A Framework for Verifying Depth-First Search Algorithms

Peter Lammich and René Neumann

February 23, 2021
Abstract

This entry presents a framework for the modular verification of DFS-based algorithms, which is described in our [CPP-2015] paper. It provides a generic DFS algorithm framework, that can be parameterized with user-defined actions on certain events (e.g. discovery of new node).

It comes with an extensible library of invariants, which can be used to derive invariants of a specific parameterization.

Using refinement techniques, efficient implementations of the algorithms can easily be derived. Here, the framework comes with templates for a recursive and a tail-recursive implementation, and also with several templates for implementing the data structures required by the DFS algorithm.

Finally, this entry contains a set of re-usable DFS-based algorithms, which illustrate the application of the framework.

Contents

1 The DFS Framework 3
  1.1 General DFS with Hooks ................................. 3
    1.1.1 State and Parameterization ....................... 3
    1.1.2 DFS operations .................................. 4
    1.1.3 DFS Algorithm .................................... 9
    1.1.4 Invariants ....................................... 10
    1.1.5 Basic Invariants .................................. 19
    1.1.6 Total Correctness .................................. 24
    1.1.7 Non-Failing Parameterization ..................... 26
  1.2 Basic Invariant Library ................................. 28
    1.2.1 Basic Timing Invariants ......................... 28
    1.2.2 Parenthesis Theorem ............................... 32
    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 .................................. 119

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
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2</td>
<td>Finding a Path between Nodes</td>
<td>131</td>
</tr>
<tr>
<td>2.2.1</td>
<td>Including empty Path</td>
<td>132</td>
</tr>
<tr>
<td>2.2.2</td>
<td>Restricting the Graph</td>
<td>136</td>
</tr>
<tr>
<td>2.2.3</td>
<td>Path of Minimal Length One, with Restriction</td>
<td>138</td>
</tr>
<tr>
<td>2.2.4</td>
<td>Path of Minimal Length One, without Restriction</td>
<td>141</td>
</tr>
<tr>
<td>2.2.5</td>
<td>Implementation</td>
<td>141</td>
</tr>
<tr>
<td>2.2.6</td>
<td>Synthesis of Executable Code</td>
<td>144</td>
</tr>
<tr>
<td>2.2.7</td>
<td>Conclusion</td>
<td>147</td>
</tr>
<tr>
<td>2.3</td>
<td>Set of Reachable Nodes</td>
<td>148</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Preliminaries</td>
<td>148</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Framework Instantiation</td>
<td>149</td>
</tr>
<tr>
<td>2.3.3</td>
<td>Correctness</td>
<td>149</td>
</tr>
<tr>
<td>2.3.4</td>
<td>Synthesis of Executable Implementation</td>
<td>151</td>
</tr>
<tr>
<td>2.3.5</td>
<td>Conclusions</td>
<td>154</td>
</tr>
<tr>
<td>2.4</td>
<td>Find a Feedback Arc Set</td>
<td>154</td>
</tr>
<tr>
<td>2.4.1</td>
<td>Instantiation of the DFS-Framework</td>
<td>155</td>
</tr>
<tr>
<td>2.4.2</td>
<td>Correctness Proof</td>
<td>156</td>
</tr>
<tr>
<td>2.4.3</td>
<td>Implementation</td>
<td>157</td>
</tr>
<tr>
<td>2.4.4</td>
<td>Synthesis of Executable Code</td>
<td>158</td>
</tr>
<tr>
<td>2.4.5</td>
<td>Feedback Arc Set with Initialization</td>
<td>160</td>
</tr>
<tr>
<td>2.4.6</td>
<td>Conclusion</td>
<td>162</td>
</tr>
<tr>
<td>2.5</td>
<td>Nested DFS</td>
<td>163</td>
</tr>
<tr>
<td>2.5.1</td>
<td>Auxiliary Lemmas</td>
<td>163</td>
</tr>
<tr>
<td>2.5.2</td>
<td>Instantiation of the Framework</td>
<td>163</td>
</tr>
<tr>
<td>2.5.3</td>
<td>Correctness Proof</td>
<td>166</td>
</tr>
<tr>
<td>2.5.4</td>
<td>Interface</td>
<td>177</td>
</tr>
<tr>
<td>2.5.5</td>
<td>Implementation</td>
<td>178</td>
</tr>
<tr>
<td>2.5.6</td>
<td>Synthesis of Executable Code</td>
<td>181</td>
</tr>
<tr>
<td>2.5.7</td>
<td>Conclusion</td>
<td>182</td>
</tr>
<tr>
<td>2.6</td>
<td>Invariants for Tarjan's Algorithm</td>
<td>183</td>
</tr>
<tr>
<td>2.7</td>
<td>Tarjan's Algorithm</td>
<td>193</td>
</tr>
<tr>
<td>2.7.1</td>
<td>Preliminaries</td>
<td>194</td>
</tr>
<tr>
<td>2.7.2</td>
<td>Instantiation of the DFS-Framework</td>
<td>194</td>
</tr>
<tr>
<td>2.7.3</td>
<td>Correctness Proof</td>
<td>197</td>
</tr>
<tr>
<td>2.7.4</td>
<td>Interface</td>
<td>223</td>
</tr>
</tbody>
</table>
Chapter 1

The DFS Framework

This chapter contains the basic DFS Framework

1.1 General DFS with Hooks

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

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

1.1.1 State and Parameterization

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

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

abbreviation NOOP s ≡ RETURN (state.more s)

Record holding the parameterization.

