \underline{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: \( \langle x \neq y \rangle \) white-path-def)
next
assume \( \text{white-path} \ s \ x \ y \)
with \( \langle x \neq y \rangle \) obtain \( p \) where
\( P: \text{path} \ E \ s \ x \ y \) and
\( \text{white}: \delta \ s \ x < \delta \ s \ y \land (\forall \ v \in \text{set} \ tl \ p). \ \delta \ s \ x < \delta \ s \ v \)
unfolding \( \text{white-path-def} \)
by blast
hence \( p \neq [] \) by auto
thus \( (x,y) \in (\text{tree-edges} \ s)\) using \( P \) \( \text{white} \) \( \text{reach} \) (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 \( \text{reach} \) single show ?case
using \( \text{fin-eq-reach} \) \( \text{finished-su}c\text{c-im}l\text{-path-in-tree[of } x \ y \] \by simp
next
case (snoc \( u \) \( u s \)) hence \( \text{path} \ E \ s \ x \ u \) by (simp add: path-append-conv)
mwmoreover hence \( (x,u) \in E^* \) by (simp add: path-is-rtrancl)
with \( \text{reach} \) \( \text{ure}c\text{ach}: \ u \in \text{reachable} \)
by (metis rtrancl-image-advance-rtrancl)
mwmoreover from snoc have \( \delta \ s \ x < \delta \ s \ u \ (\forall \ v \in \text{set} \ tl \ us). \ \delta \ s \ x < \delta \ s \ v \)
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(\( 5 \)) fin-eq-reach \( \text{finished-di}sc\text{over} \) have
\( y-f-d: y \in \text{dom} (\text{finished} \ s) \) \( y \in \text{dom} (\text{discovered} \ s) \)
by auto

from \( y \in \text{succ} \ u \) \( \text{ure}c\text{ach} \) fin-eq-reach have \( \delta \ s \ y < \varphi \ s \ u \)
using finished-su\text{c}c-fin by simp
also from \( \delta \ s \ x < \delta \ s \ w \) \underline{have} \( x \neq u \) by auto
with \( x-u \) have \( (x,u) \in (\text{tree-edges} \ s)^* \) by (metis rtrancl-eq-or-trancl)
with fin-eq-reach have \( \varphi \ s \ u < \varphi \ s \ x \)
using tree-path-impl-parenthesis
by simp
finally have \( \varphi \ s \ y < \varphi \ s \ x \)
using reach fin-eq-reach y-f-d snoc
using parenthesis-contained
by blast

hence \((x,y) \in (\text{tree-edges } s)^+\)
using reach fin-eq-reach y-f-d snoc
using parenthesis-impl-tree-path
by blast

thus \(?\text{case by auto}\)
qed
qed
qed
qed
end end

end

1.3 Invariants for SCCs

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

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

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

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

definition \(scc\text{-}\text{roots} s = \{v. \exists scc. scc\text{-}\text{root} s v scc\}\)
end
countext DFS-invar begin
lemma scc-root-is-discovered:
\[ scc-root \ s \ v \ scc \implies v \in \text{dom} \ (\text{discovered} \ s) \]
bysimp (simp add: scc-root-def)

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

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

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

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

lemma scc-root-unique-root:
\[ \text{assumes} \ scc1: \ scc-root \ s \ v \ scc \]
and \[ scc2: \ scc-root \ s \ v' \ scc \]
shows \( v = v' \)
proof (rule ccontr)
assume \( v \neq v' \)
from \( \text{scc1} \) have \( v \in scc \ v \in \text{dom} \ (\text{discovered} \ s) \)
by (simp-all add: scc-root-def)
with $scc-root-scc-tree-trancl[\text{OF } scc2]$ \(v \neq v'\) 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 $scc-root-scc-tree-trancl[\text{OF } scc1]$ \(v \neq v'\) 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 $scc-root s v (scc-of E v)$
proof –
from assms have \(v \in scc\) is-scc $E scc$ by (simp-all add: $scc-root-def$)
moreover have \(v \in scc\) $E v$ is-scc $E$ (scc-of $E v$) by simp-all
ultimately have \(scc = scc\)-of $E v$ using is-scc-unique by metis
thus ?thesis using assms by simp
qed

lemma $scc-root-finished-impl-scc-finished$:
assumes \(v \in \text{dom (finished } s)\)
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 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: is-scc-connected')
with rtrancl-is-path obtain \(p\) where path \(?E v p x\) bymetis
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 (snoc \(y\) \(ys\)) hence path \(?E v ys y\) \((y,x) \in ?E)\)
by (simp-all add: path-append-conv)
with snoc.IH have \(y \in \text{dom (finished } s)\) by simp
moreover from \((y,x) \in ?E; \text{have } (y,x) \in E x \in scc\) 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
qed

context begin interpretation timing-syntax .

**lemma scc-root-disc-le:**

**assumes** scc-root s v scc
and \( x \in \text{scc} \) \( x \in \text{dom} \text{(discovered s)} \)
**shows** \( \delta \text{s v} \leq \delta \text{s x} \)

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

qed

**lemma scc-root-fin-ge:**

**assumes** scc-root s v scc
and \( v \in \text{dom} \text{(finished s)} \)
and \( x \in \text{scc} \)
**shows** \( \varphi \text{s v} \geq \varphi \text{s x} \)

**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

**lemma scc-root-is-Min-disc:**

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

**proof**
- **rule** Min-eqI
  - **from** discovered-finite **show** finite \( ?S \text{ by} \) auto
  - **from** scc-root-disc-le[\( \text{OF assms} \)] **show** \( \forall y . \ y \in \ ?S \implies \delta \text{s v} \leq y \text{ by} \) force

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

qed

**lemma Min-disc-is-scc-root:**

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

**proof**
- {\( \text{fix} y \)
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 auto

with assms disc-unequal $A$ have $\delta \ s \ v < \delta \ s \ y$ by fastforce

} note scc-disc = this

{ fix $x$

assume $A$: $x \in \text{dom} (\text{discovered} \ s)$$

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

proof (cases $v = x$)

  case False with $A$ scc-disc have $\delta$ $\delta \ s \ v < \delta \ s \ x$ by simp

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

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

  case False with $A$ stack-set-def assms have $v$-stack: $v \in \text{set} (\text{stack} \ s)$ by auto

  show ?thesis
  
  proof (cases $x \in \text{dom} (\text{finished} \ s)$)

    case True with parenthesis-impl-tree-path-not-finished[of $v$ $x$] assms $\delta$ False

    show ?thesis by auto

  next

    case False with $A$ stack-set-def have $x \in \text{set} (\text{stack} \ s)$ by auto

    with $v$-stack $\delta$

    show ?thesis by simp

  qed

next

  case True note $v$-fin = this

let $?E = \text{Restr} \ E \text{ scc}$

{ fix $y$

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

hence $s$: $y \in \text{succ} v$ $y \in \text{scc}$ by auto

with finished-impl-succ-discovered $v$-fin have $y \in \text{dom} (\text{discovered} \ s)$ by simp

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

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

by simp

} note trancl-base = this

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

with assms have $(v,x) \in ?E^+$

by (simp add: is-scc-connected)

with $(v \neq x)$ have $(v,x) \in ?E^+$ by (metis rtrancl-eq-or-trancl)

thus ?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) ∈ (tree-edges s)⁺ by simp
with tree-path-impl-parenthesis[OF - v-fin] have
y-fin: y ∈ dom (finished s)
and y-t: δ s v < δ s y ϕ s y < ϕ s v
by auto
with finished-discovered have y-disc: y ∈ dom (discovered s)
by auto
from step have #: z ∈ succ y z ∈ see by auto
with finished-imp-succ-discovered y-fin have
z-disc: z ∈ dom (discovered s) by simp
with #: (v≠z) have δz: δ s v < δ s z by (simp add: scc-disc)

from #: edges-covered finished-no-pending[OF #: y ∈ dom (finished s)]
y-disc have (y,z) ∈ edges s by auto
thus ?thesis
proof safe
assume (y,z) ∈ tree-edges s with #: (v,y) ∈ (tree-edges s)⁺ show
?thesis ..
next
assume CE: (y,z) ∈ cross-edges s
with cross-edges-finished-decr y-fin y-t have ϕ s z < ϕ s v
by force
moreover note δz
moreover from CE cross-edges-target-finished have
z ∈ dom (finished s) by simp
ultimately show ?thesis
using parenthesis-impl-tree-path[OF v-fin] by metis
next
assume BE: (y,z) ∈ back-edges s
with back-edge-disc-lt-fin y-fin y-t have
δ s z < ϕ s v by force
moreover note δz
moreover note z-disc
ultimately have z ∈ dom (finished s) ϕ s z < ϕ s v
using parenthesis-contained[OF v-fin] by simp-all
with δz show ?thesis
using parenthesis-impl-tree-path[OF v-fin] by metis
qed
qed

67
thus \( \text{thesis} \) by auto

qed simp

\}

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

with \text{assms} show \( \text{thesis} \) by (auto intro: \text{scc-rootI})

qed

\begin{lemma}
\text{scc-root-iff-Min-disc:}
\begin{assumes}
\text{is-scc E scc r} \\
\text{scc} r \in \text{dom} (\text{discovered s})
\end{assumes}
\text{shows} \( \text{scc-root s r scc \leftarrow\rightarrow \text{Min} (\delta (s \cdot (\text{scc} \cap \text{dom} (\text{discovered s}))))} = \delta s r (\text{is L} \leftarrow\rightarrow \text{R}) \)
\end{lemma}

\begin{proof}
\begin{assumes}
\text{L} \leftarrow\rightarrow \text{R} \\
\end{assumes}
\text{assume L with scc-root-is-Min-disc show R .}
\end{proof}

\begin{next}
\begin{assumes}
\text{R} \leftarrow\rightarrow \text{L} \\
\end{assumes}
\text{assume R with Min-disc-is-scc-root \text{assms} show L by simp}
\end{next}

\begin{proof}
\begin{assumes}
\text{S} = \text{scc} \cap \text{dom} (\text{discovered s})
\end{assumes}
\text{from \text{discovered-finite} have finite (\delta s\text{S}) by auto}
\text{moreover from \text{scc} have \delta s \cdot \text{S} \neq \{\} by auto}
\text{moreover have } \forall (x::nat) \exists A. \ x \notin A \lor (\exists y. \ x = f y \land y \in A) \text{ by blast}
\text{— autogenerated by sledgehammer}
\text{ultimately have } \exists x \in \text{S}. \ \delta s x = \text{Min} (\delta (s \cdot \text{S})) \text{ by (metis Min-in)}
\text{with \text{Min-disc-is-scc-root} (is-scc E scc) show thesis by auto}
\end{proof}

\begin{lemma}
\text{scc-root-exists:}
\begin{assumes}
\text{is-scc E scc} \\
\text{and scc: scc} \cap \text{dom} (\text{discovered s}) \neq \{\}
\end{assumes}
\text{shows } \exists r. \ \text{scc-root s r scc}
\end{lemma}

\begin{proof}
\begin{assumes}
\text{S} = \text{scc} \cap \text{dom} (\text{discovered s})
\end{assumes}
\text{from \text{discovered-finite} have finite (\delta s\text{S}) by auto}
\text{moreover from \text{scc} have \delta s \cdot \text{S} \neq \{\} by auto}
\text{moreover have } \forall (x::nat) \exists A. \ x \notin A \lor (\exists y. \ x = f y \land y \in A) \text{ by blast}
\text{— autogenerated by sledgehammer}
\text{ultimately have } \exists x \in \text{S}. \ \delta s x = \text{Min} (\delta (s \cdot \text{S})) \text{ by (metis Min-in)}
\text{with \text{Min-disc-is-scc-root} (is-scc E scc) show thesis by auto}
\end{proof}

\begin{lemma}
\text{scc-root-of-node-exists:}
\begin{assumes}
\text{v} \in \text{dom} (\text{discovered s})
\end{assumes}
\text{shows } \exists r. \ \text{scc-root s r (scc-of E v)}
\end{lemma}

\begin{proof}
\begin{assumes}
\text{E v} \in \text{scc-of E v by simp}
\end{assumes}
\text{moreover have v \in \text{scc-of E v by simp}}
\text{with \text{assms} have scc-of E v \cap \text{dom} (\text{discovered s}) \neq \{\} by blast}
\text{ultimately show thesis using \text{scc-root-exists} by metis}
\end{proof}

\begin{lemma}
\text{scc-root-transfer':}
\begin{assumes}
\text{discovered s = discovered s' tree-edges s = tree-edges s'}
\end{assumes}
\text{shows scc-root s r scc \leftarrow\rightarrow scc-root s' r scc}
\end{lemma}

\text{unfolding scc-root-def}
by (simp add: assms)

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')
  \( \forall x \in \text{dom} (\text{discovered} s), \delta s x = \delta s' x \) 
  \( \forall x \in \text{dom} (\text{discovered} s') - \text{dom} (\text{discovered} s), \delta s' x \geq \text{counter} s \)
  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: \( \delta s' ?sd \neq {} \) by blast
  from discovered-finite have fin: finite (\( \delta s' ?sd \)) by simp

  from timing-less-counter d have \( \bigwedge x. x \in \delta s' ?sd' \Rightarrow x < \text{counter} s \) by auto
  hence Min: \( \text{Min} (\delta s' ?sd) < \text{counter} s \)
  using Min-less-iff by blast

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

  finally have Min (\( \delta s ' ?sd \)) = Min (\( \delta s' ?sd' \)).
  } note aux = this

show \( ?thesis \)

proof
  assume r: scc-root s r scc
  from r-d d have \( \delta s' r = \delta s r \) by simp
  also from r scc-root-is-Min-disc have \( \delta s r = \text{Min} (\delta s ' ?sd) \) by simp
  also from r aux have Min (\( \delta s ' ?sd \)) = Min (\( \delta s' ?sd' \)) by (simp add: scc-root-def)

69
finally show \( scc\text{-root} \ s' \ r \ scc \)
using \( r\cdot d \cdot r\cdot [\text{unfolded } scc\text{-root-def}] \)
by (blast intro!: \( s'.\Min\text{-disc-is-scc\text{-root}} \))

next
assume \( r': scc\text{-root} \ s' \ r \ scc \)
from \( r\cdot d \cdot d \cdot r\cdot \) have \( \delta \ s \ r = \delta \ s' \ r \) by simp
also from \( r' \cdot s'. scc\text{-root-is-Min-disc} \) have \( \delta \ s' \ r = \Min \ (\delta \ s' \ ?s'd') \) by simp
also from \( r' \cdot aux \) have \( \Min \ (\delta \ s' \ ?s'd') = \Min \ (\delta \ s \ ?s'd) \) by (simp add: scc-root-def)

finally show \( scc\text{-root} \ s \ r \ scc \)
using \( r\cdot d \cdot d \cdot r \cdot [\text{unfolded } scc\text{-root-def}] \)
by (blast intro!: \( \Min\text{-disc-is-scc\text{-root}} \))

qed

end

1.4 Generic DFS and Refinement

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

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

named-theorems DFS-code-unfold
\( \langle \text{DFS framework: Unfolding theorems to prepare term for automatic refinement} \rangle \)

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

1.4.1 Generic DFS Algorithm

record \( \langle 'v, 's \rangle \ ) \text{-gen-dfs-struct} =
gds-init :: \( 's \ nres \)
gds-is-break :: \( 's \Rightarrow \text{bool} \)
gds-is-empty-stack :: \( 's \Rightarrow \text{bool} \)
gds-new-root :: \( 'v \Rightarrow 's \Rightarrow 's \ nres \)
gds-get-pending :: \( 's \Rightarrow ( 'v \times 'v \text{ option } \times 's) \ nres \)
gds-finish :: \( 'v \Rightarrow 's \Rightarrow 's \ nres \)
gds-is-discovered :: \( 'v \Rightarrow 's \Rightarrow \text{bool} \)
gds-is-finished :: \( 'v \Rightarrow 's \Rightarrow \text{bool} \)
gds-back-edge :: \( 'v \Rightarrow 'v \Rightarrow 's \Rightarrow 's \ nres \)

70
locale gen-dfs-defs =
  fixes gds :: ('v,'s) gen-dfs-struct
  fixes V0 :: 'v set
begin

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

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

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

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

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

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

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

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

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

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

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

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

72
abbreviation gen-rwof-assert ≡ rwof (gds-init gds) gen-cond gen-step-assert

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

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

lemma gen-dfs-eq-assert: gen-dfs = gen-dfs-assert
unfolding gen-dfs-def gen-dfs-assert-def
apply (rule antisym)
subgoal
apply (unfold gen-step-def[abs-def] gen-step-assert-def[abs-def]) []
apply (unfold GHOST-elim-Let) []
apply (rule refine-IdD)
by (refine-rcg, refine-dref-type, simp-all) []

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

lemma gen-dfsT-eq-assert: gen-dfsT = gen-dfsT-assert

73
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-asym) 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-asym) 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-asym) only: gen-step-def[abs-def] gen-step-assert-def[abs-def]) []
  apply (unfold GHOST-elim-Let) []
apply (rule leafI)
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)

75
definition do-init ≡ do {
  e ← on-init param;
  gbs-init gbs e
}

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

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

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

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

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

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

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

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

sublocale gen-dfs-defs gds V0 .
end

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

context param-DFS-defs begin

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

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

sublocale gen-dfs: gen-param-dfs-defs gbs param state

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-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
t 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
t for gdsi V0i gds V0

 fixes V S

 assumes BIJV[relator-props]: bijective V
 assumes V0-param[param]: (V0i, V0) ∈ (V) set-rel
 assumes is-discovered-param[param]:
  (gds-is-discovered gdsi, gds-is-discovered gds) ∈ V → S → bool-rel
 assumes is-finished-param[param]:
  (gds-is-finished gdsi, gds-is-finished gds) ∈ V → S → bool-rel
 assumes is-empty-stack-param[param]:
  (gds-is-empty-stack gdsi, gds-is-empty-stack gds) ∈ S → bool-rel
 assumes is-break-param[param]:
  (gds-is-break gdsi, gds-is-break gds) ∈ S → bool-rel
 assumes init-refine[refine]:
  gds-init gdsi ≤⇓ S (gds-init gds)
 assumes new-root-refine[refine]:
  [a.pre-new-root v0 s; (v0i, v0) ∈ V; (si, s) ∈ S]
    ⇒ gds-new-root gdsi v0i si ≤⇓ S (gds-new-root gds v0 s)
 assumes get-pending-refine[refine]:
  [a.pre-get-pending s; (si, s) ∈ S]
    ⇒ gds-get-pending gdsi si ≤⇓ (V ×r (V) option-rel ×r S) (gds-get-pending gds s)
 assumes finish-refine[refine]:
  [a.pre-finish v s0 s; (vi, v) ∈ V; (si, s) ∈ S]
    ⇒ gds-finish gdsi vi si s ≤⇓ 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 ≤⇓ 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 ≤⇓ S (gds-back-edge gds u v s)
assumes discover-refine[refine]:
\[ \text{a.pre-discover } u \; v \; s^0 \; s; \; (u,i) \in V; \; (v,i) \in V; \; (s,i) \in S \]
\[ \Rightarrow \text{gds-discover } gdsi \; ui \; vi \; si \; \leq \Downarrow \; S \; (\text{gds-discover } gds \; u \; v \; s) \]

begin
term gds-is-discovered gdsi

lemma select-v0-refine[refine]:
assumes s-param: \((si,s) \in S\)
shows \(\text{SPEC } (\lambda v0. \; v0 \in V0i \; \land \; \neg \; \text{gds-is-discovered } gdsi \; v0 \; si)
\[ \leq \Downarrow \; V \; (\text{SPEC } (\lambda v0. \; v0 \in V0 \; \land \; \neg \; \text{gds-is-discovered } gds \; v0 \; s)) \]
apply (rule RES-refine)
apply (simp add: Bex-def[ symmetric], elim conjE)
apply (drule set-relD1[OF V0-param], elim bexE)
apply is-discovered-param[ param-fo, OF - s-param]
apply auto
done

lemma gen-rwof-refine;
assumes NF: nofail (a.gen-dfs)
assumes RW: c.gen-rwof s
obtains s’ where \((s,s’) \in S \; \land \; \text{a.gen-rwof } s’
proof
from NF have NFa: nofail (a.gen-dfs-assert)
unfolding a.gen-dfs-eq-assert .
have \(\exists s’. \; (s, s’) \in S \; \land \; \text{a.gen-rwof-assert } s’
apply (rule rwof-refine[OF RW NFa[unfolded a.gen-dfs-assert-def]])
apply (rule leofI, rule init-refine)

unfolding c.gen-cond-def a.gen-cond-def
apply (rule IdD)
apply (simp only: subset-Collect-conv)
apply parametricity
unfolding c.gen-step-def a.gen-step-assert-def GHOST-elim-Let
apply (rule leofI)
apply (refine-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[parami]: (V0i, V0)∈(V)set-rel

assumes is-discovered-param[parami]:
  (gbs-is-discovered gbsi,gbs-is-discovered gbs)∈V→S→bool-rel

assumes is-finished-param[parami]:
  (gbs-is-finished gbsi,gbs-is-finished gbs)∈V→S→bool-rel

assumes is-empty-stack-param[parami]:
(gbs-is-empty-stack gbsi, gbs-is-empty-stack gbs) ∈ S → bool-rel

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

assumes gbs-init-refine[refine]: (ei, e) ∈ ES → gbs-init gbsi ei ≤ S (gbs-init gbs e)

assumes gbs-new-root-refine[refine]:
[a.pre-new-root v0 s; (v0i, v0) ∈ V; (si, s) ∈ S] → gbs-new-root gbsi v0i si ≤ S (gbs-new-root gbs v0 s)

assumes gbs-get-pending-refine[refine]:
[a.pre-get-pending s; (si, s) ∈ S] → gbs-get-pending gbsi si ≤ S (gbs-get-pending gbs s)

assumes gbs-finish-refine[refine]:
[a.pre-finish v s0 s; (vi, v) ∈ V; (si, s) ∈ S] → gbs-finish gbsi vi s i ≤ S (gbs-finish gbs v s)

assumes gbs-cross-edge-refine[refine]:
[a.pre-cross-edge u v s0 s; (ui, u) ∈ V; (vi, v) ∈ V; (si, s) ∈ S] → gbs-cross-edge gbsi ui vi si ≤ S (gbs-cross-edge gbs u v s)

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

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

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

assumes on-init-refine[refine]: on-init parami ≤ S (on-init param)

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

assumes on-new-root-refine[refine]:
[a.pre-new-root v0 s; (v0i, v0) ∈ V; (si, s) ∈ S;
(si′, s′) ∈ S; nf-inres (gbs-new-root gbs v0 s) s′]
assumes on-finish-refine[refine]:
[a.pre-finish v s0 s; (vi, v) ∈ V; (si, s) ∈ S; (si’, s’) ∈ S;
 nf-inres (gbs-finish gbs v s) s’]
⇒ on-finish parami vi si’ ≤ ⇓ ES (on-finish param v s')

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

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

assumes on-discover-refine[refine]:
[a.pre-discover u v s0 s; (ui, u) ∈ V; (vi, v) ∈ V; (si, s) ∈ S;
 (si’, s’) ∈ S; nf-inres (gbs-discover gbs u v s) s’]
⇒ on-discover parami ui vi si’ ≤ ⇓ ES (on-discover param u v s')
1.5 Tail-Recursive Implementation

theory Tailrec-Impl
imports General-DFS-Structure
begin
locale tailrec-impl-defs = graph-defs G + gen-dfs-defs gds V0
for G :: ('v, 'more) graph-rec-scheme
and gds :: ('v,s)gen-dfs-struct
begin
definition [DFS-code-unfold]: tr-impl-while-body ≡ λs. do { (u,Vs,s) ← gds-get-pending gds s;
case Vs of
  None ⇒ gds-finish gds u s |
  Some v ⇒ do {
    if gds-is-discovered gds v s then do {
      if gds-is-finished gds v s then
gds-cross-edge gds u v s
    else
gds-back-edge gds u v s
    }
    else
gds-discover gds u v s
  }
}
definition tailrec-implT where [DFS-code-unfold]:
tailrec-implT ≡ do {
s ← gds-init gds;
FOREACHci (λit s.
  gen-rwof s
∧ (~gds-is-break gds s → gds-empty-stack gds s )
∧ V0—it ⊆ gen-discovered s)
V0
(Not o gds-is-break gds)
(λv0 s. do {
  let — ghost: s0 = s;
  if gds-is-discovered gds v0 s then
    RETURN s
  else do {
    s ← gds-new-root gds v0 s;
WHILEIT
(\(\lambda s\). \text{gen-rwof } s \land \text{insert } v_0 \ (\text{gen-discovered } s_0) \subseteq \text{gen-discovered } s)
(\(\lambda s\). \neg \text{gds-is-break } gds\ s \land \neg \text{gds-is-empty-stack } gds\ s)
\text{tr-impl-while-body } s
}
})\ s
}

\textbf{definition tailrec-impl where} [DFS-code-unfold]:
tailrec-impl \equiv \text{do } \{
s \leftarrow \text{gds-init } gds;
\text{FOREACHci}
(\(\lambda it\ s\). \text{gen-rwof } s
\land (\neg \text{gds-is-break } gds\ s \rightarrow \text{gds-is-empty-stack } gds\ s )
\land V_0 - it \subseteq \text{gen-discovered } s)
V_0
(\text{Not o gds-is-break } gds)
(\lambda v_0\ s. \text{do } \{
\text{let } \text{ghost}: s_0 = s;
\text{if gds-is-discovered } gds\ v_0\ s \text{ then }
\text{RETURN } s
\text{else } \text{do } \{
\text{s } \leftarrow \text{gds-new-root } gds\ v_0\ s;
\text{WHILEI}
(\(\lambda s\). \text{gen-rwof } s \land \text{insert } v_0 \ (\text{gen-discovered } s_0) \subseteq \text{gen-discovered } s)
(\(\lambda s\). \neg \text{gds-is-break } gds\ s \land \neg \text{gds-is-empty-stack } gds\ s)
(\(\lambda s.\) \text{do } \{
(u, Vs, s) \leftarrow \text{gds-get-pending } gds\ s;
\text{case } Vs \text{ of }
\text{None } \Rightarrow \text{gds-finish } gds\ u\ s
| \text{Some } v \Rightarrow \text{do } \{
\text{if gds-is-discovered } gds\ v\ s \text{ then } \text{do } \{
\text{if gds-is-finished } gds\ v\ s \text{ then }
gds-cross-edge gds\ u\ v\ s
\text{else}
gds-back-edge gds\ u\ v\ s
\} \text{ else }
gds-discover gds\ u\ v\ s
\} \})\ s
\} \})\ s
\}
\text{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^w SPEC (gds-is-empty-stack gds)
assumes new-root-discovered:
[pre-new-root v0 s] \Rightarrow gds-new-root gds v0 s \leq_n SPEC (\lambda s'.
insert v0 (gen-discovered s) \subseteq gen-discovered s')
assumes get-pending-incr:
[pre-get-pending s] \Rightarrow gds-get-pending gds s \leq_n SPEC (\lambda s'.
gen-discovered s \subseteq gen-discovered s')
assumes finish-incr: [pre-finish u s0 s]
\Rightarrow gds-finish gds u s \leq_n SPEC (\lambda s'.
gen-discovered s \subseteq gen-discovered s')
assumes cross-edge-incr: pre-cross-edge u v s0 s
\Rightarrow gds-cross-edge gds u v s \leq_n SPEC (\lambda s'.
gen-discovered s \subseteq gen-discovered s')
assumes back-edge-incr: pre-back-edge u v s0 s
\Rightarrow gds-back-edge gds u v s \leq_n SPEC (\lambda s'.
gen-discovered s \subseteq gen-discovered s')
assumes discover-incr: pre-discover u v s0 s
\Rightarrow gds-discover gds u v s \leq_n SPEC (\lambda s'.
gen-discovered s \subseteq gen-discovered s')

begin

class

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. gen-rwof s \land 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'.
gen-rwof s' 
\land insert v0 (gen-discovered s) \subseteq gen-discovered s')
apply (rule SPEC-rule-conj-leofI1)
apply (rule order-trans[OF - rwof-step[OF nofail]])

85
using PNR apply (unfold gen-step-def gen-cond-def pre-new-root-def) [3]
apply (simp add: pw-le-iff refine-pw-simps, blast)
apply simp
apply blast
apply (rule new-root-discovered[OF PNR])
done

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

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

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

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

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

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

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

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

moreover note get-pending-incr[OF PRE]

ultimately show ?thesis by (simp add: pw-le-iff pw-leaf-iff)
qed

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

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 ≤ SPEC (λs'. gen-rwof s'
∧ gen-discovered s ⊆ gen-discovered s')
apply (rule SPEC-rule-conj-leofII)

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 ≤ SPEC (λs'. gen-rwof s'
∧ gen-discovered s ⊆ gen-discovered s')
apply (rule SPEC-rule-conj-leofII)

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 \textit{PRE}: pre-discover \( u \) \( v \) \( s_0 \) \( s \)

shows \( \text{gds-discover gds } u \) \( v \) \( s \) \( \leq \text{SPEC } (\lambda s'. \text{gen-rwof } s' \wedge \text{gen-discovered } s \subseteq \text{gen-discovered } s') \)

apply (rule SPEC-rule-conj-leofI)

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

end

lemma \textit{gen-step-disc-incr}: 
assumes \textit{nofail gen-dfs}
assumes \( \text{gen-rwof } s \) \( \text{insert } v_0 \) \( (\text{gen-discovered } s_0) \subseteq \text{gen-discovered } s \)
assumes \( \neg \text{gds-is-break gds } s \)
\( \neg \text{gds-is-empty-stack gds } s \)
shows \( \text{gen-step } s \leq \text{SPEC } (\lambda s. \text{insert } v_0 \ (\text{gen-discovered } s_0) \subseteq \text{gen-discovered } s) \)
using \textit{assms}
apply (simp only: \textit{gen-step-def} \textit{gen-dfs-def})
apply (refine-rd \textit{refine-vcg} 
order-trans\([\text{OF gds-init-refine}]\]
order-trans\([\text{OF gds-new-root-refine}]\]
order-trans\([\text{OF gds-get-pending-refine}]\]
order-trans\([\text{OF gds-finish-refine}]\]
order-trans\([\text{OF gds-cross-edge-refine}]\]
order-trans\([\text{OF gds-back-edge-refine}]\]
order-trans\([\text{OF gds-discover-refine}]\]
)
apply (auto 
simp: \textit{it-step-insert-iff gen-cond-def} 
pre-new-root-def \textit{pre-get-pending-def} \textit{pre-finish-def}
\textit{pre-cross-edge-def} \textit{pre-back-edge-def} \textit{pre-discover-def})
done

theorem \textit{tailrec-impl}: \textit{tailrec-impl} \( \leq \text{gen-dfs} \)
unfolding \textit{gen-dfs-def}
apply (rule WHILE-refine-rwof)
unfolding \textit{tailrec-impl-def}
apply (refine-rd \textit{refine-vcg} 
order-trans\([\text{OF gds-init-refine}]\]
order-trans\([\text{OF gds-new-root-refine}]\]

88
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 ≤ gen-step s
  unfolding tr-impl-while-body-def gen-step-def
  by simp

lemma tailrecT-impl: tailrec-implT ≤ gen-dfsT
proof (rule le-nofailI)
  let ?V = rwof-rel (gds-init gds) gen-cond gen-step
  assume NF: nfail gen-dfsT
  from nfail-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': nfail 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
    "∀s. [gen-rwof s; gen-cond s] → gen-step s ≤ SPEC (λs'. (s,s')∈?V)"
    .
    hence aux: "∀s. [gen-rwof s; gen-cond s] → gen-step s ≤ SPEC (λs'. (s,s')∈?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
} }

definition gen-dfs' ≡ gds-init gds ≫ WHILE gen-cond gen-step'
abbreviation gen-rwof' ≡ rwof (gds-init gds) gen-cond gen-step'
**definition** rec-impl where [DFS-code-unfold]:

rec-impl ≡ do { s ← gds-init gds;

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

\[s; \]

\[
\text{if } gds\text{-is-break } gds \ s \ \text{then} \\
  \text{RETURN } s \\
\text{else } \{ \\
  s \leftarrow \text{choose-pending } u \ (\text{None}) \ s; \\
  s \leftarrow \text{gds-finish } gds \ u \ s; \\
  \text{RETURN } s \\
\}\) \((v_0, s)\)
\]

\[
\text{definition } \text{rec-impl-for-paper where } \text{rec-impl-for-paper} \equiv \{ \\
\text{s } \leftarrow \text{gds-init } gds; \\
\text{FOREACH} c \ V_0 \ (\text{Not } o \ gds\text{-is-break } gds) \ (\lambda v_0 \ s. \ \{ \\
  \text{if } gds\text{-is-discovered } gds \ v_0 \ s \ \text{then } \text{RETURN } s \\
  \text{else } \{ \\
  s \leftarrow \text{gds-new-root } gds \ v_0 \ s; \\
  \text{if } gds\text{-is-break } gds \ s \ \text{then } \text{RETURN } s \\
  \text{else } \{ \\
  \text{REC } \lambda (u, s). \ \{ \\
  s \leftarrow \text{FOREACH} c \ (E^{\prime}(u)) \ (\lambda s. \ \neg gds\text{-is-break } gds \ s) \ (\lambda v \ s. \ \{ \\
  s \leftarrow \text{choose-pending } u \ (\text{Some } v) \ s; \\
  \text{if } gds\text{-is-discovered } gds \ v \ s \ \text{then } \{ \\
  \text{if } gds\text{-is-finished } gds \ v \ s \ \text{then } \text{gds-cross-edge } gds \ u \ v \ s \\
  \text{else } \text{gds-back-edge } gds \ u \ v \ s \\
  \} \ \text{else } \{ \\
  s \leftarrow \text{gds-discover } gds \ u \ v \ s; \\
  \text{if } gds\text{-is-break } gds \ s \ \text{then } \text{RETURN } s \ \text{else } D(v, s) \\
  \} \\
  \}) \) \((v_0, s)\) \\
  \} \} \) \((v_0, s)\) \\
  \} \}
\]

\[92\]
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 \( s' = pending \land \text{stack} \ s' = \text{stack} \land \\
(\forall \ x. \ \text{gds-is-discovered} \ gds \ x \ s' = \text{gds-is-discovered} \ gds \ x \ s)\)

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

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

begin

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

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

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

lemma reachable-invar: 
\(\text{gen-rwof}' \ s \implies \text{set} (\text{stack} \ s) \subseteq \text{reachable} \land \text{pending} \ s \subseteq E \\
\land \text{set} (\text{stack} \ s) \subseteq \text{gen-discovered} \ s \land \text{distinct} (\text{stack} \ s)\)
\( \wedge \text{pending } s \subseteq \text{set } (\text{stack } s) \times \text{UNIV} \)

apply (erule establish-rwof-invar[rotated \(-1\)])
apply (rule leof-trans[OF init-spec], auto)
apply (subst gen-step',def)
apply (refine-reg refine-weg)

leaf-trans[OF new-root-spec]
SELECT-rule[THEN leof-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 (simp add: pre-defs, simp add: gen-cond-def)

apply ((unfold pre-defs, intro conjI); assumption?)
subgoal by (clarsimp simp: gen-cond-def)
subgoal by (clarsimp simp: gen-cond-def)
subgoal

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

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

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

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

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

subgoal
apply clarsimp
by blast

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

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

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

lemma mk-spec-aux:
\[ m \leq u \text{ SPEC } \Phi; m \leq \text{SPEC } \text{gen-rwof}\'] \implies m \leq \text{SPEC } (\lambda s. \text{gen-rwof}\' s \land \Phi s) \]
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 } (\text{stack } s0)
\land \text{inres } (\text{choose-pending } u \text{ vo } s0) s
\land \text{stack } s = \text{stack } s0
\land (\forall x. \text{gds-is-discovered } \text{gds } x \text{ s} = \text{gds-is-discovered } \text{gds } x \text{ s0})
//gds/try/gds/\text{gds/}gds/\text{gds/}gds/\text{gds/}gds/\text{gds/}gds/96
\begin{align*}
&(\text{case } vo \text{ of} \\
&\quad \text{None } \Rightarrow \text{pending } s^0'' \{u\} = \{\} \land \text{pending } s = \text{pending } s^0 \\
&\quad | \text{Some } v \Rightarrow v \in \text{pending } s^0'' \{u\} \land \text{pending } s = \text{pending } s^0 - \{(u,v)\})
\end{align*}

**context**

assumes nofail:
- nofail (gds-init gds \(\models\) \(\textbf{WHILE}\) gen-cond gen-step')

assumes nofail2:
- nofail (gen-dfs)

**begin**

lemma pcp-imp-pgp:
post-choose-pending u vo s0 s \(\Rightarrow\) post-get-pending u vo s0 s

unfolding post-choose-pending-def pre-defs
apply (intro conjI)
apply (simp add: gen-rwof'-imp-rwof[OF nofail2])
apply simp
apply (simp add: gen-cond-def)
apply (rule pwD2[OF get-pending-fmt])
apply (simp add: pre-defs gen-cond-def
  gen-rwof'-imp-rwof[OF nofail2])
apply (auto simp add: refine-pw-simps select-def split: option.splits) []
done

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

schematic-goal gds-new-root-refine:
\([\text{pre-new-root } v0 s; \text{gen-rwof'} s] \Rightarrow \text{gds-new-root } gds v0 s \leq \text{SPEC } \Phi\]
apply (rule mk-spec-aux[OF new-root-spec, assumption])
apply (rule order-trans[OF - rwof-step[OF nofail, where s=s]])
unfolding gen-step'-def pre-new-root-def gen-cond-def
apply (auto simp: pw-le-iff refine-pw-simps)
done

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

proof –
from WHILE-nofail-imp-rwof-nofail[OF nofail2] 1 3 have
  nofail (choose-pending u vo s)
unfolding pre-defs gen-step'-def gen-cond-def
by (auto simp: refine-pw-simps select-def)
split: option.splits if-split-asm
also have choose-pending u vo s ≤n SPEC (post-choose-pending u vo s)
apply (rule leaf-trans[OF choose-pending-spec[OF 1 - 3, 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 ?thesis.
qed

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

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

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

schematic-goal gds-discover-refine:
[pre-discover u v s0 s; post-choose-pending u (Some v) s0 s] ⇒ gds-discover
\[ \text{gds } u v s \leq \text{SPEC } ? \Phi \]

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

apply simp
apply blast
done
end

lemma rec-impl-aux: \[ x \in \text{Domain } P \] \implies P - \{y\} \times (\text{succ } y - \text{ita}) - \{(y, xd)\} - \{xd\} \times \text{UNIV} = P - \text{insert } (y, xd) (\{y\} \times (\text{succ } y - \text{ita}))
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-wreg
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-ppp, auto simp add: pre-defs gen-rwof'-imp-rwof); fail)
apply (auto simp: post-choose-pending-def; fail)
apply (auto simp: post-choose-pending-def; fail)
apply (auto simp: post-choose-pending-def; fail)

apply ((drule pcp-imp-pgp, auto simp add: pre-defs gen-rwof'-imp-rwof); fail)
apply (auto simp: post-choose-pending-def; fail)
apply (auto simp: post-choose-pending-def; fail)
apply (auto simp: post-choose-pending-def; fail)

apply ((drule pcp-imp-pgp, auto simp add: pre-defs gen-rwof'-imp-rwof); fail)
apply (rule order-trans)
apply rprems
apply (auto; fail)

subgoal
  apply (rule SPEC-rule)
  apply (simp add: post-choose-pending-def gen-rwof'-imp-rwof
    split: if-split-asm)
  apply (clarsimp simp add: 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

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 (λu. (u, pending s' ii {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 (λu. (u, pending s' ii {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 (v. 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 \( s \)

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

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

\[
\text{lemmas } \text{impl-defs[DFS-code-unfold]} \equiv \text{init-impl-def is-empty-stack-impl-def new-root-impl-def} \\
\text{get-pending-impl-def finish-impl-def is-discovered-impl-def} \\
is-finished-impl-def discover-impl-def
\]

\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.

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

\begin

\text{sublocale} \text{tailrec-impl-defs } G \text{ c.gds} .

\text{definition} \text{get-pending } s \equiv \bigcup \{ \text{map } (\lambda(u,Vs). \{u\} \times Vs) \ (ss-stack s) \}\)

\text{definition} \text{get-stack } s \equiv \text{map } \text{fst} \ (ss-stack s)

\text{definition} \text{choose-pending}

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

where [DFS-code-unfold]:

choose-pending \( u \) \( v \) \( s \) \( \equiv \)

\[
\begin{cases}
\text{None} & \Rightarrow \text{RETURN} \ s \\
\text{Some} \ v & \Rightarrow \begin{cases}
\text{ASSERT} (ss\text{-stack} \ s \neq []) \\
\text{let} \ (u,Vs) = \text{hd} \ (ss\text{-stack} \ s) \\
\text{RETURN} \ (s\|\text{ss\text{-stack} := (u,Vs}\setminus\{v\})\#\text{tl} \ (ss\text{-stack} \ s))
\end{cases}
\end{cases}
\]

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 \( upd-exti = \text{simple-state}.more\text{-update} \)

and \( upd-ext = \text{state}.more\text{-update} \)

and \( V0i = a.V0 \)

and \( V0 = a.V0 \)

and \( V=\text{Id} \)

and \( S = (ES)\text{simple-state-rel} \)

begin

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

unfolding \( c.\text{init-impl-def} \ a.\text{empty-state-def} \ \text{simple-state-rel-def} \)

by (auto)

lemma new-root-impl:

\[ [a.\text{gen-dfs}.\text{pre-new-root} \ v0 \ s; \ (v0i, v0)\in\text{Id}; (si, s) \in (ES)\text{simple-state-rel}] \implies c.\text{new-root-impl} \ v0 \ si \leq \Downarrow ((ES)\text{simple-state-rel}) (\text{RETURN} \ (a.\text{new-root} \ v0 \ s)) \]

unfolding simple-state-rel-def \( a.\text{gen-dfs}.\text{pre-new-root-def} \ c.\text{new-root-impl-def} \)

by (auto simp add: a.pred-defs)

lemma get-pending-impl:

\[ [a.\text{gen-dfs}.\text{pre-get-pending} \ s; \ (si, s) \in (ES)\text{simple-state-rel}] \implies c.\text{get-pending-impl} \ si \leq \Downarrow (\text{Id} \times, \text{Id} \times, (ES)\text{simple-state-rel}) (a.\text{get-pending} \ s) \]

apply (unfold a.get-pending-def c.get-pending-impl-def) []

apply (refine-req bind-refine' Let-refine' IdI)
apply (refine-dref-type)
apply (auto
  simp: simple-state-rel-def a.gen-dfs.pre-defs a.pred-defs neg-Nil-conv
  dest: DFS-invar.stack-distinct)
)
done

lemma inres-get-pending-None-conv: inres (a.get-pending s0) (v, None, s)
  \iff s=s0 ∧ v=hd (stack s0) ∧ 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)
  \iff 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] ]
  \implies 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] ]
  \implies (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] ]
  \implies (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] ]
  \implies c.discover-impl ui vi si ≤⇓ (⟨ES⟩simple-state-rel) (RETURN (a.discover
unfolding simple-state-rel-def a.gen-dfs.pre-defs c.discover-impl-def
apply (rule ASSERT-leI)
apply (clarsimp simp: inres-get-pending-Some-conv)
apply (frule DFS-invar.stack-discovered)
apply (auto simp: a.pred-defs) []
apply (clarsimp simp: inres-get-pending-Some-conv)
apply (frule DFS-invar.stack-discovered)
apply (frule DFS-invar.pending-ssE)
apply (clarsimp simp: a.pred-defs)
apply blast
done

sublocale gen-param-dfs-refine
where gbsi = c.gbs
and gbs = a.gbs
and upd-exti = simple-state.more-update
and upd-ext = state.more-update
and V0i = a.V0
and V0 = a.V0
and V = Id
and S = (ES) simple-state-rel
apply unfold-locales
apply (simp-all add: is-break-param)
apply (auto simp: a.is-discovered-def c.is-discovered-impl-def simple-state-rel-def) []
apply (auto simp: a.is-finished-def c.is-finished-impl-def simple-state-rel-def) []
apply (auto simp: a.is-empty-stack-def c.is-empty-stack-impl-def simple-state-rel-def) []
apply (refine-rcg init-impl)
apply (refine-rcg new-root-impl, simp-all) []
apply (refine-rcg get-pending-impl) []
apply (refine-rcg finish-impl, simp-all) []
apply (refine-rcg cross-edge-impl, simp-all) []
apply (refine-rcg back-edge-impl, simp-all) []
apply (refine-rcg discover-impl, simp-all) []
done

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

**Lemma** `simple-refine[refine]`: \( c.\text{gen-dfs} \leq \downarrow ((E\text{S})\text{simple-state-rel}) \ a.\text{it-dfs} \)
- **Using** `gen-dfs-refine`
- **By** `simp`

**Lemma** `simple-refineT[refine]`: \( c.\text{gen-dfsT} \leq \downarrow ((E\text{S})\text{simple-state-rel}) \ a.\text{it-dfsT} \)
- **Using** `gen-dfsT-refine`
- **By** `simp`

Link with tail-recursive implementation

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

**Lemma** `simple-tailrec-refine[refine]`: \( \text{tailrec-impl} \leq \downarrow ((E\text{S})\text{simple-state-rel}) \ a.\text{it-dfs} \)
- **Proof** –
  - **Note** `tailrec-impl` also **Note** `simple-refine` **Finally show** `?thesis` .
- **Qed**

**Lemma** `simple-tailrecT-refine[refine]`: \( \text{tailrec-implT} \leq \downarrow ((E\text{S})\text{simple-state-rel}) \ a.\text{it-dfsT} \)
- **Proof** –
  - **Note** `tailrecT-impl` also **Note** `simple-refineT` **Finally show** `?thesis` .
- **Qed**

Link to recursive implementation

**Lemma** `reachable-invar`:
- **Assumes** `c.\text{gen-rwof} \ s`
- **Shows** `set (map fst (ss-stack s)) \subseteq \text{visited} \ s`
  - **And 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-vec)`
- **Apply** `simp`

**Unfolding** `c.\text{gen-step-def} c.\text{do-action-defs} c.\text{impl-defs[abs-def]} c.\text{gds-simps} c.\text{gbs-simps}
- **Apply** `(refine-reg refine-vec)`
- **Apply** `simp-all`
- **Apply** `(fastforce simp: neq-Nil-conv)`
- **Apply** `(fastforce simp: neq-Nil-conv)`
- **Apply** `(fastforce simp: neq-Nil-conv)`
- **Apply** `(fastforce simp: neq-Nil-conv)`
- **Done**

107
**locale** 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** (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),
\[
\{ (\text{ss-stack} = s, \text{on-stack} = os, \text{visited} = vis, \ldots = m) \mid \\
si osi visi mi s os vis m. \\
(si, s) \in s-rel \land \\
(osi, os) \in vis-rel \land \\
(visi, vis) \in vis-rel \land \\
(mi, m) \in erel \}
\]

\text{consts}

\text{i-simple-state :: interface } \Rightarrow \text{ interface } \Rightarrow \text{ interface } \Rightarrow \text{ interface}

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

\text{term simple-state-ext}

\text{lemma [autoref-rules, param]:}

\text{fixes s-rel ps-rel vis-rel erel}

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

\text{shows}

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

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

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

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

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

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

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

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

\text{unfolding ss-impl-rel-def R-def}

\text{apply auto}

\text{apply parametricity+}

\text{done}

1.7.2 Simple state without on-stack

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

\text{record (si,nsi) simple-state-nos-impl =}

\text{ssnos-stack-impl :: 'si}

\text{ssnos-visited-impl :: 'nsi}

\text{definition [to-relAPP]: ssnos-impl-rel s-rel vis-rel erelields}

\text{\{ ((ssnos-stack-impl = si, ssnos-visited-impl = visi, \ldots = mi),}

\text{(ss-stack = s, on-stack = os, visited = vis, \ldots = m)) \mid \\
si visi mi s os vis m. \\
(si, s) \in s-rel \land \\
(visi, vis) \in vis-rel \land \\
(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 ⇒ (s, -)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
(λx. 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
(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]: ssns-impl-rel (R::(a×b) set) vis-rel erel ≡
{(ssns-visited-impl = visi, ... = mi),
 (ss-stack = s, on-stack = os, visited = vis, ... = m)}
visi mi s os vis m.
(visi, vis) ∈ vis-rel ∧
(mi, m) ∈ erel
lemmas [autoref-rel-intf] = REL-INTFI[of ssns-impl-rel i-simple-state]

definition op-ns-on-stack-update 
:: (- set ⇒ - set) ⇒ (\ Veronica \) simple-state-scheme ⇒ -
where op-ns-on-stack-update ≡ on-stack-update

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

class begin interpretation autoref-syn .

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

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

end

class begin 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 \Rightarrow \ RETURN \ s
    | Some \ v \Rightarrow \ do \ 
        - \ ← \ ASSERT \ (ss-stack \ s \neq \ [])';
        \ RETURN \ (op-ns-stack-update
            (\ let
                \ (u, \ Vs) = \ hd \ (ss-stack \ s)
                \ in \ (\ \lambda \ (. \ u, \ Vs \setminus \{v\}) \# \ tl \ (ss-stack \ s))
            )
        )
    s
)
unfolding choose-pending-def op-ns-stack-update-def
by (auto split: option.split prod.split)

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

end
lemma [autoref-rules, param]:
fixes s-rel vis-rel erel ANY-rel
defines \( R \equiv \langle \text{ANY-rel, vis-rel, erel} \rangle \text{ssns-impl-rel} \)
shows
\( (\text{ssns-visited-impl, visited}) \in R \rightarrow \text{vis-rel} \)
\( (\text{simple-state-ns-impl.more, simple-state.more}) \in R \rightarrow \text{erel} \)
\( \bigwedge f. (\lambda x. x, \text{op-ns-on-stack-update } f) \in R \rightarrow R \)
\( (\text{ssns-visited-impl-update, visited-update}) \in (\text{vis-rel} \rightarrow \text{vis-rel}) \rightarrow R \rightarrow R \)
\( (\text{simple-state-ns-impl.more-update, simple-state.more-update}) \in (\text{erel} \rightarrow \text{erel}) \rightarrow R \rightarrow R \)
\( (\lambda - ps vs, \text{simple-state-ns-impl-ext } ps vs, \text{simple-state-ext}) \in \text{ANY1-rel} \rightarrow \text{ANY2-rel} \rightarrow \text{vis-rel} \rightarrow \text{erel} \rightarrow R \)
unfolding ssns-impl-rel-def \( R \)-def \( \text{op-ns-on-stack-update-def} \)
apply auto
apply parametricity+
done

lemma [refine-transfer-post-simp]:
\( \bigwedge a m. a\{\text{simple-state-ns-impl.more} := m::\text{unit}\} = a \)
\( \bigwedge a m. a\{\text{simple-state-impl.more} := m::\text{unit}\} = a \)
\( \bigwedge a m. a\{\text{simple-state-ns-impl.more} := m::\text{unit}\} = a \)
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 \( G'=\langle \text{rel-restrict } E R, V0-R \rangle \) for some \( \text{fb-graph } G=\langle E, V0 \rangle \). 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 \( R \equiv \{ (s,s') \}
(\text{ss-stack } s, \text{ss-stack } s') \in (\text{Id} \times_r \{(U, U'). \ U-R = U'\}) \text{list-rel} \)
∧ on-stack s = on-stack s'
∧ visited s = visited s' ∪ R ∧ visited s' ∩ R = {}
∧ simple-state.more s = simple-state.more s'

lemma restr-rel-simps:
assumes (s,s') ∈ restr-rel R
shows visited s = visited s' ∪ R
and simple-state.more s = simple-state.more s'
using assms unfolding restr-rel-def by auto

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

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

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

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

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

abbreviation rel ≡ restr-rel R

definition gbs' ≡ gbs (!)
gbs-init := λe. RETURN
                   ( ss-stack=[], on-stack={}, visited = R, ...=e ] )

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

sublocale gen-param-dfs-defs gbs'

sublocale tailrec-impl-defs G gds.
end
locale restricted-impl =
  fb-graph +
a: simple-impl graph-restrict G R +
  restricted-impl-defs +

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

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

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

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

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

begin
lemmas rel-def = restr-rel-def[where R=R]

sublocale gen-param-dfs gbs' parami simple-state more-update V0.

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

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

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

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

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

lemma no-pending-refine:
assumes (s,s')∈rel ¬a.c.is-empty-stack-impl s'
shows (hd (ss-stack s) = u.[]) └→ hd (ss-stack s') = (u.[])
using assms
unfolding a.c.is-empty-stack-impl-def rel-def
apply (cases ss-stack s', simp)
apply (auto elim: list-relE)
done

lemma do-new-root-refine[refine]:
\[(v0i,v0) \in Id; \ (si,s) \in \text{rel}; \ v0 \notin R \]
\[\implies \text{do-new-root } v0i \ si \leq \downarrow \text{rel} \ (a.c.\text{do-new-root } v0 \ s)\]

**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]:

\[\] \[\ (s,s') \in \text{rel}; \ (u,u') \in \text{Id} \]
\[\implies \text{do-finish } u \ s \leq \downarrow \text{rel} \ (a.c.\text{do-finish } u' \ s')\]

**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:

\[\] \[\ (s,s') \in \text{rel}; \neg \text{is-empty-stack-impl } s; \ vs \in Vs; \ vs \notin R; \]
\[\text{hd} \ (\text{ss-stack } s) = (u,Vs) \implies \text{hd} \ (\text{ss-stack } s') = (u,\text{insert } vs \ (Vs-R))\]

**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** \( (s,s') \in \text{rel} \ \neg \text{is-empty-stack-impl } s \)

**shows**

**get-pending-impl** \( s \leq \ (\sup \ (\downarrow \text{rel}) \ (\inf \ (\text{get-pending-impl } s')) \)

\( (\text{SPEC } (\lambda (\cdot, Vs,\cdot). \ \text{case } Vs \text{ of } \text{None } \Rightarrow \text{True } \ | \ \text{Some } v \Rightarrow v \notin R))\)

\( (\downarrow (\text{rel}) \ (\text{get-pending-impl } s')) \)

\( (\text{SPEC } (\lambda (u, Vs, s''). \ \exists v. \ Vs=\text{Some } v \land v \in R \land s''=s') \)

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: pred-rel-simps restr-rel-simps
      RETURN-RES-refine iff)
apply (rule le-supI1)
apply (simp add: pred-rel-simps no-pending-refine restr-rel-simps
      RETURN-RES-refine-iff)
apply (rule le-supI2)
apply (rule RETURN-SPEC-refine)
apply (auto simp: rel-def is-empty-stack-impl-def neg-nil-conv)
apply (cases ss-stack s', simp) apply (auto elim!: list-relE)
apply (rule le-supI1)
apply (rule rhs-step-bind-RES, blast)
apply (simp add: rel-def is-empty-stack-impl-def)
apply (cases ss-stack s', simp)
done

qed

lemma do-discover-refine[refine]:
   [ (s, s') ∈ rel; (u,u')∈Id; (v,v')∈Id; v' ∉ R ]
\[ \rightarrow \text{do-discover } u \, v \, s \leq \downset \text{rel} (a \cdot c \cdot \text{do-discover } u' \, v' \, s') \]

unfolding do-action-defs a \cdot c \cdot do-action-defs 

apply refine-reg
  apply (rule intro-prgR[where \( R = \text{rel} \)])
  apply (simp add: discover-impl-def a \cdot c \cdot discover-impl-def)
  apply (refine-reg, 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: \[ [(s, s') \in \text{rel}; v \in R] \rightarrow \text{is-discovered-impl } v \, s \]
by (auto simp: pred-rel-simps)

lemma re-refine-aux: gen-dfs \leq \downset a \cdot c \cdot gen-dfs
unfolding a \cdot c \cdot gen-dfs-def gen-dfs-def
apply (simp del: graph-restrict-simps)

apply (rule bind-refine)
apply (refine-reg)
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 \cdot c \cdot gen-step-def del: graph-restrict-simps)
apply refine-reg
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 (rule le-supI2)

apply (rule RETURN-as-SPEC-refine)

apply (clarsimp simp add: conc-fun-SPEC)

apply (rule le-supI2)
apply (rule RETURN-as-SPEC-refine)
apply (clarsimp simp add: conc-fun-SPEC)

apply (clarsimp simp add: pred-rel-simps)

apply (clarsimp simp add: refine-rcg refine-dref-type)

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 (clarsimp 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
  apply unfold-locales
  apply (clarsimp 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 examples are re-usable algorithms, that can easily be integrated into other (refinement framework based) developments.

The cyclicity checker example contains a detailed description of how to use the DFS framework, and can be used as a guideline for own DFS-framework based developments.

2.1 Simple Cyclicity Checker

theory Cyc-Check
imports ../DFS-Framework
      CAVA-Automata.Digraph-Impl
      ../Misc/Impl-Rev-Array-Stack
begin

This example presents a simple cyclicity checker: Given a directed graph with start nodes, decide whether it’s reachable part is cyclic.

The example tries to be a tutorial on using the DFS framework, explaining every required step in detail.

We define two versions of the algorithm, a partial correct one assuming only a finitely branching graph, and a total correct one assuming finitely many reachable nodes.

2.1.1 Framework Instantiation

Define a state, based on the DFS-state. In our case, we just add a break-flag.

record 'v cyc-c-state = 'v state +
         break :: bool

Some utility lemmas for the simplifier, to handle idiosyncrasies of the record package.
lemma break-more-cong: state.more s = state.more s' \implies break s = break s'
  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 \equiv dflt-parametrization state.more
    (RETURN (| break = False |)) (|)
    on-back-edge := \lambda --. 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 \equiv do {
  ASSERT (fb-graph G);
  s \leftarrow cycc.it-dfs TYPE('a) G;
  RETURN (break s)
}

The total correct variant asserts finitely many reachable nodes.

definition cyc-checkerT G \equiv do {
  ASSERT (graph G \land finite (graph-defs.reachable G));
  s \leftarrow cycc.it-dfsT TYPE('a) G;
  RETURN (break s)
}

Next, we define a locale for the cyclicity checker’s precondition and invariant, by specializing the param-DFS locale.

locale cycc = param-DFS G cycc-params for G :: ('v, 'more) graph-rec-scheme
begin

We can easily show that our parametrization does not fail, thus we also get the DFS-locale, which gives us the correctness theorem for the DFS-scheme

sublocale DFS G cycc-params
  apply unfold-locales
  apply (simp-all add: cycc-params-def)
  done
thm it-dfs-correct — Partial correctness
thm it-dfsT-correct — Total correctness if set of reachable states is finite
end

lemma cyccI:
  assumes fb-graph G
  shows cycc G
proof —
  interpret fb-graph G by fact
  show ?thesis by unfold-locales
qed

lemma cyccI':
  assumes graph G
  and FR: finite (graph-defs.reachable G)
  shows cycc G
proof —
  interpret graph G by fact
  from FR interpret fb-graph G by (rule fb-graphI-fr)
  show ?thesis by unfold-locales
qed

Next, we specialize the DFS-invar locale to our parameterization. This locale contains all proven invariants. When proving new invariants, this locale is available as assumption, thus allowing us to re-use already proven invariants.

locale cycc-invar = DFS-invar where param = cycc-params + cycc

The lemmas to establish invariants only provide the DFS-invar locale. This lemma is used to convert it into the cycc-invar locale.

lemma cycc-invar-eq[simp]:
  shows DFS-invar G cycc-params s ←→ cycc-invar G s
proof
  assume DFS-invar G cycc-params s
  interpret DFS-invar G cycc-params s by fact
  show cycc-invar G s by unfold-locales
next
  assume cycc-invar G s
  then interpret cycc-invar G s .
  show DFS-invar G cycc-params s by unfold-locales
qed

2.1.2 Correctness Proof

We now enter the cycc-invar locale, and show correctness of our cyclicity checker.

context cycc-invar begin
We show that we break if and only if there are back edges. This is straight-forward from our parameterization, and we can use the `establish-invarI` rule provided by the DFS framework.

We use this example to illustrate the general proof scheme:

**Lemma (in cycc) i-brk-eq-back: is-invar (λs. break s ↔ back-edges s ≠ {})**

**Proof (induct rule: establish-invarI)**

The \[ \text{on-init cycc-params} \leq_n \text{SPEC} (\lambda x. ?I (\text{empty-state} x)); \wedge s s' v 0. \text{DFS-invar G cycc-params} s; ?I s; \text{cond} s; \neg \text{is-break cycc-params} s; \text{stack} s = []; v 0 \in \mathcal{V} 0; v 0 \notin \text{dom (discovered s)}; s' = \text{new-root} v 0 s \implies \text{on-new-root cycc-params} v 0 s' \leq_n \text{SPEC} (\lambda x. \text{DFS-invar G cycc-params} (s'[(\text{state.more} := x)])) \rightarrow ?I (s'[(\text{state.more} := x)]) \rightarrow \text{back-edges} s \equiv \text{is-break cycc-params} s; \text{stack} s \neq []; u = \text{hd} (\text{stack} s); \text{pending} s \equiv \{ u \} = \{ \}; s' = \text{finish} u s \implies \text{on-finish cycc-params} u s' \leq_n \text{SPEC} (\lambda x. \text{DFS-invar G cycc-params} (s'[(\text{state.more} := x)])) \rightarrow ?I (s'[(\text{state.more} := x)]) \rightarrow \text{is-break cycc-params} 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 cycc-params} u v s' \leq_n \text{SPEC} (\lambda x. \text{DFS-invar G cycc-params} (s'[(\text{state.more} := x)])) \rightarrow ?I (s'[(\text{state.more} := x)]) \rightarrow \text{is-break cycc-params} s; \text{stack} s \neq []; (u, v) \in \text{pending} s; u = \text{hd} (\text{stack} s); v \notin \text{dom (discovered s)}; s' = \text{back-edge} u v (s[(\text{pending} := \text{pending} s - \{(u, v)\}])]) \implies \text{on-back-edge cycc-params} u v s' \leq_n \text{SPEC} (\lambda x. \text{DFS-invar G cycc-params} (s'[(\text{state.more} := x)])) \rightarrow ?I (s'[(\text{state.more} := x)]) \rightarrow \text{is-break cycc-params} s; \text{stack} s \neq []; (u, v) \in \text{pending} s; u = \text{hd} (\text{stack} s); v \notin \text{dom (discovered s)}; s' = \text{discover} u v (s[(\text{pending} := \text{pending} s - \{(u, v)\}])]) \implies \text{on-discover cycc-params} u v s' \leq_n \text{SPEC} (\lambda x. \text{DFS-invar G cycc-params} (s'[(\text{state.more} := x)])) \rightarrow ?I (s'[(\text{state.more} := x)]) \rightarrow \text{is-invar} ?I \text{rule is used with the induction method, and yields cases}

**print-cases**

Our parameterization has only hooked into initialization and back-edges, so only these two cases are non-trivial:

**Case init thus ?case by (simp add: empty-state-def)**

**next**

**case (back-edge s s' u v)**

For proving invariant preservation, we may assume that the invariant holds on the previous state. Interpreting the invariant locale makes available all invariants ever proved into this locale (i.e., the invariants from all loaded libraries, and the ones you proved yourself.).

**then interpret cycc-invar G s by simp**

However, here we do not need them:

**from back-edge show ?case by simp**

**qed (simp-all cong: break-more-cong)**

For technical reasons, invariants are proved in the basic locale, and then transferred to the invariant locale:
lemmas brk-eq-back \equiv i-brk-eq-back THEN make-invar-thm

The above lemma is simple enough to have a short apply-style proof:

lemma (in cycc) i-brk-eq-back-short-proof:
is-invar (\lambda s. break s \leftrightarrow back-edges s \neq \{\})
apply (induct rule: establish-invarI)
apply (simp-all add: cond-def cong: break-more-cong)
apply (simp add: empty-state-def)
done

Now, when we know that the break flag indicates back-edges, we can easily prove correctness, using a lemma from the invariant library:

thm cycle-iff-back-edges
lemma cycc-correct-aux:
asummes NC: \neg cond s
shows break s \leftrightarrow \neg acyclic (E \cap reachable \times UNIV)
proof (cases break s, simp-all)
assume break s
with brk-eq-back have back-edges s \neq {} by simp
with cycle-iff-back-edges have \neg acyclic (edges s) by simp
with acyclic-subset[OF - edges-ss-reachable-edges]
show \neg acyclic (E \cap reachable \times UNIV) by blast
next
assume A: \neg break s
from A brk-eq-back have back-edges s = {} by simp
with cycle-iff-back-edges have acyclic (edges s) by simp
also from A nc-edges-covered[OF NC] have edges s = E \cap reachable \times UNIV
  by simp
finally show acyclic (E \cap reachable \times UNIV) .
qed

Again, we have a short two-line proof:

lemma cycc-correct-aux-short-proof:
asummes NC: \neg cond s
shows break s \leftrightarrow \neg acyclic (E \cap reachable \times UNIV)
using nc-edges-covered[OF NC] brk-eq-back cycle-iff-back-edges
by (auto dest: acyclic-subset[OF - edges-ss-reachable-edges])

end

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

definition cyc-checker-spec G \equiv do {
ASSERT (fb-graph G);
SPEC (\lambda r. r \leftrightarrow \neg acyclic (g-E G \cap ((g-E G)^* \cdot g-V0 G) \times UNIV))
}
\textbf{theorem} cyc-checker-correct: cyc-checker \( G \leq \) cyc-checker-spec \( G \)

\textbf{unfolding} cyc-checker-def cyc-checker-spec-def

\textbf{proof} (refine-vcg le-ASSERTI order-trans[OF DFS.it-dfs-correct], clarsimp-all)

assume \( \text{fb-graph} \ G \)

then interpret \( \text{fb-graph} \ G \).

interpret cycc by unfold-locales

show DFS \( G \) cycc-params by unfold-locales

next

fix \( s \)

assume cyc-invar \( G \) \( s \)

then interpret cyc-invar \( G \) \( s \).

assume \( \neg \text{cycc}\).cond TYPE('b) \( G \) \( s \)

thus break \( s = (\neg \text{acyclic} (g-E \ G \cap \text{cycc}.reachable \text{TYPE}('b) \ G \times \text{UNIV})) \)

by (rule cycc-correct-aux)

qed

The same for the total correct variant:

\textbf{definition} cyc-checkerT-spec \( G \equiv \) do {

ASSERT (graph \( G \) \& finite (graph-defs.reachable \( G \)));

SPEC (\( \lambda \) r. \( r \leftrightarrow \neg \text{acyclic} (g-E \ G \cap \text{cycc}.reachable \text{TYPE}('b) \ G \times \text{UNIV})) \}

\textbf{theorem} cyc-checkerT-correct: cyc-checkerT \( G \leq \) cyc-checkerT-spec \( G \)

\textbf{unfolding} cyc-checkerT-def cyc-checkerT-spec-def

\textbf{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 cyc-invar \( G \) \( s \)

then interpret cyc-invar \( G \) \( s \).

assume \( \neg \text{cycc}\).cond TYPE('b) \( G \) \( s \)

thus break \( s = (\neg \text{acyclic} (g-E \ G \cap \text{cycc}.reachable \text{TYPE}('b) \ G \times \text{UNIV})) \)

by (rule cycc-correct-aux)

qed

\subsection{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 ≡ { (| cycc-state-impl.break = b |, | cycc-state.break = b |) | b. True }
abbreviation cycc-rel ≡ (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
  ≡ dflt-parametrization simple-state.more (RETURN (| break = False |)) (|
  on-back-edge := λ u v s. RETURN (| break = True |),
  is-break := break []
```  

```plaintext
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)∈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)
```

```plaintext
interpretation cycc-impl: simple-impl-defs G cycc-params-impl cycc-params
  for G .
```

The above interpretation creates an iterative and a recursive implementation

```plaintext
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 ≡ do { ASSERT (fb-graph G);
  s ← cycc-impl.tailrec-impl TYPE(‘a) G;
  RETURN (break s) }
```

```plaintext
definition [DFS-code-unfold]: cyc-checker-rec-impl G ≡ do { ASSERT (fb-graph G);
  s ← cycc-impl.rec-impl TYPE(‘a) G;
```
\[\text{return} \ (\text{break } s)\]

**definition [DFS-code-unfold]:**
\[
cyc-checker-implT \ G \equiv \{ \\
do \ \\
\text{assert} \ (\text{graph } G \land \text{finite (graph-defs.reachable } G)); \\
\ s \leftarrow \text{cyc-impl.tailrec-implT TYPE('a) } G; \\
\text{return} \ (\text{break } s) \\
\}\n\]

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-relI, clarsimp simp: simple-state-rel-def, parametricity) []**

**apply (auto simp: cycc-erel-def break-impl simple-state-rel-def)**

**done**

We get that our implementation refines the abstract DFS algorithm.

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

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

**lemma**

\[
cycc-impl.tailrec-impl TYPE('more) \ G \leq \lll \ cycc-rel it-dfs \]

\[
cycc-impl.rec-impl TYPE('more) \ G \leq \lll \ cycc-rel it-dfs \]

\[
cycc-impl.tailrec-implT TYPE('more) \ G \leq \lll \ cycc-rel it-dfsT \]

using impl-refine .

end

Finally, we get correctness of our cyclicity checker implementations

**lemma cyc-checker-impl-refine: cyc-checker-impl G \leq \lll Id (cyc-checker G)**

**unfolding cyc-checker-impl-def cyc-checker-def**

**apply (refine-vcg cyc.impl-refine)**

**apply (simp-all add: break-impl cyccI)**

**done**

**lemma cyc-checker-rec-impl-refine:**
\[
cycc-checker-rec-impl G \leq \lll \Id (cyc-checker G) \]

**unfolding cyc-checker-rec-impl-def cyc-checker-def**

**apply (refine-vcg cyc.impl-refine)**

**apply (simp-all add: break-impl cyccI)**

**done**

**lemma cyc-checker-implT-refine: cyc-checker-implT G \leq \lll Id (cyc-checkerT G)**

**unfolding cyc-checker-implT-def cyc-checkerT-def**

**apply (refine-vcg cyc.impl-refine)**
apply (simp-all add: break-impl cyccl')
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)cyccl-state-impl' = ('si,'nsi)simple-state-impl +
break-impl :: bool

We define the refinement relation for the state extension

definition [to-relAPP]: cyccl-state-erel erel \equiv \{
    ((break-impl = bi, \ldots = mi),(break = b, \ldots = m)) | bi mi b m, (bi,b)\in bool-rel \land (mi,m)\in erel\}

And register it with the Autoref tool:

consts
i-cyccl-state-ext :: interface \Rightarrow interface

lemmas [autoref-rel-intf] = REL-INTFI[af cyccl-state-erel i-cyccl-state-ext]

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

lemma [autoref-rules]:
fixes ns-rel vis-rel erel
defines R \equiv \langle ns-rel,vis-rel,\langle erel \rangle cyccl-state-erel \rangle ss-impl-rel
shows
(cyccl-state-impl'-ext, cyccl-state-impl-ext) \in bool-rel \rightarrow erel \rightarrow \langle erel \rangle cyccl-state-erel
(break-impl, cyccl-state-impl,break) \in R \rightarrow bool-rel
unfolding cyccl-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 cyccl-checker-impl:
defines V \equiv Id :: ('v \times 'v::hashable) set
assumes [unfolded V-def,autoref-rules]:
(Gi, G) \in \langle Rm, V \rangle g-impl-rel-ext
notes [unfolded V-def,autoref-tyrel] =
TYRELI[where R=(V) dflt-ahs-rel]
TYRELI[where R=(V \times_r (V) list-set-rel) ras-rel]
sshows nres-of (?c::'?c dres) \leq \Downarrow R (cyccl-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:**

- **Assumption 1:** \( fb-graph \ G \)
- **Assumption 2:** \( (G_i, G) \in \langle Rm, Id \rangle g\text{-impl-rel-ext} \)
- **Assumption 4:** \( cyc\text{-checker-code} G_i \equiv d\text{RETURN} x \)

**Shows:** \( x \iff \neg \text{acyclic} (g\text{-E} G \cap ((g\text{-E} G)\ast 'g\text{-V}0 G) \times UNIV)) \)

**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

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

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

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

- **Assumptions:** [unfolded V-def, autoref-rules]:
  - \( (G_i, G) \in \langle Rm, V \rangle g\text{-impl-rel-ext} \)

- **Notes:** [unfolded V-def, autoref-tyrel] =
  - \( \text{TYRELI}[\text{where } R=(V)d\text{flt-ahs-rel}] \)
  - \( \text{TYRELI}[\text{where } R=(V \times_r (V)\text{-list-set-rel})r\text{as-rel}] \)

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

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-impl

prepare-code-thms cyc-checker-rec-code-def

export-code cyc-checker-rec-code checking SML

**Lemma cyc-checker-rec-code-correct:**

- **Assumption 1:** \( fb-graph \ G \)
- **Assumption 2:** \( (G_i, G) \in \langle Rm, Id \rangle g\text{-impl-rel-ext} \)
- **Assumption 4:** \( cyc\text{-checker-rec-code} G_i \equiv d\text{RETURN} x \)

**Shows:** \( x \iff \neg \text{acyclic} (g\text{-E} G \cap ((g\text{-E} G)\ast 'g\text{-V}0 G) \times UNIV)) \)

**Proof:**
- **Note:** cyc-checker-rec-code.refine[OF 2]
- Also note cyc-checker-rec-impl-refine
- Also note cyc-checker-correct
- Finally show ?thesis using 1 4
unfolding cyc-checker-spec-def by auto

qed
And, again, for the total correct version. Note that we generate a plain
implementation, not inside a monad:
schematic-goal cyc-checker-implT:
defines V ≡ 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]
notes [unfolded V-def, autoref-tyrel] =
  TYRELI[where R={V ×, (V)list-set-rel}ras-rel]
shows RETURN (?c::'?c') ≤⇓ R (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-defs.reachable G)
  assumes 2: (Gi, G) ∈ ⟨Rm, Id⟩g-impl-rel-ext
  shows cyc-checker-codeT Gi ←→ (¬acyclic (g-E G ∩ ((g-E G)∗ (g-V0 G) × UNIV))
proof –
  note cyc-checker-codeT.refine[OF 2]
  also note cyc-checker-implT-refine
  also note cyc-checkerT-correct
  finally show ?thesis using 1
unfolding cyc-checkerT-spec-def by auto
qed

end

2.2 Finding a Path between Nodes

theory DFS-Find-Path
imports
  ../DFS-Framework
  CAVA-Automata.Digraph-Impl
  ../Misc/Impl-Rev-Array-Stack
begin

We instantiate the DFS framework to find a path to some reachable node
that satisfies a given predicate. We present four variants of the algorithm:
Finding any path, and finding path of at least length one, combined with
searching the whole graph, and searching the graph restricted to a given

set of nodes. The restricted variants are efficiently implemented by pre-initializing the visited set (cf. DFS-Framework.Restr-Impl).

The restricted variants can be used for incremental search, ignoring already searched nodes in further searches. This is required, e.g., for the inner search of nested DFS (Buchi automaton emptiness check).

### 2.2.1 Including empty Path

**record** \(''v fp0-state = 'v state + ppath :: (''v list × ''v) option\)

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

**lemma** [simp]: \(s() state\_more := [] ppath = foo [] \) = \(s() ppath := foo []\)

**by** (cases s) simp

**abbreviation** no-path ≡ \([] ppath = None []\)

**abbreviation** a-path p v ≡ \(Some (p,v)[]\)

**definition** fp0-params :: \(\forall v \Rightarrow bool \Rightarrow ''v fp0-param\)

where

- on-init = RETURN no-path,
- on-new-root = \(\lambda v0 s. if P v0 then RETURN (a-path [] v0) else RETURN no-path,\)
- on-discover = \(\lambda u v s. if P v then — v is already on the stack, so we need to pop it again\)

  - RETURN (a-path (rev (tl (stack s))) v)

  - else RETURN no-path,
- on-finish = \(\lambda u s. RETURN (state\_more s),\)
- on-back-edge = \(\lambda u v s. RETURN (state\_more s),\)
- on-cross-edge = \(\lambda u v s. RETURN (state\_more s),\)
- is-break = \(\lambda s. ppath s \neq None []\)

**lemmas** fp0-params-simps[simp] = gen-parameterization.simps[mk-record-simp, OF fp0-params-def]

**interpretation** fp0: param-DFS-defs where param = fp0-params P for G P .

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

begin

**lemma** [simp]:

- ppath (empty-state (ppath = e[])) = e
  
  **by** (simp add: empty-state-def)

**lemma** [simp]:

- ppath (s(state\_more := state\_more s[])) = ppath s'
by \((\text{cases } s, \text{ cases } s')\) 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 \(\lambda s. \text{ppath } s = \text{None} \rightarrow \text{dom (discovered } s) \cap \text{Collect } P = \{\}\)
by (rule establish-invarI) simp-all

lemma i-path-to-P:
  is-invar \(\lambda s. \text{ppath } s = \text{Some (vs,v)} \rightarrow P v\)
by (rule establish-invarI) auto

lemma i-path-invar:
  is-invar \(\lambda s. \text{ppath } s = \text{Some (vs,v)} \rightarrow\)
  \[ (\text{vs } \neq [] \rightarrow \text{hd } vs \in V0 \land \text{path } E \text{ (hd } vs \text{ vs } v) \]
  \land \(\text{vs } = [] \rightarrow v \in V0 \land \text{path } E v vs v\)
  \land (\text{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: \(v \notin \text{set (stack } s)\) using stack-discovered by auto

end
from pendingD discover have $v \in \text{succ}(\text{hd}(\text{stack}\ s))$ by simp
with $\text{hd-succ-stack-is-path}[\text{OF ne}]$ have $\exists v_0 \in V_0. \text{path} E v_0 (\text{rev}(\text{stack}\ s)) v$.

moreover from last-stack-in-V0 ne have last (stack\ s) $\in V_0$ by simp
ultimately have path E (hd (rev (stack\ s))) (rev (stack\ s)) v hd (rev (stack\ s)) $\in V_0$
using $\text{hd-rev}[\text{OF ne}]$ path-hd[where $p=\text{rev}(\text{stack}\ s)$] ne
by auto
with ne discover vnis show ?case by (auto simp: stack-distinct)
qed auto
end

context fp0-invar
begin
lemmas no-path-no-P-discovered
  = i-no-path-no-P-discovered[\text{THEN make-invar-thm}, \text{rule-format}]

lemmas path-to-P
  = i-path-to-P[\text{THEN make-invar-thm}, \text{rule-format}]

lemmas path-invar
  = i-path-invar[\text{THEN make-invar-thm}, \text{rule-format}]

lemma path-invar-nonempty:
  assumes $\text{ppath}\ s = \text{Some}\ (vs, v)$
  and $vs \neq []$
  shows $\text{hd}\ vs \in V_0\ \text{path}\ E\ (\text{hd}\ vs)\ vs\ v$
  using assms path-invar
  by auto

lemma path-invar-empty:
  assumes $\text{ppath}\ s = \text{Some}\ (vs, v)$
  and $vs = []$
  shows $v \in V_0\ \text{path}\ E\ v\ vs\ v$
  using assms path-invar
  by auto

lemma fp0-correct:
  assumes $\neg\text{cond}\ s$
  shows case ppath\ s of
  None $\Rightarrow \neg(\exists v_0 \in V_0. \exists v. (v_0, v) \in E^* \land P v)$
  | Some (p, v) $\Rightarrow (\exists v_0 \in V_0. \text{path}\ E\ v_0\ p\ v\ \land\ P\ v\ \land\ \text{distinct}\ (p@v))$
proof (cases ppath\ s)
  case None with assms ne-discovered-eq-reaching ne-path-no-P-discovered have
  reachable $\cap\ \text{Collect}\ P = \{\}$ by auto
  thus ?thesis by (auto simp add: None)
next
  case (Some vs) then obtain v vs where [simp]: vs = (vs, v)
by (cases vv) auto

from Some path-invar[of vs v] path-to-[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 p 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 ⊆ R ∧ 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 \implies E^* ∩ R ∩ Collect P = {}
  unfolding restr-invar-def by (simp add: Image-closed-trancl)

type-synonym 'a fpr-result = 'a set + ('a list × 'a)
definition find-path0-restr-pred G P R ≡ λ r.
  case r of
    Inl R' ⇒ R' = R ∪ (g-E G)^* " g-V0 G ∧ restr-invar (g-E G) R' P
  | Inr (vs, v) ⇒ P v ∧ (\exists 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
  :: ('a, 'more) graph-rec-scheme ⇒ ('a ⇒ bool) ⇒ 'a fpr-result ⇒ 'a ⇒ bool
where
  find-path0-restr G P R ≡ do { 
    ASSERT (fb-graph G);
    ASSERT (fp0 (graph-restrict G R));
\[ s \leftarrow \text{fp0.it-dfs TYPE('more) (graph-restrict G R) P} \]

\begin{align*}
\text{case ppath s of} \\
\text{None } & \Rightarrow \text{do} \{ \\
& \text{ASSERT (dom (discovered s) = dom (finished s))} \\
& \text{RETURN (Inl (R \cup \text{dom (finished s)))} \\
\} \\
| \text{Some (vs,v)} & \Rightarrow \text{RETURN (Inr (vs,v))}
\end{align*}

\textbf{lemma \textit{find-path0-restr-correct}:}
\textit{shows} \text{find-path0-restr G P R} \leq \text{find-path0-restr-spec G P R}
\textbf{proof} (\text{rule le-ASSERT-defI1[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 \text{(rule a.fb-graph-restrict)}}
\text{assume I: restr-invar (g-E G) R P}
\text{define reachable where reachable = graph-defsreachable (graph-restrict G R)}
\text{interpret fp0 graph-restrict G R by unfold-locales}
\text{show ?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 fp0-invar (graph-restrict G R) P s}
\text{and NC[\text{simp}]: \neg 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) = dom (finished s) by simp}
\text{from nc-finished-eq-reachable}
\text{have DFR[\text{simp}]: dom (finished s) = reachable}
\text{by (simp add: reachable-def)}
\text{from I have g-E G \text{" R \subseteq R unfolding restr-invar-def by auto}}
\text{have reachable \subseteq (g-E G)* \text{" g-V0 G}
\text{unfolding reachable-def}
\text{by (rule Image-mono, rule rtrancl-mono) (auto simp: rel-restrict-def)}
\]
hence $R \cup \text{dom} \ (\text{finished } s) = R \cup (g-E \ G)^* \ \
apply \ - \ 
apply \ (\text{rule equalityI}) \ 
apply \ \text{auto } [] 
\text{unfolding} \ DFR \ \text{reachable-def} 
\text{apply} \ (\text{auto } \text{elim: } \text{E-closed-restr-cases}[\text{OF } \langle g-E \ G \ \subseteq R \ | \ \rangle] \ []) \ 
\text{done} 
\text{moreover from } \text{nc-fin-closed I} 
\text{have} \ g-E \ G \ \
apply \ \text{restr-invar-def} \ \text{by} \ \text{(simp add: } \text{rel-restrict-def}) \ blast 
\text{moreover from } \text{no-path-no-P-discovered nc-discovered-eg-finished I} 
\text{have} \ (R \cup \text{dom} \ (\text{finished } s)) \cap \text{Collect } P = {} 
\text{unfolding} \ \text{restr-invar-def} \ \text{by} \ \text{auto} 
\text{ultimately} 
\text{show} \ \text{find-path0-restr-pred } G \ P \ R \ (\text{Inl } (R \cup \text{dom} \ (\text{finished } s))) 
\text{unfolding} \ \text{restr-invar-def} \ \text{find-path0-restr-pred-def } \ \text{by} \ \text{auto} 
\} 
\} 
\text{fix } v \ vs 
\text{assume } [\text{simp}]: \ ppath \ s = \text{Some } (vs,v) 
\text{from } \text{fp0-correct} 
\text{show} \ \text{find-path0-restr-pred } G \ P \ R \ (\text{Inr } (vs, v)) 
\text{unfolding} \ \text{find-path0-restr-pred-def } \ \text{by} \ \text{auto} 
\} 
\text{qed} 
\text{qed} 

\textbf{2.2.3 Path of Minimal Length One, with Restriction} 
\textbf{definition} \text{find-path1-restr-pred } G \ P \ R \equiv \lambda r. 
\text{case } r \ of 
\text{Inl } R' \Rightarrow R' = R \cup (g-E \ G)^* \ g-V0 \ G \ \land \ \text{restr-invar-def } (g-E \ G) \ R' \ P 
| \ \text{Inr } (vs,v) \Rightarrow P \ v \ \land \ vs \neq [] \ \land \ (\exists \ v'0 \in g-V0 \ G. \ \text{path} \ (g-E \ G \ \cap \ \text{UNIV } \times \ -R) \ v'0 \ vs \ v) 
\textbf{definition} \text{find-path1-restr-spec} 
\text{— 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.} 
\textbf{where} \text{find-path1-restr-spec } G \ P \ R \equiv \text{do } \{ 
\text{ASSERT } (\text{fl-graph } G \ \land \ \text{restr-invar-def } (g-E \ G) \ R \ P); 
\text{SPEC } (\text{find-path1-restr-pred } G \ P \ R)\} 
\textbf{lemmas} \text{find-path1-restr-spec-rule[refine-vcg] } = 
\text{ASSERT-le-defI}[\text{OF } \text{find-path1-restr-spec-def}] 
\text{ASSERT-lef-defI}[\text{OF } \text{find-path1-restr-spec-def}] 
\textbf{definition} \text{find-path1-restr} 
\vdash (\forall v, \ 'more') \ \text{graph-rec-scheme } \Rightarrow (\forall v \Rightarrow \text{bool}) \ \Rightarrow \ 'v \ \text{set } \Rightarrow \ 'v \ \text{fpr-result nres}
where find-path1-restr \( G P R \equiv \)
FOREACH\(c (g\cdot V0 G) \in\text{Inl} (\lambda v0 s . \text{do }\{\) 
\hspace{1cm} ASSERT (\in\text{Inl} s); — TODO: Add FOREACH-condition as precondition in autoref!
\hspace{1cm} let R = projl s;
\hspace{1cm} \textbf{f0} \leftarrow \text{find-path0-restr-spec} (G \{ g\cdot V0 := g\cdot E G \mapsto \{v0\}\} P R);
\hspace{1cm} case f0 of
\hspace{2cm} \textbf{Inl} - \Rightarrow RETURN f0
\hspace{2cm} | \textbf{Inr} (vs,v) \Rightarrow RETURN (\textbf{Inr} (v0#vs,v))
\}) (\textbf{Inl} R)

**definition** find-path1-tailrec-invar \( G P R0 it s \equiv \)
\hspace{1cm} case s of
\hspace{2cm} \textbf{Inl} R \Rightarrow R = R0 \cup (g\cdot E G)^+ \mapsto \{v0\} \land \text{restr-invar} (g\cdot E G) R P
\hspace{2cm} | \textbf{Inr} (vs,v) \Rightarrow P v \land vs \neq [] \land (\exists v0 \in g\cdot V0 G - it. \text{path} (g\cdot E G \cap \text{UNIV} \times -R0) v0 vs v)

**lemma** find-path1-restr-correct:
shows find-path1-restr \( G P R \leq \text{find-path1-restr-spec} G P R\)
proof (rule le-ASSERT-defI1 [OF find-path1-restr-spec-def], clarify)
assume fb-graph G
interpret a: fb-graph G by fact
interpret fb0: fb-graph G \{ g\cdot E := g\cdot E G \cap \text{UNIV} \times -R \}
by (rule a.fb-graph-subset, auto)

assume I: restr-invar (g\cdot E G) R P

have aux2: \( \forall v0. v0 \in g\cdot V0 G \Longrightarrow \text{fb-graph} (G \{ g\cdot V0 := g\cdot E G \mapsto \{v0\}\}) \)
by (rule a.fb-graph-subset, auto)

\{ 
  \hspace{1cm} fix v0 it s
  \hspace{1cm} assume IT: it \subseteq g\cdot V0 G v0 \in it
  \hspace{1cm} and is-Inl s
  \hspace{1cm} and FPI: find-path1-tailrec-invar \( G P R it s \)
  \hspace{1cm} and RI: restr-invar (g\cdot E G) (projl s \cup (g\cdot E G)^+ \mapsto \{v0\}) P

  \hspace{1cm} then obtain R’ where [simp]: s = Inl R’ by (cases s) auto

from FPI have [simp]: R’ = R \cup (g\cdot E G)^+ \mapsto \{v0\} - it
unfolding find-path1-tailrec-invar-def by simp

have find-path1-tailrec-invar \( G P R (it \setminus \{v0\}) \)
  \hspace{1cm} (Inl (projl s \cup (g\cdot E G)^+ \mapsto \{v0\})))
using RI
by (auto simp: find-path1-tailrec-invar-def it-step-insert-iff[OF IT])
\} note aux4 = this
\{ 
  \textbf{fix} \ v0 \ u \ it \ s \ v \ p 
  \textbf{assume} \ IT: \ it \subseteq g-V0 \ G \ v0 \in it 
  \textbf{and} \ is-\text{Inl} \ s 
  \textbf{and} \ FPI: \ \text{find-path1-tailrec-invar} \ G \ P \ R \ it \ s 
  \textbf{and} \ PV: \ P \ v 
  \textbf{and} \ PATH: \ \text{path} \ (\text{rel-restrict} \ (g-E \ G) \ (\text{projl} \ s)) \ u \ p \ v \ (v0,u) \in (g-E \ G) 
  \textbf{and} \ PR: \ u \notin \text{projl} \ s 

  \text{then obtain} \ R' \ \text{where} \ [\text{simp}]: \ s = \text{Inl} \ R' \ \text{by} \ (\text{cases} \ s) \ \text{auto}

  \text{from} \ FPI \ \text{have} \ [\text{simp}]: \ R' = R \cup (g-E \ G)^+ \ q \ (g-V0 \ G - it) 

  \text{unfolding} \ \text{find-path1-tailrec-invar-def} \ \text{by} \ \text{simp}

  \text{have} \ \text{find-path1-tailrec-invar} \ G \ P \ R \ (it - \{v0\}) \ (\text{Inl} \ (v0 \neq p, v)) 

  \text{apply} \ (\text{simp add: find-path1-tailrec-invar-def PV}) 

  \text{apply} \ (\text{rule bexI[where x=v0]}) 

  \text{using} \ PR \ PATH(2) \ \text{path-mono[OF rel-restrict-mono2[of R] PATH(1)]}

  \text{apply} \ (\text{auto simp: path1-restr-conv}) 

  \text{using} \ IT \ \text{apply} \ \text{blast}

  \text{done}
\}
\text{note} \ aux5 = \text{this}

\text{show} \ ?\text{thesis}

\text{unfolding} \ \text{find-path1-restr-def} \ \text{find-path1-restr-spec-def} \ \text{find-path1-restr-pred-def}

\text{apply} \ (\text{refine-rcg 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} \ (\text{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}

\textbf{definition} \ \text{find-path1-pred} \ G \ P \equiv \lambda r. 
\textbf{case} \ r \ \text{of}
  \text{None} \Rightarrow (g-E \ G)^+ \ q \ g-V0 \ G \cap \text{Collect} \ P = \{\}
  \mid \text{Some} \ (vs, v) \Rightarrow P \ v \wedge vs \neq \emptyset \wedge (\exists \ v0 \in g-V0 \ G. \ \text{path} \ (g-E \ G) \ v0 \ vs \ v)

\textbf{definition} \ \text{find-path1-spec}
— Find a path of length at least one to a target node that satisfies a given predicate.

**where** find-path1-spec $G\ P \equiv \text{do}
\begin{align*}
&\text{ASSERT (fb-graph G);} \\
&\text{SPEC (find-path1-pred G P)}
\end{align*}

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

### 2.2.4 Path of Minimal Length One, without Restriction

**definition** find-path1 $:: (\text{'v, 'more})\ graph-rec-scheme \Rightarrow (\text{'v \Rightarrow bool}) \Rightarrow \text{'v}\ \text{fp-result}\ \text{nres}$

**where** find-path1 $G\ P \equiv \text{do}
\begin{align*}
r &\leftarrow \text{find-path1-restr-spec G P} \\
\text{case } r \text{ of}
\begin{cases}
  \text{Inl } - &\Rightarrow \text{RETURN None} \\
  \text{Inr } vsv &\Rightarrow \text{RETURN (Some vsv)}
\end{cases}
\end{align*}

**lemma** find-path1-correct $:\text{shows find-path1 G P} \leq \text{find-path1-spec G P}$

**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: sam.splits
  dest!: restr-invar-imp-not-reachable tranclD)

**done**

### 2.2.5 Implementation

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

**definition** fp0-erel $\equiv \{\}
\text{fp0-state-impl}.\text{ppath} = p \}\ \text{fp0-state}.\text{ppath} = p\}$

**abbreviation** fp0-rel $R \equiv \text{(fp0-erel) restr-simple-state-rel R}$

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

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

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

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

**unfolding** restr-simple-state-rel-def fp0-erel-def

**by** (cases s, cases s', auto)
lemma \textbf{fp0-ss-rel-ppath-cong}[simp]:
\[(s, s') \in \langle \text{fp0-erel} \rangle \Rightarrow \text{simple-state-rel} \implies \text{fp0-state-impl}.\text{ppath} s = \text{fp0-state-impl}.\text{ppath} s'
\]
unfolding \text{simple-state-rel-def} \text{fp0-erel-def} 
by \text{(cases s, cases s', auto)}

lemma \textbf{fp0i-cong}[cong]:
\[\text{simple-state}.\text{more} s = \text{simple-state}.\text{more} s' \Rightarrow \text{fp0-state-impl}.\text{ppath} s = \text{fp0-state-impl}.\text{ppath} s'\]
by \text{(cases s, cases s', auto)}

lemma \textbf{fp0-erelI}:
\[p = p' \Rightarrow ((\langle \text{fp0-state-impl} . \text{ppath} = p \rangle, \langle \text{fp0-state-impl} . \text{ppath} = p' \rangle) \in \text{fp0-erel})\]
unfolding \text{fp0-erel-def} by auto

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

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

interpretation \textbf{fp0-impl}:
\text{restricted-impl-defs} \text{fp0-params-impl} P \text{fp0-params} P G R 
for G P R .

locale \textbf{fp0-restr} = \text{fb-graph}
begin
sublocale \textbf{fp0}?: \text{fp0 graph-restrict} G R
apply (rule \text{fp0I})
apply (rule \text{fb-graph-restrict})
done

sublocale \textbf{impl}: \text{restricted-impl} G \text{fp0-params} P \text{fp0-params-impl} P 
\text{fp0-erel} R
apply unfold-locals
apply parametricity

apply (simp add: \text{fp0-erel-def})
apply (auto) [1]

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 ≤⇓ (∀ 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-drel-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 ≤⇓ (∀ find-path0 G P)
proof (rule refine-ASSERT-defI1[OF find-path0-def])
  assume fp0 G then interpret fp0 G.

interpret $r$: fp0-restr $G$ by unfold-locales

show ?thesis
    unfolding find-path0-impl-def find-path0-def
    apply (refine-reg r.impl.tailrec-refine[where $R=\{\}$, simplified])
    apply (auto)
    done
qed

2.2.6 Synthesis of Executable Code

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

definition [to-relAPP]:
  fp0-state-erel erel ≡ {
    (ppath-impl = pi, \ldots = mi, (ppath = p, \ldots = m)) | pi mi p m.
    (pi, p) ∈ (\langle list-rel × \text{Id} \rangle \text{option-rel} ∧ (mi, m) ∈ erel)
  }

consts
  i-fp0-state-ext :: interface ⇒ interface

lemmas [autoref-rel-intf] = REL-INTFI[of fp0-state-erel i-fp0-state-ext]

term fp0-state-impl-ext

lemma [autoref-rules]:
    fixes ns-rel vis-rel erel
    defines $R ≡ \langle ns-rel, vis-rel, \langle erel, fp0-state-erel \rangle ssnos-impl-rel$ shows
      (fp0-state-impl'-ext, fp0-state-impl-ext) ∈ (\langle list-rel × \text{Id} \rangle \text{option-rel} → erel → \langle erel, fp0-state-erel \rangle ssnos-impl-rel
      (ppath-impl, fp0-state-impl, ppath) ∈ $R → (\langle list-rel × \text{Id} \rangle \text{option-rel} → \langle erel, fp0-state-erel \rangle ssnos-impl-rel
    unfolding fp0-state-erel-def ssnos-impl-rel-def $R$-def
    by auto

schematic-goal find-path0-code:
    fixes $G :: ('v :: hashable, -) graph-rec-scheme$
    assumes [autoref-rules]:
      (Gi, G) ∈ (\langle Rm, \text{Id} \rangle g-impl-rel-ext
      (Pi, P) ∈ \text{Id} → \text{bool-rel}$
    notes [autoref-tyrel] = TYREL[where $R=\langle \text{Id}; ('v × 'v) set \rangle dflt-abs-rel]
    shows (nres-of (?c; (?c' dres), find-path0-impl G P) ∈ ?R
    unfolding find-path0-impl-def[abs-def] DFS-code-unfold ssnos-unfolds
    unfolding if-cancel not-not comp-def nres-monad-laws
    using [[autoref-trace-failed-id]]
    apply (autoref-monadic (trace))
    done

concrete-definition find-path0-code uses find-path0-code
export-code find-path0-code checking SML

lemma find-path0-autoref-aux:
assumes Vid: Rv = (Id :: 'a :: hashable rel)
shows (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.spec[refine[param-fo, THEN nres-relD]],
  assumption+]
) using find-path0-impl find-path0-correct
apply (simp add: pw-le-iff refine-pw-simps)
apply blast
done
lemmas find-path0-autref[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) ∈ ⟨⟨⟨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
assumes Vid: Rv = Id
shows (λ G P R. nres-of (find-path0-restr-code op-vis-insert op-vis-memb G P R),
  find-path0-restr-spec)
  ∈ (⟨Rm, Rv⟩g-impl-rel-ext → (Rv → bool-rel) → (Rv) vis-rel →
  ((⟨Rv⟩ vis-rel, (Rv)list-rel ×, Rv) sum-rel) nres-rel
apply (intro fun-relI nres-relI)
unfolding Vid
apply (rule
  order-trans[OF find-path0-restr-code.refine[OF 1[unfolded Vid] 2[unfolded Vid],
  param-fo, THEN nres-relD])
  )
apply assumption+
using find-path0-restr-impl find-path0-restr-correct
apply (simp add: pu-le-iff refine-pw-simps)
apply blast
done
lemmas find-path0-restr-autoref[autoref-rules] = find-path0-restr-autoref-aux[OF
  GEN-OP-D GEN-OP-D PREFER-id-D]
schematic-goal find-path1-restr-code:
fixes vis-rel :: (′v×′v) set ⇒ (′visi×′v set) set
notes [autoref-rel-intf] = REL-INTFI[of vis-rel i-set for I]
assumes [autoref-rules]: (op-vis-insert, insert)∈Id → ⟨Id⟩ vis-rel → ⟨Id⟩ vis-rel
assumes [autoref-rules]: (op-vis-memb, (∈))∈Id → ⟨Id⟩ vis-rel → bool-rel
assumes [autoref-rules]:
  (Gi, G) ∈ (⟨Rm, Id⟩g-impl-rel-ext
  (Pi,P)∈Id → bool-rel
  (Ri,R)∈⟨Id⟩ vis-rel
shows (nres-of ?c,find-path1-restr G P R)
  ∈ ((⟨Id⟩ vis-rel, ⟨Id⟩ list-rel ×, Id) sum-rel) nres-rel
unfolding find-path1-restr-def[abs-def]
using [[autoref-trace-failed-id]]
apply (autoref-monadic (trace))
done
concrete-definition find-path1-restr-code uses find-path1-restr-code
export-code find-path1-restr-code checking SML

lemma find-path1-restr-autoref-aux:
  assumes G: (op-vis-insert, insert)∈V → ⟨V⟩ vis-rel → ⟨V⟩ vis-rel
  (op-vis-memb, (∈))∈V → ⟨V⟩ vis-rel → bool-rel
  assumes Vid[simp]: V = Id
  shows (λ G P R. nres-of (find-path1-restr-code op-vis-insert op-vis-memb G P R).find-path1-restr-spec)
  ∈ (⟨Rm, V⟩g-impl-rel-ext → (V → bool-rel) → (V) vis-rel →
  ((⟨V⟩ vis-rel, (V) list-rel ×, V) sum-rel) 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 $E$: ($e,\{\}$)$\in\langle R \rangle Rs$
  assumes $I$: ($i,\text{insert}$)$\in\langle R \rangle Rs \rightarrow \langle R \rangle Rs$
  assumes $EE$: $\forall x. x \in S \implies \exists xi. (xi,x) \in R$
  shows $\exists Si. (Si,S) \in \langle R \rangle Rs$
proof
  define $S'$ where $S' = S$
  have $S \subseteq S'$ by (simp add: $S'$-def)
  from $F$ this show $\exists Si. (Si,S) \in \langle R \rangle Rs$
proof (induction)
  case empty thus ?case
    using $E$ by (blast)
next
  case (insert $x$ $S$)
  then obtain $xi Si$ where $1$: ($Si,S) \in \langle R \rangle Rs$ and $2$: ($xi,x) \in R$
    using $EE$ unfolding $S'$-def by blast
  from $I[\text{THEN fan-relD}, \text{OF 2, THEN fan-relD, OF 1}]$ show ?case ..
qed
qed
lemma obtain-finite-ahs: finite $S \implies \exists x. (x,S) \in (Id)\ dflt-ahs-rel$
  apply (erule gen-obtain-finite-set)
  apply autoref
  apply autoref
  by blast

2.3.2 Framework Instantiation

definition unit-parametrization ≡ dflt-parametrization $(\lambda\cdot () \ (RETURN ()))$

lemmas unit-parametrization-simp[simp, DFS-code-unfold] =
  dflt-parametrization-simp[mk-record-simp, OF, OF unit-parametrization-def]

interpretation unit-dfs: param-DFS-defs where param=unit-parametrization for $G$.

locale unit-DFS = param-DFS $G$ unit-parametrization for $G::(\'v, 'more\) graph-rec-scheme
begin
  sublocale DFS $G$ unit-parametrization
  by unfold-locales simp-all
end

lemma unit-DFSI[Pure.intro?, intro?):
  assumes fb-graph $G$
  shows unit-DFS $G$
proof --
  interpret fb-graph $G$ by fact
  show ?thesis by unfold-locales
qed

definition find-reachable $G \equiv do$
  ASSERT (fb-graph $G$);
  $s \leftarrow$ unit-dfs.it-dfs TYPE('a) $G$;
  RETURN (dom (discovered $s$))

definition find-reachableT $G \equiv do$
  ASSERT (fb-graph $G$);
  $s \leftarrow$ unit-dfs.it-seqT TYPE('a) $G$;
  RETURN (dom (discovered $s$))

2.3.3 Correctness

context unit-DFS begin
  lemma find-reachable-correct: find-reachable $G \leq SPEC (\lambda r. r = reachable)$
  unfolding find-reachable-def

149
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 \implies find-reachableT G \leq SPEC (\lambda r. r = reachable)
unfolding find-reachableT-def
apply (refine-vcg order-trans[of it-dfsT-correct])
apply unfold-locales
apply clarify
apply (drule (1) DFS-invar.nc-discovered-eq-reachable)
by auto
end

context unit-DFS begin

sublocale simple-impl G unit-parametrization unit-parametrization unit-rel
apply unfold-locales
apply (clarsimp simp: simple-state-rel-def) []
by auto

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

interpretation unit-simple-impl:
simple-impl-defs G unit-parametrization unit-parametrization
for G .
term unit-simple-impl.tailrec-impl term unit-simple-impl.rec-impl

definition [DFS-code-unfold]: find-reachable-impl G \equiv do { 
  ASSERT (fb-graph G);
  s \leftarrow unit-simple-impl.tailrec-impl TYPE('a) G;
  RETURN (simple-state.visited s)
}
definition [DFS-code-unfold]: find-reachable-implT G \equiv do { 
  ASSERT (fb-graph G);
  s \leftarrow unit-simple-impl.tailrec-implT TYPE('a) G;
  RETURN (simple-state.visited s)
}
definition [DFS-code-unfold]: find-reachable-rec-impl G \equiv do { 
  ASSERT (fb-graph G);
  s \leftarrow unit-simple-impl.rec-impl TYPE('a) G;
  RETURN (visited s)
}
lemma find-reachable-impl-refine:
find-reachable-impl G ≤ ⇓ Id (find-reachable G)
unfolding find-reachable-impl-def find-reachable-def
apply (refine-vcg unit-DFS.impl-refine)
apply (simp-all add: unit-DFSI simple-state-rel-def)
done

lemma find-reachable-implT-refine:
find-reachable-implT G ≤ ⇓ Id (find-reachableT G)
unfolding find-reachable-implT-def find-reachableT-def
apply (refine-vcg unit-DFS.impl-refine)
apply (simp-all add: unit-DFSI simple-state-rel-def)
done

lemma find-reachable-rec-impl-refine:
find-reachable-rec-impl G ≤ ⇓ Id (find-reachable G)
unfolding find-reachable-rec-impl-def find-reachable-def
apply (refine-vcg unit-DFS.impl-refine)
apply (simp-all add: unit-DFSI simple-state-rel-def)
done

2.3.4 Synthesis of Executable Implementation

schematic-goal find-reachable-impl:
defines V ≡ 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]
sshows 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 × '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
notes [autoref-tyrel] =
TYRELI[where R=⟨V⟩\ahs-rel bhc]
TYRELI[where R=⟨V ×r (V)\list-set-rel\ras-rel⟩]
sends RETURN (?c::?c) ≤⇓?R (find-reachable-implT G)
unfolding if-cancel DFS-code-unfold ssnos-unfolds
using [[[autoref-trace-failed-id, goals-limit=1]]]
apply (autoref-monadic (plain,trace))
done
concrete-definition find-reachable-codeT for eq bhc sz Gi
uses find-reachable-implT
export-code find-reachable-codeT checking SML

lemma find-reachable-codeT-correct:
fixes V :: ('vi × 'v) set
assumes G: graph G
assumes FR: finite ((g-E G)∗ "g-V0 G)
assumes BHC: is-bounded-hashcode V eq bhc
assumes EQ: (eq,(=)) ∈ V → 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_refs[OF BHC EQ VDS 2]
also note find-reachable-implT-refine
also note find-reachable-implT-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

152
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

→(R)unit-list-rel→(R)unit-list-rel

by auto

schematic-goal find-reachable-rec-impl:

defines V ≡ Id :: (v × 'v::hashable) set

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

(Gi, G) ∈ (Rm, V)g-impl-rel-ext

notes [unfolded V-def,autoref-tyrel] = TYRELI |where R=⟨V⟩dflt-ahs-rel|

shows mres-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 (tracce))

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]
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 2: (Gi, G) ∈ ⟨Rm, V⟩ g-impl-rel-ext
  shows (find-reachable-codeT eq bhc sz Gi,
    (OP op-reachable :: ⟨Rm, V⟩ g-impl-rel-ext → ⟨V⟩ ahs-rel bhc)∈⟨V⟩ ahs-rel bhc
  using assms
  by (simp add: find-reachable-codeT-correct)
end

2.3.5 Conclusions

We have defined an efficient DFS-based implementation for op-reachable, and declared it to Autoref.
end

2.4 Find a Feedback Arc Set

theory Feedback-Arcs
imports
  ../DFS-Framework
  CAVA-Automata.Digraph-Impl
  Reachable-Nodes
begin
A feedback arc set is a set of edges that breaks all reachable cycles. In this theory, we define an algorithm to find a feedback arc set.

definition is-fas :: (′v, ′more) graph-rec-scheme ⇒ ′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)∗
ultimately show False by blast

next

fix u v0

assume 1: v0 ∈ g-V0 G and 2: (v0, u) ∈ (g-E G)∗ and 3: (u, u) ∈ (g-E G − EC)∗

have (u, u) ∈ (Restr (g-E G − EC) ((g-E G)∗ “ g-V0 G))∗

apply (rule trancl-restrict-reachable[OF 3, where S = (g-E G)∗ “ g-V0 G])

apply (rule order-trans[OF rtrancl-image-unfold-right])

using 1 2 by auto

hence (u, u) ∈ (g-E G ∩ (g-E G)∗ “ g-V0 G × UNIV − EC)∗

by (rule trancl-mono) auto

moreover assume ∀ x. (x, x) /∈ (g-E G ∩ (g-E G)∗ “ g-V0 G × UNIV − EC)∗

ultimately show False by blast

qed

2.4.1 Instantiation of the DFS-Framework

record ’v fas-state = ’v state +
fas :: (’v × ’v) set

lemma fas-more-cong: state.more s = state.more s’ ⇒ fas s = fas s’

by (cases s, cases s’, simp)

lemma [simp]: s[] state.more := () fas = foo [] s[] fas := foo []

by (cases s) simp

definition fas-params :: (’v, (’v, unit) fas-state-ext) parameterization

where fas-params ≡ dflt-parametrization state.more

(RETURN () fas = {}) []

on-back-edge := λ u v s. RETURN () fas = insert (u,v) (fas s) []

[]

lemmas fas-params-simp[simp] =

gen-parameterization.simps[mk-record-simp, OF fas-params-def[simplified]]

interpretation fas: param-DFS-defs where param=fas-params for G.

Find feedback arc set

definition find-fas G ≡ do {

ASSERT (graph G);

ASSERT (finite ((g-E G)∗ “ g-V0 G));

s ← fas.it-dfsT TYPE ’a G;

RETURN (fas-state.fas s)
}

locale fas =

param-DFS G fas-params

for G :: (’v, ’more) graph-rec-scheme

+ assumes finite-reachable[simp, intro!]: finite ((g-E G)∗ “ g-V0 G)
begin

sublocale DFS G fas-params
  apply unfold-locales
  apply (simp-all add: fas-params-def)
done

end

lemma fasI:
  assumes graph G
  assumes finite ((g-E G)∗ ≺ g-V0 G)
  shows fas G
proof −
  interpret graph G by fact
  interpret fb-graph G by (rule fb-graphI-fr[OF assms(2)])
  show ?thesis by unfold-locales fact
qed

2.4.2 Correctness Proof

locale fas-invar = DFS-invar where
  param = fas-params + fas
begin

lemma (in fas) i-fas-eq-back: is-invar (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 (refine-req le-ASSERTI order-trans[OF DFS.it-dfsT-correct], clarsimp-all)

interpret graph \( \ G \) by fact

assume finite \((g\cdot E \ G)\) \( \cdots \ g\cdot V0 \ G)\)

then interpret \( \text{fb-graph} \ G \) by (rule \( \text{fb-graphI-fr} \))

interpret \( \text{fas} \) by unfold-locales fact

show \( \text{DFS} \ G \) \( \text{fas}-\text{params} \) by unfold-locales

next

fix \( \ s \)

assume \( \text{DFS-invar} \ G \) \( \text{fas}-\text{params} \ s \)

then interpret \( \text{DFS-invar} \ G \) \( \text{fas}-\text{params} \ s \)

interpret \( \text{fas-invar} \ G \) \( \ s \) by unfold-locales fact

assume \( \neg \text{fas} \) \( \text{cond TYPE} \) \((b) \ G \) \( \ s \)

thus \( \text{is-fas} \ G \) \((\text{fas-state} \ . \text{fas} \ s) \)

by (rule \( \text{find-fas-correct-aux} \))

qed (rule \( \text{assms} \) ) +

2.4.3 Implementation

record \( ^'v \) \( \text{fas-state-impl} = ^'v \) \( \text{simple-state} + \)

\( \text{fas} = (^'v \times ^'v) \) set

definition \( \text{fas-erel} \equiv \{

(\langle \text{fas-state-impl} \ . \text{fas} = f \rangle, \langle \text{fas-state} \ . \text{fas} = f \rangle) \ | \ f. \text{True} \}

abbreviation \( \text{fas-rel} \equiv \langle \text{fas-erel} \rangle \text{simple-state-rel} \)

definition \( \text{fas-params-impl} \equiv \langle \text{fas-erel} \rangle \text{simple-state-rel} \)

where \( \text{fas-params-impl} \equiv \text{dflt-parametrization simple-state}.\text{more} \ (\text{RETURN} (\langle \text{fas} = \{\} \rangle) \langle\)

\( \text{on-back-edge} \equiv \lambda ^'v \ s. \text{RETURN} (\langle \text{fas} = \text{insert} (u,v) (\text{fas} s) \rangle) \)

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

lemma \( \text{fas-impl} \) : \((si, s) \in \text{fas-rel} \)

\( \Longrightarrow \text{fas-state-impl}.\text{fas} \ s = \text{fas-state} \ s \)

by (cases si, cases s, simp add: simple-state-rel-def \( \text{fas-erel-def} \) )

interpretation \( \text{fas-impl} : \text{simple-impl-defs} \ G \) \( \text{fas-params-impl} \) \( \text{fas-params} \)

for \( \ G \).

term \( \text{fas-impl}.\text{tailrec-impl} \) term \( \text{fas-impl}.\text{tailrec-implT} \) term \( \text{fas-impl}.\text{rec-impl} \)

definition \( \text{DFS-code-unfold} \) : \( \text{find-fas-impl} \ G \equiv \) do 

\( \text{ASSERT} \) (\( \text{graph} \ G) \) ;
ASSERT (finite ((g-E G)^+ g-V0 G));
 s ← fas_impl.tailrec_impl TYPE('a) G;
 RETURN (fas s)
}

context fas begin

sublocale simple_impl G fas-params fas-params_impl fas-erel
 apply unfold-locales
 apply (intro fun-relI, clarsimp simp: simple-state-rel-def, parametricity) []
 apply (auto simp: fas-erel-def fas-impl simple-state-rel-def)
 done

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

lemma find-fas-impl-refine: find-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 ≡ {
 (finfasimpl = fi, ..., = mi),([fas = f, ..., = m]) | fi mi f m.
 (fi,f) ∈ frel ∧ (mi,m) ∈ erel
}

consts
 i-fas-state-ext :: interface ⇒ interface ⇒ interface
lemmas [autoref-rel-intf] = REL-INTFI[of fas-state-erel i-fas-state-ext]

term fas-update
term fas-state-impl', fas-impl-update
lemma [autoref-rules]:
 fixes ns-rel vis-rel frel erel
 defines R ≡ ⟨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

158
schematic-goal find-fas-impl:
fixes $V :: (\text{\textquotesingle}v\times \text{\textquotesingle}v)^*$ set
assumes [autoref-ga-rules]: is-bounded-hashcode $V$ eq bhc
assumes [autoref-rules]: $(eq, (\equiv)) \in V \to V \to \text{bool-rel}$
assumes [autoref-ga-rules]: is-valid-def-hm-size $\text{TYPE} (\text{\textquotesingle}v)$ sz
assumes [autoref-rules]:
$(Gi, G) \in (\text{\textquotesingle}v\times \text{\textquotesingle}v)^*$ g-impl-rel-ext

notes [autoref-tyrel] =
$\text{TYRELI}[\text{where} R=\langle V\rangle \text{ahs-rel bhc}]$
$\text{TYRELI}[\text{where} R=\langle V \times_r V\rangle \text{ahs-rel} \ (\text{prod-bhc bhc bhc})]
\text{TYRELI}[\text{where} R=\langle V \times_r (\langle V \rangle) \text{list-set-rel} \rangle \text{ras-rel}]$
shows RETURN $(?c::?e'c) \leq \|$R (find-fas-impl $G$)

unfolding DFS-code-unfold
using [[autoref-trace-failed-id, goals-limit=1]]
apply (autoref-monadic (trace))
done

concrete-definition find-fas-code for eq bhc sz $Gi$ uses find-fas-impl
export-code find-fas-code checking SML

thm find-fas-code.refine

lemma find-fas-code-refine[refine]:
fixes $V :: (\text{\textquotesingle}v\times \text{\textquotesingle}v)^*$ set
assumes is-bounded-hashcode $V$ eq bhc
assumes $(eq, (\equiv)) \in V \to V \to \text{bool-rel}$
assumes is-valid-def-hm-size $\text{TYPE} (\text{\textquotesingle}v)$ sz
assumes $2: (Gi, G) \in (\text{\textquotesingle}v\times \text{\textquotesingle}v)^*$ g-impl-rel-ext
shows RETURN $(\text{find-fas-code eq bhc sz Gi}) \leq \|$\langle V \times_r V\rangle \text{ahs-rel} \ (\text{prod-bhc bhc bhc}) \rangle (\text{find-fas} G)$

proof –
note find-fas-code.refine[OF assms]
also note find-fas-impl-refine
finally show ?thesis .
qed

context begin interpretation autoref-syn .

Declare this algorithm to Autoref:

theorem find-fas-code-autoref[autoref-rules]:
fixes $V :: (\text{\textquotesingle}v\times \text{\textquotesingle}v)^*$ set and bhc
defines $RR \equiv \langle \langle V \times_r V\rangle \text{ahs-rel} \ (\text{prod-bhc bhc bhc})\rangle \text{nres-rel}$
assumes $BHC$: SIDE-GEN-ALGO (is-bounded-hashcode $V$ eq bhc)
assumes $EQ$: GEN-OP $eq (\equiv) (V \to V \to \text{bool-rel})$
assumes $VDS$: SIDE-GEN-ALGO (is-valid-def-hm-size $\text{TYPE} (\text{\textquotesingle}v)$ sz)
assumes $2: (Gi, G) \in (\text{\textquotesingle}v\times \text{\textquotesingle}v)^*$ g-impl-rel-ext
shows $(\text{RETURN} \ (\text{find-fas-code eq bhc sz Gi}),
\ (OP \text{find-fas}$
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 ≡ do {
        ASSERT (graph G);
        ASSERT (finite ((g-E G)* " g-V0 G));
        let nodes = (g-E G)* " g-V0 G;
        fas ← find-fas (\ g-V = g-V G, g-E = g-E G − FI, g-V0 = nodes \);
        RETURN (FI ∪ 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 ∪ F2) ↔ is-fas (∧ g-V = g-V G, g-E = g-E G − F2, g-V0 = (g-E G)* " g-V0 G \) F1

lemma graphI-init:
    assumes graph G
    shows graph (\ g-V = g-V G, g-E = g-E G − FI, g-V0 = (g-E G)* " g-V0 G \)

proof –
    interpret graph G by fact
    show thesis
        apply unfold-locales
        using reachable-V apply simp
        using E-ss apply force
    done

qed

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 (\lambda fas. is-fas G fas \land FI \subseteq fas)

unfolding find-fas-init-def
apply (refine-evg 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-refcl rtrancl-trancl-refcl
    trancl-rtrancl-absorb)
apply (simp add: is-fas-join[where ?F2.0=FI]
    Un-commute)
done

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

proof –
note [autoref-rules] = GEN-OP-D[OF INS]
note [autoref-rules] = GEN-OP-D[OF EM]
note [autoref-ga-rules] = SIDE-GEN-ALGO-D[OF IT]
have 1: CAST = (\lambda s \cup \{\}) by auto
show \thesis
  unfolding 1
  by autoref
qed

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

proof –
  have A: \\{ s. (\lambda x. x\in s,CAST s) \in br Collect (\lambda-. True)
  by (auto simp: br-def)
show \thesis
  unfolding fun-set-rel-def
  apply rule
  apply rule
  defer
  apply (rule A)
  using INS[simplified]
  apply parametricity
done
qed

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

Let $(E \ast V') = \text{Let } (\text{CAST } (E \ast V'))$

$(\lambda S. \text{RETURN } (F I \cup S)) = (\lambda S. \text{RETURN } (F I \cup \text{CAST } S))$

by simp-all

**schematic-goal** find-fas-init-impl:

**fixes** $V :: (\forall \mathbf{v} \times \forall \mathbf{v}) \text{ set and bhc}$

**assumes** [autoref-ga-rules]: is-bounded-hashcode $V$ eq $bhc$

**assumes** [autoref-rules]: $(eq, (=)) \in V \rightarrow V \rightarrow \text{bool-rel}$

**assumes** [autoref-ga-rules]: is-valid-def-hm-size $\text{TYPE } (\forall \mathbf{v})$ sz

**assumes** [autoref-rules]:

$(G_i, G) \in (\mathcal{R}_m, V) \text{ g-impl-rel-ext}$

$(F I, FI) \in (V \times rV) \text{ fun-set-rel}$

**shows** RETURN $(\lambda : : \forall \mathbf{v}') \leq \forall R \text{ (find-fas-init } G \text{ FI)}$

**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 $G_i$ $F I_i$

**uses** find-fas-init-impl

**export-code** find-fas-init-code checking SML

**context** begin interpretation autoref-syn .

The following theorem declares our implementation to Autoref:

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

**fixes** $V :: (\forall \mathbf{v} \times \forall \mathbf{v}) \text{ set and bhc}$

**defines** $RR \equiv (V \times rV) \text{ fun-set-rel}$

**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 $\text{TYPE } (\forall \mathbf{v})$ sz)

**shows** $(\lambda G_i F I_r. \text{RETURN } (\text{find-fas-init-code } eq \ bhc \ sz \ G_i \ F I_i, \text{find-fas-init})$

$\in (\mathcal{R}_m, V) \text{ g-impl-rel-ext } RR \rightarrow (RR) \text{nres-rel}$

**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}.

\textbf{thm} \textit{find-fas-code-autoref} \textit{find-fas-init-code-autoref}

\textbf{thm} \textit{find-fas-correct} \textbf{thm} \textit{find-fas-init-correct}

\textbf{end}

\section{Nested DFS}

\textbf{theory} \textit{Nested-DFS} \textbf{imports} \textit{DFS-Find-Path} \textbf{begin}

Nested DFS is a standard method for Buchi-Automaton emptiness check.

\subsection{Auxiliary Lemmas}

\textbf{lemma} \textit{closed-restrict-aux}:  
assumes \textsc{CL}: \( E``F \subseteq F \cup S \)  
assumes \textsc{NR}: \( E``U \cap S = \{\} \)  
assumes \textsc{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

\textbf{proof} \textbf{clarify}  
\textbf{fix} \( u, v \)  
\textbf{assume} \( A: (u,v) \in E`` u \in U \)  
\textbf{hence} \( M: E``\{u\} \cap S = \{\} \ u \in F \) \textbf{using} \textsc{NR} \textsc{SS} \textbf{by} \textit{blast+}

\textbf{from} \( A(1) \) \textbf{show} \( v \in F \)  
\textbf{apply} \textbf{(}induct rule: converse-rtrancl-induct\textbf{)}  
\textbf{using} \textsc{CL} \textbf{apply} \textbf{(}auto dest: rtrancl-Image-advance-ss\textbf{)}  
\textbf{done}

\textbf{qed}

\subsection{Instantiation of the Framework}

\textbf{record} \( 'v \text{ blue-dfs-state} = 'v \text{ state} + \text{ lasso :: ('v list } \times \text{ 'v list}) \text{ option} \)  
\textbf{red} \( \text{ : 'v set} \)

\textbf{type-synonym} \( 'v \text{ blue-dfs-param} = ('v, ('v,unit) \text{ blue-dfs-state-ext}) \text{ parameterization} \)

\textbf{lemma} \textit{lasso-more-cong[cong]}: \( \text{state}.more \ s = \text{state}.more \ s' \implies \text{lasso} \ s = \text{lasso} \ s' \)  
\textbf{by} \textbf{(}cases \ s, \text{ cases} \ s'\textbf{)} \textbf{simp}
lemma red-more-cong: state.more s = state.more s' → red s = red s'
by (cases s, cases s') simp

lemma [simp]: s[] state.more := (lasso = foo, red = bar []) = s (lasso := foo, red := bar [])
by (cases s) simp

abbreviation dropWhileNot v ≡ dropWhile ((#) v)
abbreviation takeWhileNot v ≡ takeWhile ((#) v)

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

begin

definition blue s ≡ dom (finished s) − red s
definition cyan s ≡ set (stack s)
definition white s ≡ − dom (discovered s)

abbreviation red-dfs R ss x ≡ find-path1-restr-spec (G | g-V0 := {x}) ss R

definition mk-blue-witness :: 'v blue-dfs-state ⇒ 'v fpr-result ⇒ ('v,unit) blue-dfs-state-ext
where
  mk-blue-witness s redS ≡ case redS of
    Inl R' ⇒ (lasso = None, red = (R' //////// red s))
  | Inr (vs, v) ⇒ let rs = rev (stack s) in
    (lasso = Some (rs, vs@dropWhileNot v rs), red = red s)

definition run-red-dfs :: 'v ⇒ 'v blue-dfs-state ⇒ ('v,unit) blue-dfs-state-ext nres
where
  run-red-dfs u s ≡ case lasso s of None ⇒ do
    redS ← red-dfs (red s) (λx. x = u ∨ x ∈ cyan s) u;
    RETURN (mk-blue-witness s redS)
  | - ⇒ NOOP s

Schwoon-Esparza extension
definition se-back-edge u v s ≡ case lasso s of
  None ⇒
    — it’s a back edge, so u and v are both on stack
    — we differentiate whether u or v is the ‘culprit’
    — to generate a better counter example
  if accpt u then
    let rs = rev (tl (stack s));
    ur = rs;
    ul = u#dropWhileNot v rs
  else
    RETURN (μ v. run-red-dfs v)
in RETURN (lasso = Some (ur,ul), red = red s)
else if accpt v then
  let rs = rev (stack s);
  vr = takeWhileNot v rs;
  vl = dropWhileNot v rs
  in RETURN (lasso = Some (vr, vl), red = red s)
else NOOP s
|
| - ⇒ NOOP s

definition blue-dfs-params :: 'v blue-dfs-param
where blue-dfs-params = {} 
  on-init = RETURN (lasso = None, red = {} ),
  on-new-root = λv0 s. NOOP s, 
  on-discover = λu v s. NOOP s,
  on-finish = λu v s. if accpt u then run-red-dfs u s else NOOP s, 
  on-back-edge = σe-back-edge, 
  on-cross-edge = λu v s. NOOP s, 
  is-break = λs. lasso s ≠ None []

schematic-goal blue-dfs-params-simps[simp]:
  on-init blue-dfs-params = ?OI
  on-new-root blue-dfs-params = ?ONR
  on-discover blue-dfs-params = ?OD
  on-finish blue-dfs-params = ?OF
  on-back-edge blue-dfs-params = ?OBE
  on-cross-edge blue-dfs-params = ?OCE
  is-break blue-dfs-params = ?IB

  unfolding blue-dfs-params-def gen-parameterization.simps
  by (rule refl)+

sublocale param-DFS-defs G blue-dfs-params 
by unfold-locales

end

locale BlueDFS = BlueDFS-defs G 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 accpt s
proof
  assume DFS-invar G blue-dfs-params s
  interpret DFS-invar G blue-dfs-params s
  by fact
  show BlueDFS-invar G accpt s
  by unfold-locales
next
  assume BlueDFS-invar G accpt s
  then interpret BlueDFS-invar G accpt s.
  show DFS-invar G blue-dfs-params s by unfold-locales
qed

end

2.5.3 Correctness Proof
context BlueDFS begin

definition blue-basic-invar s ≡
  case lasso s of
  None ⇒ restr-invar E (red s) (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)
do

lemma (in BlueDFS-invar) red-dfs-pres-bbi:
  assumes BI: blue-basic-invar s
  assumes [simp]: lasso s = None and SNE: stack s ≠ []
  assumes pending s "{hd (stack s)} = {}"
  shows run-red-dfs (hd (stack s)) (finish (hd (stack s)) s) ≤n SPEC (λe.
  SPEC (λe.

166
DFS-invar \( G \) blue-dfs-params \((\text{finish} (\text{hd} (\text{stack} s)) s | \text{state}.\text{more} := e)\) 
\( \rightarrow \) blue-basic-invar \((\text{finish} (\text{hd} (\text{stack} s)) s | \text{state}.\text{more} := e)\)

proof

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 \(\text{auto simp: neq-Nil-conv}\)

show \(\text{thesis}\)

unfolding \(\text{run-red-dfs-def}\)

apply \(\text{simp}\)

apply \(\text{(refine-vcg)}\)

apply \(\text{simp}\)

proof –

fix \(fp1\)

define \(s'\) where \(s' = \text{finish} (\text{hd} (\text{stack} s)) s\)

assume \(\text{FP-spec}: \text{find-path1-restr-pred} (G | g-V0 ::= \{\text{hd} (\text{stack} s)\}) (\lambda x. x \in \text{set} (\text{stack} s))\)

(\(\text{red} s\)) \(fp1\)

assume \(\text{BlueDFS-invar G accept} (s'|\text{state}.\text{more} := \text{mk-blue-witness} s' fp1)\)

then interpret \(i: \text{BlueDFS-invar G accept} (s'|\text{state}.\text{more} := \text{mk-blue-witness} s' fp1)\)

by \(\text{simp}\)

have \([\text{simp}]: \text{red} s' = \text{red} s\)

discovered \(s' = \text{discovered s}\)

dom \((\text{finished} s') = \text{insert} (\text{hd} (\text{stack} s)) \) \(\text{dom} \) \((\text{finished} s)\)

unfolding \(s'-\text{def finish-def}\) by \(\text{auto}\)

\{

fix \(R'\)

assume \([\text{simp}]: fp1 = \text{Inl} R'\)

from \(\text{FP-spec [unfolded find-path1-restr-pred-def, simplified]}\)

have

\(R'\text{FMT}: R' = \text{red} s \cup E^+ \{\text{hd} (\text{stack} s)\}\)

and \(\text{RI: restr-invar E R'} (\lambda x. x \in \text{set} (\text{stack} s))\)

by \(\text{auto}\)

from \(\text{BI}\) have \(\text{red} s \subseteq \text{dom} \) \((\text{finished} s)\)

unfolding \(\text{blue-basic-invar-def}\) by \(\text{auto}\)

also have \(E^+ \{\text{hd} (\text{stack} s)\} \subseteq \text{dom} \) \((\text{finished} s)\)

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 \(\text{(auto simp: trancl-unfold-left)}\)

167
from RI have NR: \( \{ \text{hd} \ (\text{stack} \ s) \} \cap \text{set} \ (\text{stack} \ s) = \{ \} \)
  unfolding restr-invar-def by (auto simp: R'FMT)

with \( \langle \text{hd} \ (\text{stack} \ s), u \rangle \in E \) have u \\notin \text{set} \ (\text{stack} \ s) by auto

have UID: \( u \in \text{dom} \ (\text{finished} \ s) \) by (auto simp: stack-set-def)

from NR \( \langle \text{hd} \ (\text{stack} \ s), u \rangle \in E \) have NR': \( E^* \{ u \} \cap \text{set} \ (\text{stack} \ s) = \{ \} \)
  by (auto simp: trancl-unfold-left)

have CL: \( E \vdash \text{dom} \ (\text{finished} \ s) \subseteq \text{dom} \ (\text{finished} \ s) \cup \text{set} \ (\text{stack} \ s) \)
  using finished-closed discovered-eq-finished-un-stack by simp

from closed-restrict-aux[OF CL NR'] UID have E'' \( \{ u \} \subseteq \text{dom} \ (\text{finished} \ s) \)
  unfolding finished-closed discovered-eq-finished-un-stack by simp

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 \( s' \langle \text{state} \ . \ more := \text{mk-blue-witness} \ s' \ fp1 \rangle \)
  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: restr-invar-def neq-Nil-conv)

  apply (auto dest!: aux1)
done
qed

lemma blue-basic-invar: is-invar blue-basic-invar
proof (induct rule: establish-invar1)
case (finish s) then interpret BlueDFS-invar where s=s by simp

have [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 (stack s \neq [])
  unfolding finish-def cyan-def by (auto simp: neq-Nil-conv)
from finish show ?case
  apply (simp)
  apply (intro conjI impI)
  apply (rule leof-trans[OF red-dfs-pres-bbi], assumption+, simp)
  apply (auto simp: restr-invar-def blue-basic-invar-def neq-Nil-conv) []
    done
qed (auto simp: blue-basic-invar-def cond-def se-back-edge-def
    simp: restr-invar-def empty-state-def pred-defs
    simp: DFS-invar.discovered-eq-finished-un-stack
    simp del: BlueDFS-invar-eq
    split: option.splits)

lemmas (in BlueDFS-invar) s-blue-basic-invar = blue-basic-invar[THEN make-invar-thm]

lemmas (in BlueDFS-invar) red-DFS-precond = red-DFS-precond-aux[OF s-blue-basic-invar]

sublocale DFS G blue-dfs-params
  apply unfold-locales
  apply (clarsimp-all simp: se-back-edge-def run-red-dfs-def refine-pw-simps pre-on-defs
    split: option.splits)
  unfolding nofail-SPEC-iff
  apply (refine-vcg)
  apply (erule BlueDFS-invar.red-DFS-precond, auto) []
    apply (simp add: cyan-def finish-def)
  apply (erule BlueDFS-invar.red-DFS-precond, auto) []
    apply (rule TrueI)
    done

end

context BlueDFS-invar
begin

context assumes [simp]: lasso s = None
begin
  lemma red-closed:
    E " red s ⊆ red s
  using s-blue-basic-invar
  unfolding blue-basic-invar-def restr-invar-def
  by simp

end
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-invar1)
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: neq-Nil-conv)
have 1: g-E (G [] g-V0 := {hd (stack s)} []) = E by simp
{
  fix R':v set
  let ?R' = R' \rel-restr stack-set-def
  let ?s = s'# lasso := None, red := ?R'
  assume \forall v. (hd (stack s), v) ∈ ?rE⁺ =⇒ ¬?onstack v
  and accept: accept u

and \( \text{NL[simp]}: \text{lasso} \ s = \text{None} \)

hence \( \text{no-hd-cycle}: (\text{hd} (\text{stack} \ s), \text{hd} (\text{stack} \ s)) \notin \ ?rE^+ \)

by auto

from \( \text{finish} \) have \( \text{stack} \ s \neq \ [] \) by simp

from \( \text{hd-in-set}[\text{OF this}] \) have \( \text{hd} (\text{stack} \ s) \notin \ \text{red} \ s \)

using \( \text{red-stack-disjoint} \)

by auto

hence \( (\text{hd} (\text{stack} \ s), \text{hd} (\text{stack} \ s)) \notin \ E^+ \)

using \( \text{no-hd-cycle rel-restrict-tranclI red-closed}[\text{OF NL}] \)

by \( \text{metis} \)

with \( \text{accept} \ \text{finish} \) have

\[ \forall \ x. \ \text{accept} \ x \land x \in \text{dom} (\text{finished} \ ?s) \longrightarrow (x,x) \notin E^+ \]

by auto

} with \( \text{finish} \) have

\( \text{red-dfs} (\text{red} \ s) \ ?\text{onstack} (\text{hd} (\text{stack} \ s)) \)

\[ \leq \ \text{SPEC} (\lambda \ x. \ \forall \ R. \ x = \text{Inl} \ R \longrightarrow \ DFS\text{-invar} \ G \ \text{blue-dfs-params (lasso-update Map.empty} s'(\text{red} := R \ \text{\forall/}) \}

\[ \longrightarrow\]

\[ (\forall \ x. \ \text{accept} \ x \land x \in \text{dom} (\text{finished} \ s') \longrightarrow (x,x) \notin E^+) \]

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 \( \text{aux} = \text{leaf-trans}[\text{OF this}[\text{simplied},\text{THEN} \ \text{leaf-lift}] \]

note \( \text{[refine-vcg del]} = \text{find-path1-restr-spec-rule} \)

from \( \text{finish} \) show \( ?\text{case} \)

apply simp

apply (intro conjI impI)

unfolding \[\text{run-red-dfs-def mk-blue-witness-def cyan-def} \]

apply clarsimp

apply (refined-vcg aux)

apply (auto split: sum.splits)

done

next

\[ \text{case back-edge} \] thus \( ?\text{case} \)

by (simp add: se-back-edge-def split: option.split)

qed simp-all

lemma \( \text{no-accept-cycle-in-finish}: \)

\[ [\text{lasso} \ s = \text{None}; \ \text{accept} \ v; v \in \text{dom} (\text{finished} \ s)] \Longrightarrow (v,v) \notin E^+ \]

using \( \text{i-no-accept-ycle-in-finish[THEN make-invar-thm]} \)
by blast

end

context BlueDFS

begin

definition lasso-inv where
  lasso-inv s ≡ ∀ pr pl. lasso s = Some (pr,pl) →
  pl ≠ [] ∧ (∃ vθ ∈ V0. path E vθ pr (hd pl)) ∧
  accept (hd pl) ∧
  path E (hd pl) pl (hd pl)

lemma (in BlueDFS-invar) se-back-edge-lasso-inv:
  assumes b-inv: lasso-inv s
  and ne: stack s ≠ []
  and R: lasso s = None
  and p: (hd (stack s), v) ∈ pending s
  and v: v ∈ dom (discovered s) v ∉ dom (finished s)
  and s': s' = back-edge (hd (stack s)) v (s[pending := pending s - {(u,v)}])
  shows se-back-edge (hd (stack s)) v s'
    ≤ SPEC (λe. DFS-invar G blue-dfs-params (s][state.more := e]) →
    lasso-inv (s'][state.more := e])

proof –

from v stack-set-def have v-in: v ∈ set (stack s) by simp
from p have uv-edg: (hd (stack s), v) ∈ E by (auto dest: pendingD)

{ assume accept: accept (hd (stack s))
  let ?ur = rev (tl (stack s))
  let ?ul = hd (stack s)#dropWhileNot v (rev (tl (stack s)))
  let ?s = s'][lasso := Some (?ur, ?ul), red := red s]

  assume DFS-invar G blue-dfs-params ?s

  have [simp]: stack ?s = stack s
    by (simp add: s')

  have hd-ul[simp]: hd ?ul = hd (stack s) by simp

  have ?ul ≠ [] by simp

  moreover have P: ∃ vθ ∈ V0. path E vθ ?ur (hd ?ul)
    using stack-is-path[OF ne]
    by auto

  moreover
  from accept have accept (hd ?ul) by simp

172
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 accept: accept 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 v (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 accept have accept (hd ?vl) by simp

  moreover from P ne v-in have path E (hd ?vl) ?vl (hd ?vl)
  by (force intro: dropWhileNot-path)

  ultimately have lasso-inv ?s by (simp add: lasso-inv-def)
}
moreover

\begin{align*}
\text{assume } & \neg \text{accept (hd (stack s))} \rightarrow \neg \text{accept v} \\
\text{let } & ?s = s'[\text{state.more} := \text{state.more } s'] \\
\text{assume } & \text{DFS-invar G blue-dfs-params ?s} \\
\text{from } & \text{assms have lasso-inv ?s} \\
\text{by } & (\text{auto simp add: lasso-inv-def})
\end{align*}

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

\begin{align*}
\text{using } & R \ s' \\
\text{unfolding } & \text{se-back-edge-def} \\
\text{by } & (\text{auto split: option.splits})
\end{align*}

qed

\textbf{lemma lasso-inv:}

\textit{is-invar lasso-inv}

\textbf{proof (induct rule: establish-invarI)}

\begin{align*}
\text{case } & (\text{finish s s' u}) \text{ then interpret BlueDFS-invar where } s=s \text{ by simp} \\
& \text{let } \text{o}n\text{stack } = \lambda x. x \in \text{set (stack s)} \\
& \text{let } ?rE = \text{rel-restrict E (red s)} \\
& \text{let } ?\text{revs } = \text{rev (tl (stack s))} \\
& \text{note } ne = \langle \text{stack s } \neq [\] \rangle \\
& \text{note } [\text{simp } ] = \langle u=\text{hd (stack s)} \rangle \\
& \text{from } \text{finish have } [\text{simp } ]: \\
& \quad \forall x. x = \text{hd (stack s)} \lor x \in \text{set (stack s')} \iff x\in\text{set (stack s)} \\
& \quad \text{red s'} = \text{red s} \\
& \quad \text{lasso s'} = \text{lasso s} \\
& \text{by } (\text{auto simp: neg-Nil-conv})
\end{align*}

\begin{align*}
\text{fix } & v \ vs \\
& \text{let } ?\text{cyc } = \text{vs }@\text{dropWhileNot v } ?\text{revs} \\
& \text{let } ?s = s'[\text{lasso } := \text{Some (?revs, ?cyc)}, \text{red } := \text{red s}] \\
\text{assume } & \text{DFS-invar G blue-dfs-params ?s} \\
& \text{and } \text{vs}: \text{vs } \neq [\] \text{ path } ?\text{rE (hd (stack s)) vs v} \\
& \text{and } v: \text{?onstack v} \\
& \text{and } \text{accept: accept (hd (stack s))}
\end{align*}
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.
    rs = Inr (vs,v)) →
  DFS-invar G blue-dfs-params (s[lasso := Some (?revs, vs @
    dropWhileNot v ?revs), red:= red s]) →
  lasso-inv (s[lasso := Some (?revs, vs @ dropWhileNot v ?revs),

175
\texttt{red:=\texttt{red s}})

\begin{verbatim}
apply clarsimp
apply (rule find-path1-restr-spec-rule, intro conjI)
apply (rule red-DFS-precond, simp-all add: ne) []
apply (simp, rule red-DFS-precond, simp-all add: ne) []

using red-stack-disjoint ne

apply clarsimp
apply rprems
apply (simp-all add: find-path1-restr-pred-def restr-invar-def)
apply (fastforce intro: path-restrict-tl rel-restrictI)
done

note aux1 = this[rule-format, THEN leof-lift]

show \texttt{?case}
apply simp
unfolding run-red-dfs-def mk-blue-witness-def cyan-def

apply (simp
  add: run-red-dfs-def mk-blue-witness-def cyan-def
  split: option.splits)
apply (intro conjI impI)
apply (refine-vcg leof-trans[OF aux1])
using finish
apply (auto simp add: lasso-inv-def split: sum.split)
done
next
  case (back-edge s s' u v)
  then interpret BlueDFS-invar where s=s by simp

  from back-edge se-back-edge-lasso-inv[THEN leaf-lift] show \texttt{?case}
  by auto
qed (simp-all add: lasso-inv-def empty-state-def)

end

context BlueDFS-invar
begin
lemmas s-lasso-inv = lasso-inv[THEN make-invar-thm]

lemma
  assumes lasso s = Some (pr,pl)
  shows loop-nonempty: pl \neq []
  and accpt-loop: accpt (hd pl)
  and loop-is-path: path E (hd pl) pl (hd pl)
  and loop-reachable: \exists v0 \in V0. path E v0 pr (hd pl)
  using assms s-lasso-inv
  by (simp-all add: lasso-inv-def)

lemma blue-dfs-correct:

end
\end{verbatim}
assumes $NC$: $\neg \, \text{cond } s$

shows case lasso $s$ of

$\text{None} \Rightarrow \neg (\exists v_0 \in V_0. \exists v. (v_0,v) \in E^* \land \text{accpt } v \land (v,v) \in E^+)$$

$| \text{Some } (pr,pl) \Rightarrow (\exists v_0 \in V_0. \exists v, \text{path } E v_0 \, pr \, v \land \text{accpt } v \land \text{pl} \neq \[] \land \text{path } E \, v \, \text{pl } v)$

proof (cases lasso $s$)

case None

moreover

{ fix $v$ $v_0$
 assume $v_0 \in V_0$ $(v_0,v) \in E^*$ accpt $v$ $(v,v) \in E^+$
 moreover
 hence $v \in \text{reachable by } (\text{auto})$
 with $\text{nc-finished-eq-reachable } NC$ None have $v \in \text{dom } (\text{finished } s)$
 by simp
 moreover note $\text{no-accept-cycle-in-finish } \text{None}$
 ultimately have False by blast
}

ultimately show $\?thesis$ by auto

next

case (Some prpl) with $s$-lasso-inv show $\?thesis$
 by (cases prpl)
 (auto intro: path-is-rtrancl path-is-trancl simp: lasso-inv-def)

qed

end

2.5.4 Interface

interpretation BlueDFS-defs for $G$ accpt.

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

$\text{None} \Rightarrow \neg (\exists v_0 \in g-V_0 \, G. \exists v. (v_0,v) \in (g-E \, G)^* \land \text{accpt } v \land (v,v) \in (g-E \, G)^+)$

$| \text{Some } (pr,pl) \Rightarrow (\exists v_0 \in g-V_0 \, G. \exists v, \text{path } (g-E \, G) v_0 \, pr \, v \land \text{accpt } v \land \text{pl} \neq \[] \land \text{path } (g-E \, G) \, v \, \text{pl } v)$

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

ASSERT (fb-graph $G$);

$s \leftarrow \text{it-dfs TYPE('a) } G$ accpt;

RETURN (lasso $s$)\n
$\}$

theorem nested-dfs-correct:

assumes $\text{fb-graph } G$

shows nested-dfs $G$ accpt $\leq \text{SPEC } (\text{nested-dfs-spec } G$ accpt)

proof -

interpret $\text{fb-graph } G$ by fact

interpret BlueDFS $G$ accpt by unfold-locales

177
show thesis
unfolding nested-dfs-def
apply (refine-rcg refine-vcg)
apply fact
apply (rule weaken-SPEC[OF it-dfs-correct])
apply clarsimp
proof –
  fix s
  assume BlueDFS-invar G 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

2.5.5 Implementation

record 'v bdfs-state-impl = 'v simple-state +
  lasso-impl :: ('v list × 'v list) option
  red-impl :: 'v set

definition bdfs-erel ≡
  {((lasso-impl=li,red-impl=ri),(lasso=l, red=r)) |
   li ri l r. li=l ∧ ri=r}

abbreviation bdfs-rel ≡ ⟨bdfs-erel⟩ simple-state-rel

definition mk-blue-witness-impl :: 'v bdfs-state-impl ⇒ 'v fpr-result ⇒ ('v,unit) bdfs-state-impl-ext
  where
  mk-blue-witness-impl s redS ≡
  case redS of
    Inl R' ⇒
      { lasso-impl = None, red-impl = (R'初创/初创/初创/初创red-impl s) |
        Inr (vs, v) ⇒ let
          rs = rev (map fst (CAST (ss-stack s)))
        in |
          lasso-impl = Some (rs, vs@dropWhileNot v rs),
          red-impl = red-impl s}

lemma mk-blue-witness-impl[refine]:
[ (si, s) ∈ bdfs-rel; (ri, r) ∈ (Id, Id) list-rel ×, Id) sum-rel ]
⇒ (mk-blue-witness-impl si ri, mk-blue-witness s r) ∈ bdfs-erel
unfolding mk-blue-witness-impl-def mk-blue-witness-def
apply parametricity
apply (cases si, cases s)
apply (auto simp: bdfs-erel-def simple-state-rel-def) []
apply (rule introR[where R=(Id)list-rel])
apply (cases si, cases s)
apply (auto simp: bdfs-erel-def simple-state-rel-def comp-def) []
apply (cases si, cases s)
apply (auto simp: bdfs-erel-def simple-state-rel-def) []
done

definition cyan-impl s ≡ on-stack s
lemma cyan-impl [refine]: [(si,s)∈bdfs-rel] ⟹ (cyan-impl si, cyan s)∈Id
  unfolding cyan-impl-def cyan-def
  by (auto simp: bdfs-erel-def simple-state-rel-def)

definition run-red-dfs-impl :: ('v,'more) graph-rec-scheme ⇒ 'v ⇒ 'v bdfs-state-impl ⇒ ('v,unit) bdfs-state-impl-ext
where
  run-red-dfs-impl G u s ≡ case lasso-impl s of None ⇒ do {
    redS ← red-dfs TYPE('more) G (red-impl s) (λx. x = u ∨ x ∈ cyan-impl s) u;
    RETURN (mk-blue-witness-impl s redS)
  } |
                  - ⇒ RETURN (simple-state.more s)
lemma run-red-dfs-impl [refine]: [(Gi,G)∈Id; (ui,u)∈Id; (si,s)∈bdfs-rel] ⟹ run-red-dfs-impl Gi ui si ≤⇓ bdfs-erel (run-red-dfs TYPE('a) G u s)
  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-def cyan-def) []
  apply (auto simp: bdfs-erel-def simple-state-rel-def) [2]
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)
  | - ⇒ RETURN (simple-state.more s)
lemma se-back-edge-impl [refine]: [(accepti,accept)∈Id; (ui,u)∈Id; (vi,v)∈Id; (si,s)∈bdfs-rel]
\[ \Rightarrow \text{se-back-edge-impl accpt } u \text{ } v \text{ } s \leq \triangledown \text{bdfs-erel (se-back-edge accpt } u \text{ } v \text{ } s) \]

**unfolding** se-back-edge-impl-def se-back-edge-def

**apply** refine-rcg

**apply** refine-dref-type

**apply** simp-all

**apply** (simp-all add: bdfs-erel-def simple-state-rel-def)

**apply** (cases si, cases s, (auto [])]

**apply** (cases si, cases s, (auto simp: map-tl comp-def) [])

**apply** (cases si, cases s, (auto simp: comp-def) [])

**done**

**lemma** NOOP-impl: (si, s) \(\in\) bdfs-rel

\[ \Rightarrow \text{RETURN } (\text{simple-state}.\text{more } si) \leq \triangledown \text{bdfs-erel } (\text{NOOP } s) \]

**apply** (simp add: pw-le-iff refine-pw-simps)

**apply** (auto simp: simple-state-rel-def)

**done**

**definition** bdfs-params-impl

:: \((v, 'more) \text{ graph-rec-scheme } \Rightarrow (v \Rightarrow \text{bool}) \Rightarrow (v, 'v \text{ bdfs-state-impl}, ('v, unit) bdfs-state-impl-ext)\)

**gen-parameterization**

**where** bdfs-params-impl G accpt \(\equiv\) ()

on-init = RETURN \(\langle\text{lasso-impl } = \text{None, red-impl } \equiv \{\}\rangle\),

on-new-root = \(\lambda u \text{ } v \text{ } s. \text{RETURN } (\text{simple-state}.\text{more } s)\),

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

on-finish = \(\lambda u \text{ } s.\)

if accpt u then run-red-dfs-impl G u s else RETURN \(\text{simple-state}.\text{more } s\),

on-back-edge = se-back-edge-impl accpt,

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

is-break = \(\lambda s.\) 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 accpt blue-dfs-params

TYPE('a) G accpt

for G accpt .

**context BlueDFS begin**

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

**apply** unfold-locales

**apply** (simp-all

add: bdfs-params-impl-def run-red-dfs-impl se-back-edge-impl NOOP-impl)

**apply** parametricity

**apply** (clarsimp-all simp: pw-le-iff refine-pw-simps bdfs-erel-def simple-state-rel-def)
apply (rename-tac si s x y, case-tac si, case-tac s)
apply (auto simp add: bdfs-erel-def simple-state-rel-def) []
done

lemmas impl = impl.simple-tailrec-refine
end

definition nested-dfs-impl G accept ≡ do {
  ASSERT (fb-graph G);
  s ← impl.tailrec-impl TYPE('a) G accept;
  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-rcg
apply simp-all
apply (rule intro-prgR[where R=bdfs-rel])
defer
apply (rename-tac si s)
apply (case-tac si, case-tac s)
apply (auto simp: bdfs-erel-def simple-state-rel-def) []
proof –
  assume fb-graph G
  then interpret fb-graph G.
  interpret BlueDFS G by unfold-locales

  from impl show impl.tailrec-impl TYPE('b) G accept ≤⇓ bdfs-rel (it-dfs TYPE('b) G accept) .
qed

2.5.6 Synthesis of Executable Code

record ('v,'si,'nsi)bdfs-state-impl' = ('si,'nsi)simple-state-impl +
  lasso-impl': ('v list × 'v list) option
red-impl': 'nsi

definition [to-relAPP]: bdfs-state-erel' Vi ≡ {
  (〈lasso-impl' = li, red-impl'='ri〉〈lasso-impl = l, red-impl = r〉) | li ri l r.
  (li,l)∈(⟨Vi⟩list-rel ×r ⟨Vi⟩list-rel)option-rel∧(ri,r)∈(Vi)dfil-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 ×r ⟨Vi⟩list-rel⟩option-rel → ⟨⟨Vi⟩dflt-ahs-rel → unit-rel → ⟨Vi⟩bdfs-state-erel′
    (lasso-impl', lasso-impl) ∈ R → ⟨⟨Vi⟩list-rel ×r ⟨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]:
    (Gi, 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 ×r ⟨V⟩list-rel⟩option-rel⟩nres-rel
unfolding nested-dfs-impl-def[abs-def] Vid
  sc-back-edge-impl-def run-red-dfs-impl-def mk-blue-witness-impl-def
  cyan-impl-def
  DFS-code-unfold
using [[autoref-trace-failed-id]]
apply (autoref-monadic (trace))
done

concrete-definition nested-dfs-code uses nested-dfs-code

export-code nested-dfs-code checking SML

2.5.7 Conclusion

We have implemented an efficiently executable nested DFS algorithm. The following theorem declares this implementation to the Autoref tool, such that it uses it to synthesize efficient code for nested-dfs. Note that you will need the lemma nested-dfs-correct to link nested-dfs to an abstract specification, which is usually done in a previous refinement step.

theorem nested-dfs-autoref[autoref-rules]:
  assumes PREFER-id V
  shows (λ G accept. nres-of (nested-dfs-code G accept), nested-dfs) ∈
    ⟨Rm, V⟩g-impl-rel-ext → (V → bool-rel) →

182
\[(\langle V \rangle \text{list-rel} \times_r (V \{\text{list-rel}\}) \text{option-rel}) \text{nres-rel}\]

**Proof** –

from **assms have** \( \text{Vid} : V = \text{Id} \) by **simp**

**note** nested-dfs-code.refine[\( \text{OF Vid, param-fo, THEN nres-relD} \)]

**also note** nested-dfs-impl

**finally show** ?thesis by (fastforce intro: nres-relI)

qed

end

### 2.6 Invariants for Tarjan’s Algorithm

**theory** Tarjan-LowLink

**imports**

../DFS-Framework

../Invars/DFS-Invars-SCC

**begin**

**context** param-DFS-defs begin

**definition**

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

**definition**

\[
\text{lowlink-set } s \; v \equiv \{w \in \text{dom } (\text{discovered } s). \quad v = w \quad \lor (v,w) \in E^+ \land (w,v) \in E^+ \land (\exists p. \text{lowlink-path } s \; v \; p \; w)\}
\]

**context** begin interpretation timing-syntax .

**abbreviation** LowLink where

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

end

end

**context** DFS-invar begin

**lemma** lowlink-setI:

**assumes** lowlink-path \( s \; v \; p \; w \)

and \( w \in \text{dom } (\text{discovered } s) \)

and \( (v,w) \in E^+ \land (w,v) \in E^+ \land (\exists p. \text{lowlink-path } s \; v \; p \; w)\)

**shows** \( w \in \text{lowlink-set } s \; v \)

**proof** (cases \( v = w \))
case True thus \( \text{thesis by (simp add: lowlink-set-def assms)} \)
next
case False with assms have \((v,w) \in E^+ (w,v) \in E^+ \) by (metis rtrancl-eq-or-trancl+)
with assms show \( \text{thesis by (auto simp add: lowlink-set-def)} \)
qed

lemma lowlink-set-discovered:
lowlink-set \( s \ v \subseteq \text{dom (discovered } s) \)
unfolding lowlink-set-def
by blast

lemma lowlink-set-finite[simp, intro!]:
finite (lowlink-set \( s \ v \))
using lowlink-set-discovered discovered-finite
by (metis finite-subset)

lemma lowlink-set-not-empty:
assumes \( v \in \text{dom (discovered } s) \)
shows lowlink-set \( s \ v \neq \{\} \)
unfolding lowlink-set-def
using assms by auto

lemma lowlink-path-single:
assumes \((v,v) \in \text{cross-edges } s \cup \text{back-edges } s \)
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: \( (x\#xs)!1 \in \text{dom (finished } s) \)
and t: \( (\forall k < \text{length } xs. (((x\#xs)!k, (x\#xs)!Suc k) \in \text{tree-edges } s) \)
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 \( k' = Suc k \)
assume \( k < \text{length } xs - 1 \)
with \( k'-def \) have \( k' < \text{length } xs \) by simp
with \( t \) have \( ((x\#xs)!k', (x\#xs)!Suc k') \in \text{tree-edges } s \) by simp
hence \( (xs!k, xs!Suc k) \in \text{tree-edges } s \) by (simp add: k'-def nth-Cons')
\} note \( t' = this \)
moreover { 
  assume \( \ast \): \( \text{length} \ x s > 1 \)
  from \( f \) have \( x s!0 \in \text{dom} \ (\text{finished} \ s) \) by auto
  moreover from \( t' \{0\} \ast \) have \( (x s!0,x s!1) \in \text{tree-edges} \ s \) by simp
  ultimately have \( x s!1 \in \text{dom} \ (\text{finished} \ s) \) using \text{tree-edge-impl-parenthesis}
  by metis 
}

ultimately have \( \text{lowlink-path} \ s \ u x s \ w \) by (auto simp add: \text{lowlink-path-def})
thus \( \theta \text{thesis} \).

qed

lemma \text{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: \text{lowlink-path-def})
  thus \( \theta \text{thesis} \) using \text{p j k}
proof (induction arbitrary: \( v \ j \ k \) rule: list-nonempty-induct)
  case single
  thus \( \theta \text{case} \) by auto
next
  case (\text{cons} \( x \) \( x s \))
  define \( j' \) where \( j' = j - 1 \)
  with \text{cons} have \( j' - \text{le}: j' < \text{length} \ x s \) and \( k \leq j' \) and \( j: j = \text{Suc} \ j' \) by auto

  from \text{cons} \text{lowlink-path-Cons} obtain \( u \) where \( p \): \text{lowlink-path} \ s \ u x s \ w \) by blast

  show \( \theta \text{case} \)
proof (cases \( k = 0 \))
  case True
  from \text{cons} have \( \\bigwedge k. k < \text{length} \ (x#xs) - 1 \implies ((x#xs)!k,(x#xs)!\text{Suc} \ k) \in \text{tree-edges} \ s \)
  unfolding \text{lowlink-path-def}
  by auto
  moreover from \text{True} \text{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 \( \theta \text{thesis} \)
proof (cases \( j' = 0 \))
  case True with \( \ast \) \( k = 0 \) show \( \theta \text{thesis} \) by simp
next
  case \text{False} with \text{True} \ have \( j' > k \) by simp
  with \text{cons.IH[OF} \ p \ j' - \text{le]} have \( (x s!k, x s!j') \in (\text{tree-edges} \ s)^+ \).
  with \( j \) have \( ((x#xs)!k,(x#xs)!j) \in (\text{tree-edges} \ s)^+ \) by simp
  with \( \ast \) show \( \theta \text{thesis} \) by (metis \text{trancl-into-trancl2})
qed

185
next
  case False
define $k' \text{ where } k' = k - 1$
with False \langle $k \leq j' \text{ and } k = \text{Suc } k' \text{ by simp-all} \rangle
with cons.IH[OF p $j'\leq$] have $(x # xs'; x # xs') \in (\text{tree-edges } s)^+$ by simp
hence $((x # xs) ! \text{Suc } k', (x # xs) ! \text{Suc } j') \in (\text{tree-edges } s)^+$ by simp
with $k j$ show ?thesis by simp
qed

lemma lowlink-path-finished:
assumes $p: \text{lowlink-path } s \ v \ p \ w$
and $j: j < \text{length } p$ $j > 0$
shows $\text{plj} \in \text{dom } (\text{finished } s)$
proof –
from $j$ have length $p > 1$ by simp
with $p$ have $f: p ! 1 \in \text{dom } (\text{finished } s)$ by (simp add: lowlink-path-def)
thus ?thesis
proof (cases $j = 1$)
case False with $j$ have $j > 1$ by simp
with assms lowlink-path-in-tree[where $k = 1$] have $(p ! 1, p ! j) \in (\text{tree-edges } s)^+$
by simp
  with $f$ tree-path-impl-parenthesis show ?thesis by simp
qed simp
qed

lemma lowlink-path-tree-prepend:
assumes $p: \text{lowlink-path } s \ v \ p \ w$
and $\text{tree-edges}: (u, v) \in (\text{tree-edges } s)^+$
and fin: $u \in \text{dom } (\text{finished } s) \lor (\text{stack } s \neq [] \land u = \text{hd } (\text{stack } s))$
shows $\exists p. \text{lowlink-path } s \ u \ p \ w$
proof –
  note lowlink-path-def[simp]
from tree-edges trancl-is-path obtain $tp$ where
  $tp: \text{path } (\text{tree-edges } s) \ u \ tp \ v \ tp \neq []$
  by metis
from tree-path-impl-parenthesis assms $\text{hd-stack-tree-path-finished}$ have
  $v$-fin: $v \in \text{dom } (\text{finished } s)$ by blast
from $p$ have $p ! 0 = \text{hd } p$ by (simp add: hd-conv-nth)
with $p$ have $p ! 0 = v$ by (auto simp add: path-hd)
let ?$p = tp @ p$
{
  from $tp$ path-mono[OF tree-edges-ssE] have $\text{path } E \ u \ tp \ v$ by simp
}
also from \( p \) have \( \text{path } E \ v \ p \ w \) by simp
finally have \( \text{path } E \ u \ ?p \ w \).

moreover from \( p \) have \( ?p \neq [] \) by simp

moreover from \( p \) have \((\text{last } ?p, w) \in \text{cross-edges } s \cup \text{back-edges } s\) by simp

moreover { assume length \( ?p > 1 \)
have \( ?p ! 1 \in \text{dom } (\text{finished } s) \)
proof (cases length \( tp > 1 \))
case True hence \( tp1: ?p ! 1 = tp ! 1 \) by (simp add: nth-append)
from \( tp \) True have \( (tp ! 0, tp ! 1) \in (\text{tree-edges } s)^+ \)
by (auto simp add: path-nth-conv nth-append elim: allE[where \( x=0 \)])
also from \( True \) \( \text{have } tp ! 0 = \text{hd } tp \) by (simp add: hd-conv-nth)
also from \( \text{tp have } \text{hd } tp = u \) by (simp add: path-hd)
finally have \( \text{tp ! 1 } \in \text{dom } (\text{finished } s) \)
using tree-path-impl-parenthesis fin hd-stack-tree-path-finished by blast
thus \( \text{?thesis } \) by (subst \( tp1 \))
next case False with \( \text{tp have } \text{length } tp = 1 \) by (cases \( tp \)) auto
with \( p-0 \) have \( ?p ! 1 = v \) by (simp add: nth-append)
thus \( \text{?thesis } \) by (simp add: v-fin)
qed

also have \( \forall k < \text{length } ?p - 1. (\ ?p!k, \ ?p!Suc k) \in \text{tree-edges } s \)
proof (safe)
fix \( k \)
assume \( A: k < \text{length } ?p - 1 \)
show \( (\ ?p!k, \ ?p!Suc k) \in \text{tree-edges } s \)
proof (cases \( k < \text{length } tp \))
case True hence \( k: ?p ! k = tp ! k \) by (simp add: nth-append)
show \( \text{?thesis } \)
proof (cases Suc \( k < \text{length } tp \))
case True hence \( ?p ! Suc k = tp ! Suc k \) by (simp add: nth-append)
moreover from \( True \) \( \text{tp have } (tp ! k, tp ! Suc k) \in \text{tree-edges } s \)
by (auto simp add: path-nth-conv nth-append
elim: allE[where \( x=k \)])
ultimately show \( \text{?thesis } \) using \( k \) by simp
next case False with \( True \) \( \text{have } *: Suc k = \text{length } tp \) by simp
with \( tp \) 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 \( \text{?thesis } \) by (simp add: \( k \) )

187
qed
next
  case False hence \( \ast \): Suc \( k - \text{length} \ tp = \text{Suc} \ (k - \text{length} \ tp) \) by simp
  define \( k' \) where \( k' = k - \text{length} \ tp \)
  with False \( \ast \) have \( k' : \ ?p \ ! k = p \ ! k' \ ?p ! \text{Suc} \ k = p ! \text{Suc} \ k' \) by (simp-all add: nth-append)
  from \( k'-\text{def} \) False A have \( k' < \text{length} \ p - 1 \) by simp
  with \( p \) have \( (p!k', p!\text{Suc} \ k') \in \text{tree-edges} \ s \) by simp
  with \( k' \) show \( \?\text{thesis} \) by simp
qed
qed

also \( \text{(conjI)} \) note calculation

ultimately have \( \text{lowlink-path} \ s \ u \ ?p \ w \) by simp
thus \( \?\text{thesis} \) ..
qed


lemma \( \text{lowlink-path-complex} \):
  assumes \( (u,v) \in (\text{tree-edges} \ s)^+ \)
  and \( u \in \text{dom} \ (\text{finished} \ s) \lor (\text{stack} \ s \neq [] \land u = \text{hd} \ (\text{stack} \ s)) \)
  and \( (v,w) \in \text{cross-edges} \ s \cup \text{back-edges} \ s \)
  shows \( \exists p. \ \text{lowlink-path} \ s \ u \ p \ w \)
proof
  from assms lowlink-path-single have \( \text{lowlink-path} \ s \ v \ [v] \ w \) by simp
  with assms lowlink-path-tree-prepend show \( \?\text{thesis} \) by simp
qed

lemma \( \text{no-path-imp-no-lowlink-path} \):
  assumes \( \text{edges} \ s = \{v\} \)
  shows \( \neg \text{lowlink-path} \ s \ v \ p \ w \)
proof
  assume \( p: \ \text{lowlink-path} \ s \ v \ p \ w \)
  hence \( [\text{simp}]: p \neq [] \) by (simp add: lowlink-path-def)
  from \( p \) have \( \text{hd} \ p = v \) by (auto simp add: lowlink-path-def-path-hd)
  with \( \text{hd-conv-nth} \ (\text{OF} \ p \neq []) \) have \( v: p!0 = v \) by simp
  show False
proof \( \text{(cases length} \ p > 1 \) )
  case True with \( p \) have \( (p!0,p!1) \in \text{tree-edges} \ s \) by (simp add: lowlink-path-def)
  with \( v \) assms show False by auto
next
  case False with \( p \neq [] \) have \( \text{length} \ p = 1 \) by (cases \( p \) ) auto
  hence \( \text{last} \ p = v \) by (simp add: last-conv-nth \( v \) )
  with \( p \) have \( (v,w) \in \text{edges} \ s \) by (simp add: lowlink-path-def)
  with assms show False by auto

188
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: ∀x. [x ∈ X; x ∈ lowlink-set s' w] ⟹ δ s' x ≥ LowLink s w
  shows LowLink s w = 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 \text{lowlink-set-discovered \ sub \ sub-m \ w-disc} have

- $\text{sub': } \delta s' \ \text{lowlink-set } s \ w \subseteq \delta s' \ \text{lowlink-set } s' \ w$
- $\text{and } \text{w-disc': } w \ \in \ \text{dom (discovered } s')$
- $\text{and } \text{eq: } \forall ll. \ ll \in \text{lowlink-set } s \ w \implies \delta s' \ ll = \delta s \ ll$
- $\text{by (force simp: map-le-def)}$

from \text{lowlink-set-not-empty[OF w-disc]} A \text{ Min-antimono[OF sub'} s', \text{lowlink-set-finite}

have

$\text{LowLink } s' \ w < \text{LowLink } s \ w$ by \text{fastforce}

then obtain $ll$ where $ll \in \text{lowlink-set } s' \ w$ and $ll-le: \delta s' \ ll < \text{LowLink } s \ w$
- $\text{by (metis s'} \text{LowLink-lessE } w-disc'$

with \text{rev-sub have } ll \in \text{lowlink-set } s \ w \lor ll \in X \text{ 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 ?thesis by force}
- next

- assume $ll \in X$ with $ll$ show ?thesis by (metis X)

qed

with $ll-le$ show False by simp

qed

\text{lemma \text{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 \text{assms finished-discovered stack-discovered hd-in-set have disc: } v \in \text{dom (discovered } s)$ by \text{blast}

with \text{assms LowLink-le-disc } A \text{ have } \text{LowLink } s \ v < \delta s \ v \text{ by force}

with $\text{disc obtain } w$ where

- $w: \delta s \ w < \delta s \ v \ \in \ \text{lowlink-set } s \ v$

- $\text{by (metis LowLink-lessE)}$

with $\text{lowlink-set-discovered}$ have $\text{wdisc: } w \in \text{dom (discovered } s)$ by \text{auto}

from $w$ have $(v,w) \in E^* (w,v) \in E^*$ by (auto simp add: lowlink-set-def)

moreover have $\text{is-scc } E \ ?sce \ v \in ?sce \ \text{by simp-all}$

ultimately have $w \in ?sce \ \text{by (metis is-scc-closed)}$

with $\text{wdisc sce-root-disc-le[OF sce]}$ have $\delta s \ v \leq \delta s \ w$ by simp

with $w$ show $\text{False by auto}$

qed

next
assume LL: \text{LowLink } s \ v = \delta \ s \ v

from assms finished-discovered stack-discovered hd-in-set have
\text{v-disc: } v \in \text{dom (discovered s)} by blast

from assms finished-no-pending have
\text{v-no-p: pending s } \{v\} = \{\} by blast

let ?scc = scc-of E v
have is-scc: is-scc E ?scc by simp

{ fix r
assume r \neq v
and r \in ?scc r \in \text{dom (discovered s)}

have v \in ?scc by simp
with (r \in ?scc) is-scc have (v, r) \in (\text{Restr E } ?scc)^*
by (simp add: is-scc-connected')

hence (v, r) \in (\text{tree-edges s})^+ using (r \neq v)
proof (induction rule: rtrancl-induct)
case (step y z) hence (v, z) \in (\text{Restr E } ?scc)^*
by (metis rtrancl-into-rtrancl)

hence (v, z) \in E^* by (metis Restr-rtrancl-mono)

from step have (z, v) \in E^* by (simp add: is-scc-connected[\text{OF is-scc}])

{ assume \text{z-disc: } z \in \text{dom (discovered s)}
and \exists p. \text{lowlink-path s v p z}
with \langle z, v \rangle \in E^* \langle v, z \rangle \in E^* have ll: z \in \text{lowlink-set s v}
by (metis lowlink-setI)

have (v, z) \in \text{edges s} by auto
proof (rule ccontr)

have (z, v) \in E^* \text{ disc-unequal have } \delta \ s \ v
z < \delta \ s \ v by fastforce

with LL show False by simp
qed simp
} note \delta z = this

show \text{?case}

proof (cases y=v)
case True note [simp] = this
with step v-no-p v-disc no-pending-imp-succ-discovered have
z-disc: z \in \text{dom (discovered s)} by blast

from step edges-covered v-no-p v-disc have (v, z) \in \text{edges s} by auto

191
thus \(?thesis\)

proof (rule edgesE-CB)
  assume \((v,z) \in \text{tree-edges } s\) thus \(?thesis\) ..

next
  assume \(CB\): \((v,z) \in \text{cross-edges } s \cup \text{back-edges } s\)
  hence \(\text{lowlink-path } s v [v] z\)
    by (simp add: lowlink-path-single)
  with \(\delta z\) [OF \(\text{z-disc}\)] \(\text{no-pending-succ-impl-path-in-tree v-disc v-no-p step}\)

show \(?thesis\)
  by auto

qed

next

case \text{False} with \text{step.IH} have \(T\): \((v,y) \in (\text{tree-edges } s)^+\)
  with \text{tree-path-impl-parenthesis} assms hd-stack-tree-path-finished tree-path-disc have
  \(y\)-fin: \(y \in \text{dom (finished } s)\)
  and \(y\)-\(\delta\): \(\delta s v < \delta s y\) by blast+
  with \text{step} have \(\text{z-disc}: z \in \text{dom (discovered } s)\)
    using \text{finished-imp-succ-discovered}
    by auto

from \text{step} edges-covered finished-no-pending[\text{of } y] \(y\)-fin finished-discovered have
  \((y,z) \in \text{edges } s\)
  by fast

thus \(?thesis\)

proof (rule edgesE-CB)
  assume \((y,z) \in \text{tree-edges } s\) with \(\text{T}\) show \(?thesis\) ..

next
  assume \(CB\): \((y,z) \in \text{cross-edges } s \cup \text{back-edges } s\)
  with \text{lowlink-path-complex}[OF \(T\)] assms have
    \(\exists p. \text{lowlink-path } s v p z\) by blast
  with \(\delta z\) \(\text{z-disc}\) have \(\delta z\): \(\delta s v < \delta s z\) by simp

show \(?thesis\)

proof (cases \(v \in \text{dom (finished } s)\))
  case \text{True} with \text{tree-path-impl-parenthesis} \text{T} have \(y\)-\(f\): \(\varphi s y < \varphi s v\)
    by blast

from \text{CB} show \(?thesis\)

proof
  assume \(C\): \((y,z) \in \text{cross-edges } s\)
  with \text{cross-edges-finished-decr} \(y\)-\(\text{fin}\) \(y\)-\(f\) have \(\varphi s z < \varphi s v\)
    by force
  moreover note \(\delta z\)
  moreover from \(C\) \text{cross-edges-target-finished} have
    \(z \in \text{dom (finished } s)\) by simp
  ultimately show \(?thesis\)
    using parenthesis-impl-tree-path[OF \(\text{True}\)] by metis
next
  assume B: \( (y,z) \in \text{back-edges}\, s \)
  with back-edge-disc-lt-fin y-fin y-f have \( \delta\, s\, z < \varphi\, s\, v \)
  by force
moreover note \( \delta\, z\, \text{disc} \)
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 ?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 " \{v\} = \{
  have z \in \text{dom}\, (\text{finished}\, s)
  proof (rule ccontr)
    assume z \notin \text{dom}\, (\text{finished}\, s)
    with \( z\, \text{disc} \) have z \in set (stack s) by (simp add: stack-set-def)
    with \( (z \neq v): \text{st} \) have z \in set (tl (stack s)) by (cases stack s) auto
    with \( \text{st}\, \text{tl}\,\text{-lt}\,\text{stack}\,\text{-hd}\,\text{discover} \) \( \delta\, z\) show False by force
  qed
  with \( \delta\, z\) parenthesis-impl-tree-path-not-finished \( v\, \text{disc} \) False show ?thesis
  by simp
  qed
  qed
  qed simp
  hence \( v \in (\text{tree-edges}\, s)^* \) " \{v\} by auto
  }
  hence ?scc \cap \text{dom}\, (\text{discovered}\, s) \subseteq (\text{tree-edges}\, s)^* " \{v\}
  by fastforce
  thus scc-root s v ?scc by (auto intro!: scc-rootI \( v\, \text{disc} \))
  qed
end end

\section{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-eq-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:
\[ \text{tj-stack-pop } tjs \ u \leq \text{SPEC } (\lambda(tjs',scc). \ u \in \text{set } tjs \implies \text{set } tjs = \text{set } tjs' \cup scc \land u \in scc) \]
proof -
from \[ \text{tjs-union[of u tjs]} \] show \( ? \text{thesis} \)
unfolding \( \text{tj-stack-pop-def} \)
by \( \text{refine-vcg) auto} \)
qed

lemmas \( \text{tj-stack-pop-set-leof-rule} = \text{weaken-SPEC[OF tj-stack-pop-set, THEN leaf-lift]} \)

definition \( \text{tarjan-fin} :: 'v \Rightarrow 'v \text{ tarjan-state } \Rightarrow ('v,unit) \text{ tarjan-state-ext nres} \)
where
\[ \text{tarjan-fin } v \ s = \text{do } \{ \]
  let ll = (if stack s = [] then lowlink s
  else let u = hd (stack s) in
  (lowlink s)(u \mapsto \text{min } (\zeta s u) (\zeta s v)));
  let s' = s [] lowlink := ll [;]
  ASSERT (v \in \text{set } (\text{tj-stack } s));
  ASSERT (\text{distinct } (\text{tj-stack } s));
  if \( \zeta s v = \delta s v \) then do \{
    ASSERT (\text{sec-root'E s v } (\text{sec-of E v}));
    (tjs,scc) \leftarrow \text{tj-stack-pop } (\text{tj-stack } s) \ v;
    \text{RETURN } (\text{state.more } (s[]) (\text{tj-stack := tjs}, \text{sccs := insert scs (scs s))})\}\]
  else do \{
    ASSERT (\text{\neg sec-root'E s v } (\text{sec-of E v}));
    \text{RETURN } (\text{state.more s'}))
  \}\}

definition \( \text{tarjan-back} :: 'v \Rightarrow 'v \Rightarrow 'v \text{ tarjan-state } \Rightarrow ('v,unit) \text{ tarjan-state-ext nres} \)
where
\[ \text{tarjan-back } u \ v \ s = \text{ (} \]
  if \( \delta s v < \delta s u \land v \in \text{set } (\text{tj-stack } s) \) then
  let ul' = \text{min } (\zeta s u) (\delta s v)
  in \text{RETURN } (\text{state.more } (s[]) \text{ lowlink := (lowlink s)(w\rightarrow ul') [)})
  else \text{NOOP s)\}
end

definition \( \text{tarjan-params} :: 'v \text{ tarjan-param} \) where
\[ \text{tarjan-params = []} \]
  on-init = \text{RETURN } () \text{ sccs = [], lowlink = Map.empty, tj-stack = [] []},
  on-new-root = \text{tarjan-disc},
  on-discover = \lambda u. \text{tarjan-disc},
  on-finish = \text{tarjan-fin},
  on-back-edge = \text{tarjan-back},
\[on\text{-}cross\text{-}edge = \text{tarjan\text{-}back},\]
\[is\text{-}break = \lambda s. \text{False} \]

\textbf{schematic\text{-}goal tarjan\text{-}params\text{-}simps[simp]}:
\begin{align*}
on\text{-}init & \text{tarjan\text{-}params} = \text{?OI} \\
on\text{-}new\text{-}root & \text{tarjan\text{-}params} = \text{?ONR} \\
on\text{-}discover & \text{tarjan\text{-}params} = \text{?OD} \\
on\text{-}finish & \text{tarjan\text{-}params} = \text{?OF} \\
on\text{-}back\text{-}edge & \text{tarjan\text{-}params} = \text{?OBE} \\
on\text{-}cross\text{-}edge & \text{tarjan\text{-}params} = \text{?OCE} \\
is\text{-}break & \text{tarjan\text{-}params} = \text{?IB} \\
\end{align*}
\textbf{unfolding tarjan\text{-}params\text{-}def gen\text{-}parameterization.simps}
\textbf{by} (rule \text{refl})+

\textbf{sublocale} \text{param\text{-}DFS\text{-}defs G} \text{tarjan\text{-}params}.

\textbf{locale} \text{Tarjan} = \text{Tarjan\text{-}def G} + \text{param\text{-}DFS G} \text{tarjan\text{-}params}
\textbf{for} \text{G :: ('}v, 'more\text{) graph\text{-}rec\text{-}scheme}
\textbf{begin}

\textbf{lemma} \text{[simp]}:
\begin{align*}
\text{sccs (empty\text{-}state (}\text{sccs} = s, \text{lowlink} = l, \text{tj\text{-}stack} = t\text{)) = s} \\
\text{lowlink (empty\text{-}state (}\text{sccs} = s, \text{lowlink} = l, \text{tj\text{-}stack} = t\text{)) = l} \\
\text{tj\text{-}stack (empty\text{-}state (}\text{sccs} = s, \text{lowlink} = l, \text{tj\text{-}stack} = t\text{)) = t} \\
\text{by (simp\text{-}all add: empty\text{-}state\text{-}def)} \\
\end{align*}

\textbf{lemma} \text{sccs\text{-}more\text{-}cong[cong]}:\text{state\text{-}more s = state\text{-}more s' \implies sccs s = sccs s'}
\textbf{by} (cases s, cases s') simp

\textbf{lemma} \text{lowlink\text{-}more\text{-}cong[cong]}:\text{state\text{-}more s = state\text{-}more s' \implies lowlink s = lowlink s'}
\textbf{by} (cases s, cases s') simp

\textbf{lemma} \text{tj\text{-}stack\text{-}more\text{-}cong[cong]}:\text{state\text{-}more s = state\text{-}more s' \implies tj\text{-}stack s = tj\text{-}stack s'}
\textbf{by} (cases s, cases s') simp

\textbf{lemma} \text{[simp]}:
\begin{align*}
s & (\text{state\text{-}more} := (}\text{sccs} = sc, \text{lowlink} = l, \text{tj\text{-}stack} = t\text{)) \\
& = s (\text{sccs} := sc, \text{lowlink} := l, \text{tj\text{-}stack} := t) \\
\text{by (cases s) simp} \\
\end{align*}

\textbf{end}

\textbf{locale} \text{Tarjan\text{-}invar} = \text{Tarjan} + \text{DFS\text{-}invar where} \text{param = tarjan\text{-}params}

\textbf{context} \text{Tarjan\text{-}def begin}
\textbf{lemma} \text{Tarjan\text{-}invar\text{-}eq[ simp]}:
\text{DFS\text{-}invar G tarjan\text{-}params s \leftrightarrow Tarjan\text{-}invar G s (is ?D \leftrightarrow ?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

2.7.3 Correctness Proof

context Tarjan begin

lemma i-tj-stack-discovered:
  is-invar (\lambda s. set (tj-stack s) \subseteq \text{dom} (discovered s))
proof (induct rule: establish-invarI)
case (finish s)
  from finish show ?case
  apply simp
  unfolding tarjan-fin-def
  apply (refine-vcg tj-stack-pop-set-leof-rule)
  apply auto
  done
qed (auto simp add: tarjan-disc-def tarjan-back-def)

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

lemma i-tj-stack-distinct:
  is-invar (\lambda s. \text{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))

197
proof (induct rule: establish-invarI-ND)
case (new-discover s s' v) then interpret Tarjan-invar where s=s by simp

from new-discover tj-stack-discovered have v \notin set (tj-stack s) by auto
moreover {
  fix k j
  assume k < Suc (length (tj-stack s)) j < k
  hence k - Suc 0 < length (tj-stack s) by simp
  hence tj-stack s! (k - Suc 0) \in set (tj-stack s) using nth-mem by metis
  with tj-stack-discovered timing-less-counter have \delta s (tj-stack s! (k - Suc 0)) < counter s by blast
}
moreover {
  fix k j
  define k' where k' = k - Suc 0
  define j' where j' = j - Suc 0

  assume A: k < Suc (length (tj-stack s)) j < k (v#tj-stack s)! j \neq v
  hence gt-0: j > 0 \land k>0 by (cases j=0) simp-all
  moreover with \langle j < k \rangle have j' < k' by (simp add: j'-def k'-def)
  moreover from A have k' < length (tj-stack s) by (simp add: k'-def)
  ultimately have \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 = dropWhile (\neq) u (tj-stack s)
  let ?tw = takeWhile (\neq) u (tj-stack s)

  fix a k j
  assume A: a = tl ?dw k < length a j < k
  and u \in set (tj-stack s)
  hence ?dw \neq [] by auto

  define j' k' where j' = Suc j + length ?tw and k' = Suc k + length ?tw
  with \langle j < k \rangle have j' < k' by simp

  have length (tj-stack s) = length ?tw + length ?dw
    by (simp add: length-append[symmetric])
  moreover from A have \*: Suc k < length ?dw and **: Suc j < length ?dw

198
ultimately have \( k' < \text{length} \ (tj\text{-stack} \ s) \) by (simp add: \( k'\text{-def} \))

with \( j' < k' \) have \( \delta \ s \ (tj\text{-stack} \ s ! k') < \delta \ s \ (tj\text{-stack} \ s ! j') \) by simp
also from dropWhile-nth[\( OF \) *] have \( tj\text{-stack} \ s ! k' = ?dw ! \ Suc \ k \)
by (simp add: \( k'\text{-def} \))
also from dropWhile-nth[\( OF \) **] have \( tj\text{-stack} \ s ! j' = ?dw ! \ Suc \ j \)
by (simp add: \( j'\text{-def} \))
also from nth-tl[\( \langle \ ?dw \neq [] \rangle \) have \( \Suc \ k = a \ ! k \) by (simp add: A)
also from nth-tl[\( \langle \ ?dw \neq [] \rangle \) have \( \Suc \ j = a \ ! j \) by (simp add: A)
finally have \( \delta \ s \ (a ! k) < \delta \ s \ (a ! j) \).

} note aux = this

from finish show \( ?\)case
apply simp
unfolding tarjan-fin-def tj-stack-pop-def
apply refine-vcg
apply (auto intro!: aux)
done
qed (simp-all add: tarjan-back-def)
end end

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

lemma tj-stack-incr-disc:
  assumes \( k < \text{length} \ (tj\text{-stack} \ s) \)
  and \( j < k \)
  shows \( \delta \ s \ (tj\text{-stack} \ s ! j) > \delta \ s \ (tj\text{-stack} \ s ! k) \)
  using assms i-tj-stack-incr-disc[THEN make-invar-thm]
  by blast

lemma tjs-disc-dw-tw:
  fixes \( u \)
  defines \( dw \equiv \text{dropWhile} \ ((\neq) \ u) \ (tj\text{-stack} \ s) \)
  defines \( tw \equiv \text{takeWhile} \ ((\neq) \ u) \ (tj\text{-stack} \ s) \)
  assumes \( x \in \text{set} \ dw \ y \in \text{set} \ tw \)
  shows \( \delta \ s \ x < \delta \ s \ y \)
  proof –
  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 \( \text{length} \ (tj\text{-stack} \ s) = \text{length} \ tw + \text{length} \ dw \)
  by (simp add: length-append[symmetric] tw-def dw-def)
  with \( k \ j \) have \( \delta \ s \ (tj\text{-stack} \ s ! (k + \text{length} \ tw)) < \delta \ s \ (tj\text{-stack} \ s ! j) \)
  by (simp add: tj-stack-incr-disc)
  also from \( j \) takeWhile-nth have \( tj\text{-stack} \ s ! j = y \) by (metis tw-def)
  also from dropWhile-nth \( k \) have \( tj\text{-stack} \ s ! (k + \text{length} \ tw) = x \) by (metis tw-def dw-def)
  finally show \( ?\)thesis .
context Tarjan begin context begin interpretation timing-syntax.

lemma i-sccs-finished-stack-ss-tj-stack:
is-invar (\lambda s. \bigcup (\text{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-invar1)
case (\text{finish } s \ s' \ u) then interpret Tarjan-invar
where
s = s by simp

let \?tw = takeWhile (\neq u) (\text{tj-stack } s)
let \?dw = dropWhile (\neq u) (\text{tj-stack } s)

{ fix x
  assume A: x \neq u x \in \text{set } \?tw u \in \text{set } (\text{tj-stack } s)
  hence x-tj: x \in \text{set } (\text{tj-stack } s) by (auto dest: set-takeWhileD)
  have x \in \text{dom } (\text{finished } 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{stack } s by blast
      moreover
      from A have \?dw \neq [] by simp
      with hd-dropWhile[\text{OF this}] hd-in-set have u \in \text{set } \?dw by metis
      with tjs-disc-dw-tw \langle x \in \text{set } \?tw \rangle 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 stack-not-finished have x \notin \text{dom } (\text{finished } s) by simp
  with A aux-scc[\langle x \neq u \rangle] have x \notin \text{set } \?tw by blast

  moreover
  from \langle x \in \text{set } (\text{stack } s) \rangle have x \in \text{set } (\text{tj-stack } s) by auto

  moreover note tjs-union[\langle u \in \text{set } (\text{tj-stack } s) \rangle]

qed

end end
ultimately have $x \in \text{(tl ?dw)}$ by blast

} note aux-tj = this

from finish show ?case
  apply simp
  unfolding tarjan-fin-def tj-stack-pop-def
  apply (refine-vcg)
  using aux-scc aux-tj apply (auto dest: in-set-tlD)
done
qed (auto simp add: tarjan-disc-def tarjan-back-def)

lemma i-tj-stack-ss-stack-finished:
  is-invar $(\lambda s. \text{set (tj-stack s)} \subseteq \text{set (stack s)} \cup \text{dom (finished s)})$
proof (induct rule: establish-invarI)
  case (finish s) thus ?case
  apply simp
  unfolding tarjan-fin-def
  apply (refine-vcg tj-stack-pop-set-leof-rule)
  apply (simp, cases stack s, simp-all) +
done
qed (auto simp add: tarjan-disc-def tarjan-back-def)

lemma i-finished-ss-sccs-tj-stack:
  is-invar $(\lambda s. \text{dom (finished s)} \subseteq \bigcup (\text{sccs s}) \cup \text{set (tj-stack s)})$
proof (induction rule: establish-invarI-ND)
  case (new-discover s s' v)
  then interpret Tarjan-invar where $s=s$ by simp
  from new-discover finished-discovered show ?case
  by (auto simp add: tarjan-disc-def)
next
  case (finish s s' u)
  then interpret Tarjan-invar where $s=s$ by simp
  from finish show ?case
  apply simp
  unfolding tarjan-fin-def
  apply (refine-vcg tj-stack-pop-set-leaf-rule)
  apply auto
done
qed (simp-all add: tarjan-back-def)
end

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

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

lemma sccs-finished:
  $\bigcup (\text{sccs s}) \subseteq \text{dom (finished s)}$

end
using i-sccs-finished-stack-ss-tj-stack THEN make-invar-thm
by blast

lemma stack-ss-tj-stack:
set (stack s) ⊆ set (tj-stack s)
using i-sccs-finished-stack-ss-tj-stack THEN make-invar-thm
by blast

lemma hd-stack-in-tj-stack:
stack s ≠ [] ⇒ hd (stack s) ∈ set (tj-stack s)
using stack-ss-tj-stack hd-in-set
by auto
end

context Tarjan begin context begin interpretation timing-syntax .
lemma i-no-finished-root:
is-invar (λs. scc-root s r scc ∧ r ∈ dom (finished s) → (∀ x ∈ scc. x \notin set (tj-stack s)))
proof (induct rule: establish-invar-ND-CB)
case (new-discover s s' v) then interpret Tarjan-invar where s=s by simp
{ fix x
  let ?s = s'[state.more := x]

  assume TRANS: \(∀ \Psi . \text{tarjan-disc } v s' \leq_n \text{SPEC } \Psi \implies \Psi x\)
  and inv': DFS-invar G tarjan-params (s'[state.more := x])
  and r: scc-root ?s r scc r ∈ dom (finished s')

  from inv' interpret s': Tarjan-invar where s=?s by simp

  have tj-stack ?s = v#tj-stack s
    by (rule TRANS) (simp add: new-discover tarjan-disc-def)

  moreover from r s'.scc-root-finished-impl-scc-finished have scc ⊆ dom (finished ?s) by auto
  with new-discover finished-discovered have v \notin scc by force

  moreover from r finished-discovered new-discover have r ∈ dom (discovered s) by auto
  with r inv' new-discover have scc-root s r scc
    apply (intro scc-root-transfer[where s'=?s, THEN iiffD2])
    apply clarsimp-all
    done
  with new-discover r have \(∀ x ∈ scc. x \notin set (tj-stack s')\) by simp

  ultimately have \(∀ x\in scc. x \notin set (tj-stack ?s)\) by (auto simp: new-discover)
}
with new-discover show ?case by (simp add: pw-leaf-iff)
next
case (cross-back-edge s s' u v) then interpret Tarjan-invar where s=s by simp
{
  fix x
  let t's = s'[(state.more := x)]
  assume TRANS: \(\Psi : \text{tarjan-back} u v s' \leq_n \text{SPEC} \Psi \implies \Psi x\)
  and r: scc-root t's r scc r \in \text{dom} (finished s')
  with cross-back-edge have scc-root s r scc
  by (simp add: scc-root-transfer[where s'=t's])

  moreover
  have tj-stack t's = tj-stack s by (rule TRANS) (simp add: cross-back-edge tarjan-back-def)

  ultimately have \(\forall x \in \text{scc}. \ x \notin \text{set} (tj-stack ?s)\)
  using cross-back-edge r by simp
}
with cross-back-edge show ?case by (simp add: pw-leaf-iff)
next
case (finish s s' u) then interpret Tarjan-invar where s=s by simp
{
  fix x
  let t's = s'[(state.more := x)]
  assume TRANS: \(\Psi : \text{tarjan-fin} u s' \leq_n \text{SPEC} \Psi \implies \Psi x\)
  and inv': DFS-invar G tarjan-params (s'[state.more := x])
  and r: scc-root t's r scc r \in \text{dom} (finished s')

  from inv' interpret s': Tarjan-invar where s'=?s by simp

  have \(\forall x \in \text{scc}. \ x \notin \text{set} (tj-stack ?s)\)
  proof (cases r = u)
    case False with finish r have \(\forall x \in \text{scc}. \ x \notin \text{set} (tj-stack s)\)
    using scc-root-transfer[where s'=?s]
    by simp
    moreover have set (tj-stack ?s) \subseteq set (tj-stack s)
    apply (rule TRANS)
    unfolding tarjan-fin-def
    apply (refine-vcg tj-stack-pop-set-leaf-rule)
    apply (simp-all add: finish)
    done
    ultimately show ?thesis by blast
  next
    case True with r s'.scc-root-unique-is-scc have scc-root ?s u (scc-of E u) by simp
    with s'.scc-root-transfer[where s'=s] finish have scc-root s' u (scc-of E u)
    by simp

  203
moreover

hence \([\text{simp}]: \text{tj-stack} \ ?s = \text{tl} \ (\text{dropWhile} \ ((\not=) \ u) \ (\text{tj-stack} \ s))\)

apply (rule-tac TRANS)

unfolding tarjan-fin-def tj-stack-pop-def

apply (refine-vcg)

apply (simp-all add: finish)

done

{
let \(?dw = \text{dropWhile} \ ((\not=) \ u) \ (\text{tj-stack} \ s)\)
let \(?tw = \text{takeWhile} \ ((\not=) \ u) \ (\text{tj-stack} \ s)\)
fix \(x\)
define \(j::\text{nat}\) where \(j = 0\)

assume \(x: \ x \in \text{set} \ (\text{tj-stack} \ ?s)\)
then obtain \(i\) where \(i: i < \text{length} \ (\text{tj-stack} \ ?s) \ \text{tj-stack} \ ?s ! i = x\)

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])

with \(i\) have \(\delta \ s \ (\text{tj-stack} \ s ! (\text{Suc} \ i + \text{length} \ ?tw)) < \delta \ s \ (\text{tj-stack} \ s ! \text{length} \ ?tw)\)

by (simp add: tj-stack-incr-disc)

also from \(\text{hd-stack-in-tj-stack}\) finish have \(?ne \not= []\) and \(\text{length} \ ?dw > 0\) by simp-all

from \(\text{hd-dropWhile}[\OF \ ?ne]\ \text{hd-conv-nth}[\OF \ ?ne]\) have \(?dw ! 0 = u\) by simp

with \(\text{dropWhile-nth}[\OF \ (\text{length} \ ?dw > 0)]\) have \(\text{tj-stack} \ s ! \text{length} \ ?tw = u\) by simp

also from \(i\) have \(?dw ! \text{Suc} \ i = x \ \text{Suc} \ i < \text{length} \ ?dw\) by (simp-all add: nth-tl[\OF \ ?ne])

with \(\text{dropWhile-nth}[\OF \ \text{this}(2)]\) have \(\text{tj-stack} \ s ! (\text{Suc} \ i + \text{length} \ ?tw) = x\) by simp

finally have \(\delta \ ?s x < \delta \ ?s u\) by (simp add: finish)

moreover from \(x \ s'.\text{tj-stack-discovered}\) have \(x \in \text{dom} \ (\text{discovered} \ ?s)\) by auto

ultimately have \(x \notin \text{scc}\) using \(s'.\text{scc-root-disc-le \ r \ True}\) by force

} thus \(?thesis\) by metis
qed

} with finish show \(?case\) by (simp add: pw-leaf-iff)

qed simp-all
end

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: scc-root s r (scc-of E u) 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
    assume r ∈ set (tl (stack s))
    with tl-stack-hd-tree-path ne have (r, hd (stack s)) ∈ (tree-edges s)+ by simp
    with trancl-mono-mp tree-edges-ssE have (r, hd (stack s))∈E∗ by (metis rtrancl-eq-or-trancl)

205
context Tarjan begin
lemma i-sccs-are-sccs: is-invar (λs. ∀ scc ∈ sccs s. is-scc E scc)
proof (induction rule: establish-invar1)
case (finish s s' u) then interpret Tarjan-invar where s=s by simp
from finish have EQ[simp]:
  finished s' = (finished s)(u ↦→ counter s)
  discovered s' = discovered s
  tree-edges s' = tree-edges s
  sccs s' = sccs s
  tj-stack s' = tj-stack s
by simp-all

{ fix x

let ?s = s'[(state.more := x)]
assume TRANS: \∃!Ψ. tarjan-fin u s' ≤n SPEC Ψ \implies Ψ x
and inv': DFS-invar G tarjan-params (s'[(state.more := x)])
then interpret s': Tarjan-invar where s=?s by simp

from finish hd-in-set stack-set-def have
  u-disc: u ∈ dom (discovered s)
  and u-n-fin: u /∈ dom (finished s) by blast+

have ∀ scc ∈ sccs ?s. is-scc E scc
proof (cases scc-root s' u (scc-of E u))
case False
have sccs ?s = sccs s
  apply (rule TRANS s)
unfolding \texttt{tarjan-fin-def tj-stack-pop-def}
by \texttt{(refine-vcg) (simp-all add: False)}
thus \texttt{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 \texttt{tarjan-fin-def tj-stack-pop-def}
by \texttt{(refine-vcg) (simp-all add: True)}

have \([\text{simp}]: \text{tj-stack } ?s = \text{tl } ?dw\)
apply (rule TRANS)
unfolding \texttt{tarjan-fin-def tj-stack-pop-def}
by \texttt{(refine-vcg) (simp-all add: True)}

from True scc-root-transfer[\text{where } s' = s'] have scc-root s u (scc-of E u) by simp
with inv' scc-root-transfer[\text{where } s' = s'] u-disc have u-root: scc-root ?s u (scc-of E u) by simp

have \(?tw' \subseteq \text{scc-of } E u\)
proof
fix v assume v': v \in \?tw'
show v \in \text{scc-of } E u
proof cases
assume v \neq u with v have v': 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-tl-stack-hd-discover finish have \( \delta s v < \delta s u \) by simp
with \( \delta s v > \delta s u \) show \text{False by force}
qed

ultimately have \( (u, v) \in (\text{tree-edges } s)^+ \)
using parenthesis-impl-tree-path-not-finished[\text{OF u-disc}] u-n-fin by force

207
with trancl-mono-mp tree-edges-ssE have \((u,v) \in E^*\) by (metis trancl-eq-or-trancl)

moreover
from tj-stack-reach-hd-stack v-tj finish have \((v,u) \in E^*\) by simp

moreover have is-scc \(E\) (scc-of \(E\) u) \(u \in\) scc-of \(E\) u by simp-all
ultimately show ?thesis using is-scc-closed by metis
qed simp

qed
moreover have scc-of \(E\) u \(\subseteq \) ?tw'
proof
fix v assume v: v \(\in\) scc-of \(E\) u
moreover note u-root
moreover have u \(\in\) dom (finished ?s) by simp
ultimately have \(v \in\) dom (finished ?s) \(v \notin\) set (tj-stack ?s)
using s'.scc-root-finished-impl-scc-finished s'.no-finished-root by auto
with s'.finished-ss-sccs-tj-stack have v \(\in\) \(\bigcup\) (sccs ?s) by blast
hence v \(\in\) \(\bigcup\) (sccs s) \(\lor\) 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 add: tarjan-back-def tarjan-disc-def)
end

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

context begin interpretation timing-syntax .

lemma i-lowlink-eq-LowLink:
is-invar \((\lambda s. \forall x \in\) dom (discovered s). \(\zeta\) 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: \( \land w. \{ Tarjan-invar G ?s; w \in \text{dom} (\text{discovered } ?s); w \neq v \} \implies \text{lowlink-set } ?s w \subseteq \text{lowlink-set } s w \cup \{ v \} \)  
  assume tree-sub : \( \text{tree-edges } s' = \text{tree-edges } s \lor (\exists u. u \neq v \land \text{tree-edges } s' = \text{tree-edges } s \cup \{ (u,v) \}) \)  

  assume Tarjan-invar G s  
  assume [simp]: \( \text{discovered } s' = (\text{discovered } s)(v \mapsto \text{counter } s) \)  
  finished s' = finished s  
  lowlink s' = lowlink s  
  cross-edges s' = cross-edges s \( \land \) back-edges s' = back-edges s  
  assume v-n-disc: \( v \notin \text{dom} (\text{discovered } s) \)  
  assume IH: \( \land w. w \in \text{dom} (\text{discovered } s) \implies \zeta s w = \text{LowLink } s w \)  

  assume TRANS: \( \land \Psi. \text{tarjan-disc } v s' \leq_n \text{SPEC } \Psi \implies \Psi x \)  
  and INV: \( \text{DFS-invar G } \text{tarjan-params } ?s \)  
  and w-disc: \( w \in \text{dom} (\text{discovered } ?s) \)  

  interpret Tarjan-invar where \( s = s \) by fact  
  from INV interpret \( s' : \text{Tarjan-invar where } s = ?s \) by simp  

  have [simp]: \( \text{lowlink } ?s = (\text{lowlink } s)(v \mapsto \text{counter } s) \)  
  by (rule TRANS) (auto simp: tarjan-disc-def)  

  from v-n-disc edge-imp-discovered have edges s' \( \{ v \} = \{ \} \) by auto  
  with tree-sub edge-imp-discovered have edges ?s' \( \{ v \} = \{ \} \) by auto  
  with s'.no-path-imp-no-lowlink-path have \( \land w. \neg(\exists p. \text{lowlink-path } ?s v p w) \)  
  by metis  

  hence ll-v: \( \text{lowlink-set } ?s v = \{ v \} \)  
  unfolding lowlink-set-def by auto  

  have \( \zeta ?s w = \text{LowLink } ?s w \)  
  proof (cases \( w = v \))  
    case True with ll-v show \(?thesis \) by simp  
  next  
    case False hence \( \zeta ?s w = \zeta s w \) by simp  
    also from IH have \( \zeta s w = \text{LowLink } s w \) using w-disc False by simp  
    also have \( \text{LowLink } s w = \text{LowLink } ?s w \)  
    proof (rule LowLink-eqI[OF INV])  
      from v-n-disc show discovered s \( \subseteq_m \) discovered ?s by (simp add: map-le-def)  
    from tree-sub show \( \text{lowlink-set } s w \subseteq \text{lowlink-set } ?s w \)  
  
  
\}
show \textit{lowlink-set} \( \exists \ w \subseteq \textit{lowlink-set} \ s \cup \{v\} \)

**proof** (cases \( w = v \))

- **case** \( \text{True with ll-}v \) **show** \(?\text{thesis}\) by auto

**next**

- **case** \( \text{False thus } \ ?\text{thesis}\)
  
  **using** \( \text{pre-ll-sub-rev} w \text{-disc INV} \)

  by simp

**qed**

**show** \( w \in \text{dom (discovered } s \text{)} \) **using** \( w \text{-disc False by simp} \)

**fix** \( ll \) **assume** \( ll \in \{v\} \) **with** \text{timing-less-counter} \text{lowlink-set-discovered} **have**

\( \forall x. x \in \delta \ 	ext{lowlink-set} \ s \ w \implies x < \delta \ ?s \ ll \) by simp force

**moreover from** \text{Min-in} \text{lowlink-set-finite} \text{lowlink-set-not-empty} \text{w-disc} **have**

\( \text{LowLink } s \ w \in \delta \ 	ext{lowlink-set} \ s \ w \) by auto

ultimately **show** \( \text{LowLink } s \ w \leq \delta \ ?s \ ll \) by force

**qed**

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

**qed**

\( \text{note} \ 	ext{tarjan-disc-aux} = \text{this} \)

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

**proof** (**induct rule: establish-invarI-CB**)

- **case** \( \text{new-root } s \ s' \ v0 \)

  \( \{ \)

  **fix** \( w \) \( x \)

  **let** \( ?s = \text{new-root } v0 \ s(\text{state} : = x) \)

  **have** \( \text{lowlink-set } ?s \ w \subseteq \text{lowlink-set} \ s \ w \cup \{v0\} \)

  unfolding \( \text{lowlink-set-def lowlink-path-def} \)

  by auto

  \( \} \text{ note } * = \text{this} \)

**from** \text{new-root} **show** \( \text{?case} \)

**using** \text{tarjan-disc-aux[OF *]} \n
by (auto simp add: \text{pw-leaf-iff})

**next**

- **case** \( \text{discover } s \ s' \ w \) **then interpret** \text{Tarjan-invar where} \( s = s \text{ by simp} \)

  **let** \( ?s' = \text{discover (hd (stack } s\text{))} \ v \ (s(\text{pending} : = \text{pending } s \ - \ {(\text{hd (stack } s\text{)}, v)}) \}) \)

  \( \{ \)

  **fix** \( w \) \( x \)

  **let** \( ?s = ?s'(\text{state} : = x) \)

  **assume** \( \text{INV: Tarjan-invar } G ?s \)

  **and** \( d : w \in \text{dom (discovered } ?s') \)

210
and \( w \neq v \)

**interpret** \( s' \): Tarjan-invar **where** \( s = s' \) **by** fact

**have** lowlink-set \( s \subseteq \text{lowlink-set } s \cup \{v\} \)

**proof**

**fix** \( ll \)

**assume** \( ll : ll \in \text{lowlink-set } s \)

**hence** \( ll = w \lor (\exists p. \text{lowlink-path } s \ w \ p \ ll) \) **by** \( \text{(auto simp add: lowlink-set-def)} \)

**thus** \( ll \in \text{lowlink-set } s \cup \{v\} \) **(is \( ll \in ?L \))**

**proof**

**assume** \( ll = w \) **with** \( d \)

**show** ?thesis **by** \( \text{(auto simp add: lowlink-set-def)} \)

**next**

**assume** \( \exists p. \text{lowlink-path } s \ w \ p \ ll \)

**then guess** \( p \)

.. note \( p = \text{this} \)

**hence** \[ \text{(simp): } p \neq[] \] **by** \( \text{(simp add: lowlink-path-def)} \)

**from** \( p \) **have** \( \text{hd } p = w \) **by** \( \text{(auto simp add: lowlink-path-def path-hd)} \)

**show** ?thesis

**proof** \( \text{(rule tri-caseE)} \)

**assume** \( v \neq ll \) **v \not\notin set p**

**hence** \( \text{lowlink-path } s \ w \ p \ ll \)

**using** \( p \) **by** \( \text{(auto simp add: lowlink-path-def)} \)

**with** \( ll \) **show** ?thesis **by** \( \text{(auto simp add: lowlink-set-def)} \)

**next**

**assume** \( v = ll \) **thus** ?thesis **by** simp

**next**

**assume** \( v \in set p \) **v \neq ll \)

**then obtain** \( i \) **where** \( i : i < \text{length } p \ | i = v \)

**by** \( \text{(metis in-set-conv-nth)} \)

**have** False

**proof** \( \text{(cases i)} \)

**case** 0 **with** \( i \) **have** \( \text{hd } p = v \) **by** \( \text{(simp add: hd-conv-nth)} \)

**with** \( \langle \text{hd } p = w \rangle \) **w \neq v \) **show** False **by** simp

**next**

**case** (Suc \( n \)) **with** \( i \) \( s'.\text{lowlink-path-finished}[OF p, \text{where } j=i] \) **have** \( v \in \text{dom (finished } ?s)\) **by** simp

**with** \( \text{finished-discovered } discover \) **show** False **by** auto

qed

**thus** ?thesis ..

qed

qed

} note * = this

**from** discover \( \text{hd-in-set stack-set-def} \) **have** \( v \neq u \) **by** auto

**with** discover **have** **=:** tree-edges \( ?s' = \text{tree-edges } s \lor (\exists u. \ u \neq v \land \text{tree-edges} \)

211
\[s' = \text{tree-edges } s \cup \{(u,v)\}\] by auto

from discover show ?case
   using tarjan-disc-aux[OF **]
   by (auto simp: pw-leaf-off)
next
   case (cross-back-edge \(s'\ u\ v\)) then interpret Tarjan-invar where \(s=s\) by simp
from cross-back-edge have [simp]:
   discovered \(s' = \text{discovered } s\)
   finished \(s' = \text{finished } s\)
   tree-edges \(s' = \text{tree-edges } s\)
   lowlink \(s' = \text{lowlink } s\)
   by simp-all
{
  fix \(w::'v\)
  fix \(x\)

  let \(?s = s'(\text{state.more} := x)\)
  let \(?L = \delta s\ \text{lowlink-set } s\ w\)
  let \(?L' = \delta ?s\ \text{lowlink-set } ?s\ w\)

  assume TRANS: \(\forall \Psi. \text{tarjan-back } u\ v\ s' \leq_n \text{SPEC } \Psi \longrightarrow \Psi x\)
  and inv': DFS-invar \(G\) \text{tarjan-params } ?s
  and w-disc': \(w \in \text{dom (discovered } ?s)\)

  from inv' interpret \(s':\text{Tarjan-invar where } s=?s\) by simp
  have ll-sub: \(\text{lowlink-set } s\ w \subseteq \text{lowlink-set } ?s\ w\)
     unfolding lowlink-set-def lowlink-path-def
     by (auto simp: cross-back-edge)
  have ll-sub-rev: \(\text{lowlink-set } ?s\ w \subseteq \text{lowlink-set } s\ w \cup \{v\}\)
     unfolding lowlink-set-def lowlink-path-def
     by (auto simp: cross-back-edge)

  from w-disc' have w-disc: \(w \in \text{dom (discovered } s)\) by simp
  with LowLink-le-disc have LLw: \(\text{LowLink } s\ w \leq \delta s\ w\) by simp

  from cross-back-edge hd-in-set have u-n-fin: \(u \notin \text{dom (finished } s)\)
     using stack-not-finished by auto
{
  assume s: \(v \in \text{lowlink-set } ?s\ w \implies \text{LowLink } s\ w \leq \delta ?s\ v\)
  have LowLink \(s\ w = \text{LowLink } ?s\ w\)
  proof (rule LowLink-eqI[OF inv' - ll-sub ll-sub-rev w-disc])
    show discovered \(s \subseteq_m \text{discovered } ?s\) by simp
    fix ll assume ll \(\in \{v\}\) ll \(\in \text{lowlink-set } ?s\ w\)
with * show \( \text{LowLink } s \ w \leq \delta \) \( ?s \ ll \) 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 \( \delta s v < \delta s w \wedge v \in \text{set} \ (tj-stack \ s) \wedge \delta s v < \zeta s w \) )
  case False note all-False = this 
  with \( \langle w = u \rangle \) have \( \zeta s w \) by (rule-tac TRANS) (auto simp add: tarjan-back-def cross-back-edge)

next

  case True with all-True = this 
  with \( \langle w = u \rangle \) have \( \zeta s w \) by (rule-tac TRANS) (auto simp add: tarjan-back-def cross-back-edge)

    with finished-ss-sscs-tj-stack v-n-tj: \( v \notin \text{set} \ (stack \ s) \) by auto
    with cross-back-edge have \( v \notin \text{set} \ (tj-stack \ s) \) by auto

    from cb \( s \cdot \text{cross-edges-ssE} \) \( s' \cdot \text{back-edges-ssE} \) have \( (u,v) \in \) auto

qed

finally show ?thesis .

next

  case True note all-True = this 
  with \( \langle w=u \rangle \) 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 \( \delta s v = \text{Min} \) \( (?L \cup \{ \delta s v \}) \) by simp

    also have \( v \in \text{lowlink-set } ?s \ w \) 

    proof

      have cb: \( (u,v) \in \text{cross-edges } ?s \cup \text{back-edges } ?s \) by (simp add: cross-back-edge)
      moreover from cb \( s' \cdot \text{cross-edges-ssE} \) \( s' \cdot \text{back-edges-ssE} \) have \( (u,v) \in \) auto

  qed
\[ E \text{ by blast} \]

\[
\text{hence } (u,v) \in E^* \ldots \\
\text{moreover from all-True tj-stack-reach-hd-stack have } (v,u) \in E^* \text{ by (simp add: cross-back-edge)} \\
\text{moreover note } (v \in \text{dom } (\text{discovered } s)) \\
\text{ultimately show } \text{thesis by (auto intro: s'.lowlink-setI simp: } w=w \text{)} \\
\text{qed} \\
\text{with ll-sub ll-sub-rev have lowlink-set } s w = \text{lowlink-set } s w \cup \{v\} \text{ by auto} \\
\text{hence } \text{thesis} \text{ by simp} \\
\text{finally show } \text{thesis} \text{.} \\
\text{qed} \\
\text{next} \\
\text{case False} \quad w \neq u \\
\text{hence } \zeta s w = \zeta s w \\
\text{by (rule-tac TRANS) (simp add: tarjan-back-def cross-back-edge)} \\
\text{also have } \zeta s w = \text{LowLink } s w \text{ using } w\text{-disc False by (simp add: cross-back-edge)} \\
\text{also have } \text{LowLink } s w = \text{LowLink } s w \\
\text{proof (rule LL-eqI)} \\
\text{assume } v : v \in \text{lowlink-set } s w \\
\text{thus } \text{LowLink } s w \leq \delta s v \text{ using } LLw \\
\text{proof cases} \\
\text{assume } v \neq w \\
\text{with } v \text{ obtain } p \text{ where } p : \text{lowlink-path } s w p p v \neq [] \\
\text{by (auto simp add: lowlink-set-def lowlink-path-def)} \\
\text{hence } \text{hd } p = w \text{ by (auto simp add: lowlink-path-def path-hd)} \\
\text{show } \text{thesis} \\
\text{proof (cases } u \in \text{ set } p) \\
\text{case False with last-in-set } p \text{ cross-back-edge have last } p \neq \text{hd } (\text{stack } s) \text{ by force} \\
\text{with } p \text{ have } \text{lowlink-path } s w p v \\
\text{by (auto simp: cross-back-edge lowlink-path-def)} \\
\text{with } v \text{ have } v \in \text{lowlink-set } s w \\
\text{by (auto intro: lowlink-setI simp: lowlink-set-def cross-back-edge)} \\
\text{thus } \text{thesis by simp} \\
\text{next} \\
\text{case True then obtain } i \text{ where } i : i < \text{length } p \text{ p!i } u \\
\text{by (metis in-set-conv-nth)} \\
\text{have } False \\
\text{proof (cases } i) \\
\text{case 0 with } i \text{ have } \text{hd } p = u \text{ by (simp add: hd-conv-nth)} \\
\text{with } (\text{hd } p = w \text{) } w \neq u \text{ show False by simp} \\
\text{next} \\
\text{case } (\text{Suc } n) \text{ with } i s'.\text{lowlink-path-finished}[OF } p(1), \text{ where } j=i \\
\text{have } u \in \text{dom } (\text{finished } s) \text{ by simp} \\
\text{with } u\text{-n-fin show } \text{thesis by simp} \\
\text{214} \]
qed
thus ?thesis ..
qed
qed simp
qed
finally show ?thesis .
qed
}

note aux = this

with cross-back-edge show ?case by (auto simp: pw-leaf-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: ∀Ψ. tarjan-fin u s' ≤n SPEC Ψ ⇒ Ψ 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 ∈ lowlink-set ?s w
      hence ll = w ∨ (∃p. lowlink-path ?s w p ll) by (auto simp add: lowlink-set-def)

  215
thus \( \ell \in \text{lowlink-set} \ s \ w \cup \text{lowlink-set} \ s \ u \)

proof (rule disjE1)

assume \( \ell = w \) with \( w \)-disc show \(?thesis\) by (auto simp add: lowlink-set-def)

next

assume \( \ell \neq w \)

assume \( \exists \ p. \ \text{lowlink-path} \ ?s \ w \ p \ \ell \)

then guess \( \ p \) .. note \( \ p = \) 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 \(?thesis\)

proof (cases \( u \in \text{set} \ p \))

\begin{itemize}
  \item case False hence \( \text{lowlink-path} \ s \ w \ p \ \ell \) using \( p \) by (auto simp add: lowlink-path-def)
  \item case True then obtain \( \ i \) where \( i \ < \text{length} \ p \ \text{p!} \ i = u \)

by (metis in-set-conv-nth)

moreover let \( \text{dp} = \text{drop} \ i \ p \)

from \( i \) have \( \text{dp} \neq [] \) by simp

from \( i \) have \( \text{hd} \ \text{dp} = u \) by (simp add: hd-drop-conv-nth)

moreover from \( i \) have \( \text{last} \ \text{dp} = \text{last} \ p \) by simp

moreover {
fix \( k \)
assume \( 1 < \text{length} \ \text{dp} \)
and \( k < \text{length} \ \text{dp} - 1 \)

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 \( (?dp!k, ?dp!\text{Suc} \ k) \in \text{tree-edges} \ s \) by (simp add: add.commute)

\} note \( \text{aux} = \text{this} \)

moreover {
assume \( \ast: \ 1 < \text{length} \ \text{dp} \)

hence \( l: \ 1 + i < \text{length} \ p \) by simp

with \( s'.\text{lowlink-path-finished}(\OF \ p) \) have \( p \ ! (1+i) \in \text{dom} \ (\text{finished} \ ?s) \) by auto

moreover from \( l \) have \( \text{i+1}\leq\text{length} \ p \) by simp

ultimately have \( ?dp!1 \in \text{dom} \ (\text{finished} \ ?s) \) by simp
\end{itemize}
moreover from aux[of 0] * have (?dp!0, ?dp!Suc 0) ∈ tree-edges s by simp

with (hd ?dp = w) hd-conv-nth[of ?dp] * have (u, ?dp!Suc 0) ∈ tree-edges s by simp

with no-self-loop-in-tree have ?dp!1 ≠ u by auto
ultimately have ?dp!1 ∈ dom (finished s) by simp

moreover
case p of
have P: path E w p ll by (simp add: lowlink-path-def)

have p = (take i p)@[?dp by simp

(take i p) x by metis

with (?dp ≠ []) path-hd have hd ?dp = x by metis

with ⟨hd ?dp = w⟩ p' have u-path: path E u ?dp ll and path-u: path E w (take i p) u by metis+

ultimately have lowlink-path s u ?dp ll using p by (simp add: lowlink-path-def)

moreover from u-path path-is-transcl (?dp ≠ []) have (u, ll) ∈ E⁺ by force

moreover { from ll (ll ≠ w) have (ll, w) ∈ E⁺ by (auto simp add: lowlink-set-def)

also from path-u path-is-rtranscl have (w, u) ∈ E⁺ by metis

finally have (ll, u) ∈ E⁺ .
}
moreover note ll u-disc
ultimately have ll ∈ lowlink-set s u unfolding lowlink-set-def by auto

thus ?thesis by auto
qed
qed

gold hence ll-sub-rev': ?L' ⊆ ?L ∪ ?Lu by auto

have ref-ne: stack ?s ≠ [] ⇒ lowlink ?s = (lowlink s)(hd (stack ?s) ↦→ min (ζ s (hd (stack ?s)))) (ζ s u))

apply (rule TRANS)
unfolding tarjan-fin-def tj-stack-pop-def
by refine-vcg simp-all

have ref-c: stack ?s = [] ⇒ lowlink ?s = lowlink s
apply (rule TRANS)
unfolding tarjan-fin-def tj-stack-pop-def
by refine-vcg simp-all

have ref-tj: ζ s u ≠ δ s u ⇒ tj-stack ?s = tj-stack s
apply (rule TRANS)
unfolding tarjan-fin-def tj-stack-pop-def
by refine-eqv simp-all

have ζ ?s w = LowLink ?s w
proof (cases w = hd (stack ?s) ∧ stack ?s ≠ [])
case True note all-True = this
with refine have *; ζ ?s w = min (ζ s w) (ζ s u) by simp
show ?thesis
proof (cases ζ s u < ζ s w)
case False with * finish w-disc have ζ ?s w = LowLink s w by simp
also have LowLink s w = LowLink ?s w
proof (rule LowLink-eqI [OF inv′ - ll-sub ll-sub-rev])
  from w-disc show w ∈ dom (discovered s) by simp
fix ll assume ll ∈ lowlink-set s u
  hence LowLink s u ≤ δ s ll by simp
moreover from False finish w-disc u-disc have LowLink s w ≤ LowLink s u by simp
ultimately show LowLink s w ≤ δ ?s ll by simp
qed simp
finally show ?thesis.
next
case True note ζ rel = this
have LowLink s u ∈ $?L′
proof
  from all-True finish have w-tl: w∈set (tl (stack s)) by auto
  obtain ll where ll: ll ∈ lowlink-set s u δ s ll = LowLink s u
  by fastforce
  have ll ∈ lowlink-set ?s w
proof (cases δ s u = ζ s u)
case True
  moreover from w-tl finish tl-tl-stack-hd-discover have δ s w < δ s u by simp
moreover from w-disc have LowLink s w ≤ δ s w by (simp add: LowLink-te-disc)
with w-disc finish have ζ s w ≤ δ s w by simp
moreover note ζ rel
ultimately have False by force
thus ?thesis ..
next
case False with u-disc finish ll have u ≠ ll by auto
with ll have
c: (ll,u) ∈ E⁺ (u,ll) ∈ E⁺ and
p: ∃ p. lowlink-path s u p ll and
ll-disc: ll ∈ dom (discovered s)
by (auto simp: lowlink-set-def)
from p have p': ∃ p. lowlink-path ?s u p ll

218
unfolding lowlink-path-def
by auto
from w-tl tl-stack-hd-tree-path finish have T: (w,u) ∈ (tree-edges ?s)+ by simp
with s’.lowlink-path-tree-prepend all-True p’ have ∃ p. lowlink-path
?s w p ll by blast
moreover from T trancl-mono-mp[OF s’.tree-edges-ssE] have (w,u) ∈ E+ by blast
with e have (w,ll) ∈ E+ by simp
moreover { note e(1)
also from finish False ref-tj have tj-stack ?s = tj-stack s by simp
with hd-in-set finish stack-ss-tj-stack have u ∈ set (tj-stack ?s) by auto
with s’.tj-stack-reach-stack obtain x where x ∈ set (tj-stack ?s)
(u,x) ∈ E+ by blast
note this(2)
also have (x,w) ∈ E+ proof (rule rtrancl-eq-or-trancl[THEN iffD2], safe)
assume x ≠ w with all-True x have x ∈ set (tl (stack ?s)) by blast
(cases stack ?s) auto
with s’.tl-stack-hd-tree-path all-True have (x,w) ∈ (tree-edges ?s)+
by auto
with trancl-mono-mp[OF tree-edges-ssE] show (x,w) ∈ E+ by simp
qed
finally have (ll,w) ∈ E+ .
}
moreover note ll-disc
ultimately show ?thesis by (simp add: lowlink-set-def)
qed
hence δ s ll ∈ ?L’ by auto
with ll show ?thesis by simp
qed
hence LowLink ?s w ll ≤ LowLink s u using Min-le-iff[of ?L’] s’.lowlink-set-not-empty w-disc s’.lowlink-set-finite
by fastforce
also from True u-disc w-disc finish have LowLink s u < LowLink s w by simp
w-disc by simp
hence LowLink s u ≤ LowLink ?s w using Min-antimono[OF ll-sub-rev’] lowlink-set-finite s’.lowlink-set-not-empty
w-disc by auto
also from True u-disc finish * have LowLink s u = ζ ?s w by simp
finally show ?thesis ..
qed

next
case False note all-False = this
have ζ ?s w = ζ s w
proof (cases stack ?s = [])
  case True with ref-e show ?thesis by simp
next
case False with all-False have w ≠ hd (stack ?s) by simp
with False ref-ne show ?thesis by simp
qed
also from finish have ζ s w = LowLink s w using w-disc by simp

also {
  fix v
  assume v ∈ lowlink-set s u
  and *: v ∈ lowlink-set s w
  hence v ≠ w w ≠ u by (auto simp add: lowlink-set-def)
  have v ∈ lowlink-set ?s w
proof (rule notI)
  assume v: v ∈ lowlink-set ?s w
  hence e: (v, w) ∈ E* (w, v) ∈ E*
  and v-disc: v ∈ dom (discovered s) by (auto simp add: lowlink-set-def)

from v ⟨w ≠ w⟩ obtain p where p: lowlink-path ?s w p v by (auto simp add: lowlink-set-def)
  hence [simp]: p ≠ [] by (simp add: lowlink-path-def)

from p have hd p = w by (auto simp add: lowlink-path-def path-hd)

show False
proof (cases u ∈ set p)
  case False hence lowlink-path s w p v
    using p by (auto simp add: lowlink-path-def)
  with e v-disc have v ∈ lowlink-set s w by (auto intro: lowlink-setI)
  with * show False ..
next
case True
then obtain i where i: i < length p p!i = u
  by (metis in-set-conv-nth)
show False
proof (cases i)
  case 0 with i have hd p = u by (simp add: hd-conv-nth)
  with ⟨hd p = w⟩ ⟨w ≠ u⟩ show False by simp
next
case (Suc n) with i p have *: (p!n, u) ∈ tree-edges s n < length p
    unfolding lowlink-path-def
  by auto
  with tree-edge-imp-discovered have p!n ∈ dom (discovered s) by auto
moreover from finish hd-in-set stack-not-finished have u ∉ 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 * 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 s-ne: stack ?s \neq [] by auto

  with all-False have w \neq \text{hd} (stack ?s) by simp

from stack-is-tree-path finish obtain v\emptyset where

  path (tree-edges s) v\emptyset (rev (stack ?s)) u

by auto

  with s-ne have (hd (stack ?s), u) \in tree-edges s by (auto simp:

  neg-Nil-conv path-simps)

  with * tree-eq-rule have **: \text{hd} (stack ?s) = p\!n by simp

  show ?thesis

proof (cases n)

  case 0 with * have hd p = p\!n by (simp add: \text{hd-conv-nth})

  with (hd p = w) ** have w = \text{hd} (stack ?s) by simp

  with (w \neq \text{hd} (stack ?s)) show False ..

next

  case (Suc m) with * ** s'.lowlink-path-finished[OF p, where

  \ j=n] have

  \text{hd} (stack ?s) \in \text{dom} (finished ?s) by simp

  with \text{hd-in-set}[OF s-ne] s'.stack-not-finished show ?thesis by blast

q̄d

q̄d

q̄d

} with ll-sub ll-sub-rev have lowlink-set ?s w = lowlink-set s w by auto

hence LowLink s w = LowLink ?s w by simp

finally show ?thesis .

qed

with finish show ?case by (auto simp: pw-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} (finished s) \lor (stack s \neq [] \land v = \text{hd} (stack s) \land pending s

  " \{v\} = \{\})

  shows \\zeta s v = \delta s v \longleftrightarrow \text{scc-root s v (scc-of E v)}

proof –
from assms have \( v \in \text{dom (discovered s)} \) 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: \( \lnot \ \text{cond s} \)

shows reachable = \( \bigcup (\text{sccs s}) \)

proof

from nc-finished-eq-reachable NC have \( \text{simp} \): reachable = \( \text{dom (finished s)} \)

by simp

with \( \text{sccs-finished} \) show \( \bigcup (\text{sccs s}) \subseteq \text{reachable} \) by simp

from NC have stack s = [] by (simp add: cond-alt)

with \( \text{stacks-eq-iff} \) have \( \text{tj-stack s} = [] \) by simp

with \( \text{finished-ss-sccs-tj-stack} \) show reachable \( \subseteq \bigcup (\text{sccs s}) \) by simp

qed

end end

context Tarjan begin

lemma tarjan-fin-nofail:

assumes pre-on-finish u s'

shows nofail (tarjan-fin u s')

proof

from \\
\begin{align*}
\text{assms} \ \text{obtain} \ s \ \text{where} & \ s: \text{DFS-invar G tarjan-params s stack s} \\
& \neq [] \ \text{by (auto simp: pre-on-finish-def)}
\end{align*}

\( u = \text{hd (stack s)} \ s' = \text{finish u s cond s pending s} " \ \{u\} = \{\} \)

by (auto simp: pre-on-finish-def)

then interpret Tarjan-invar where \\
\( s=s \) by simp

from s hd-stack-in-tj-stack have u \( \in \text{set (tj-stack s')} \) by simp

moreover from s tj-stack-distinct have distinct (tj-stack s') by simp

moreover have \( (\text{lowlink s'} u) = (\text{discovered s'} u) \leftrightarrow \text{scc-root s'} u \)

(scc-of E u)

proof

from s have \( (\text{lowlink s'} u) = (\text{discovered s'} u) \leftrightarrow (\text{lowlink s u}) \)

by (simp add: cond-alt)

also from \( s \) lowlink-eq-disc-iff-scc-root have \( \ldots \leftrightarrow \text{scc-root s u} \)

(scc-of E u)

by blast

also from \( s \) scc-root-transfer"[where \( s'=s \)] \( scc-root s' u \)

by blast

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

qed

ultimately show \( \text{thesis} \)

unfolding tarjan-fin-def tj-stack-pop-def

by simp

qed
**sublocale** DFS G tarjan-params
  
  **by** unfold-locales (simp-all add: tarjan-disc-def tarjan-back-def tarjan-fin-nofail)

**end**

**interpretation** tarjan: Tarjan-def for G .

### 2.7.4 Interface

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

**definition** tarjan-spec G ≡ do {
  ASSERT (fb-graph G);
  SPEC (λsccs. (∀scc ∈ sccs. is-scc (g-E G) scc)
           ∧ ∪ sccs = tarjanreachable 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])
  assumefb-graph G
  then interpret fb-graph G .
  interpret Tarjan ..
  show DFS G (tarjan.tarjan-params TYPE('b) G) ..
next
  fix s
  assumeC: 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 sccs-are-sccs show ∀ scc ∈ sccs s. is-scc (g-E G) scc .

from ne-sccs-eq-reachable C show ∪ (sccs s) = tarjanreachable TYPE('b) G
  by simp

qed

**end**