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

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

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

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

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

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

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

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

1.1.2 DFS operations

Node predicates

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

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

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

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

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

**definition** discover
:: 'v ⇒ 'v ⇒ ('v,'es) state-scheme ⇒ ('v,'es) state-scheme
**where**
discover u v s ⇔ let
d = (discovered s)(v ↦ counter s); c = counter s + 1;
st = v#stack s;
p = pending s ∪ {v} × E''{v};
t = insert (u,v) (tree-edges s)
in s[] discovered := d, counter := c, stack := st, pending := p, tree-edges := t]

**lemma** discover-simps[simp]:
counter (discover u v s) = Suc (counter s)
discovered (discover u v s) = (discovered s)(v ↦ counter s)
finished (discover u v s) = finished s
stack (discover u v s) = v#stack s
pending (discover u v s) = pending s ∪ {v} × E''{v}
tree-edges (discover u v s) = insert (u,v) (tree-edges s)
cross-edges (discover u v s) = cross-edges s
back-edges (discover u v s) = back-edges s
state.more (discover u v s) = state.more s
by (simp-all add: discover-def)

**definition** finish
:: 'v ⇒ ('v,'es) state-scheme ⇒ ('v,'es) state-scheme
**where**
finish u s ⇔ let
f = (finished s)(u ↦ counter s); c = counter s + 1;
st = tl (stack s)
in s[] finished := f, counter := c, stack := st]

**lemma** finish-simps[simp]:
counter (finish u s) = Suc (counter s)
discovered (finish u s) = discovered s
finished (finish u s) = (finished s)(u ↦ counter s)
stack (finish u s) = tl (stack s)
pending (finish u s) = pending s
tree-edges (finish u s) = tree-edges s
cross-edges (finish u s) = cross-edges s
back-edges (finish u s) = back-edges s
state.more (finish u s) = state.more s
by (simp-all add: finish-def)

**definition** back-edge
:: 'v ⇒ 'v ⇒ ('v,'es) state-scheme ⇒ ('v,'es) state-scheme
**where**
back-edge u v s ⇔ let
\( b = \text{insert} \ (u,v) \ (\text{back-edges} \ s) \)

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

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

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

**definition** \( \text{cross-edge} \)
\[
\begin{align*}
&: (u \Rightarrow (v,es)) \text{-state-scheme} \Rightarrow (u,es) \text{-state-scheme} \\
\text{where}
\end{align*}
\]
\[
\begin{align*}
\text{cross-edge} \ u \ v \ s \equiv \text{let} \\
&c = \text{insert} \ (u,v) \ (\text{cross-edges} \ s) \\
\text{in} \ s \{/ \ \text{cross-edges} := c \} \\
\end{align*}
\]

**lemma** \( \text{cross-edge-simps[simp]} \):
\[
\begin{align*}
\text{counter} \ (\text{cross-edge} \ u \ v \ s) &= \text{counter} \ s \\
\text{discovered} \ (\text{cross-edge} \ u \ v \ s) &= \text{discovered} \ s \\
\text{finished} \ (\text{cross-edge} \ u \ v \ s) &= \text{finished} \ s \\
\text{stack} \ (\text{cross-edge} \ u \ v \ s) &= \text{stack} \ s \\
\text{pending} \ (\text{cross-edge} \ u \ v \ s) &= \text{pending} \ s \\
\text{tree-edges} \ (\text{cross-edge} \ u \ v \ s) &= \text{tree-edges} \ s \\
\text{cross-edges} \ (\text{cross-edge} \ u \ v \ s) &= \text{insert} \ (u,v) \ (\text{cross-edges} \ s) \\
\text{back-edges} \ (\text{cross-edge} \ u \ v \ s) &= \text{back-edges} \ s \\
\text{state}.\text{more} \ (\text{cross-edge} \ u \ v \ s) &= \text{state}.\text{more} \ s \\
\text{by} \ (\text{simp-all add: cross-edge-def})
\end{align*}
\]

**definition** \( \text{new-root} \)
\[
\begin{align*}
&: (v \Rightarrow (v,es)) \text{-state-scheme} \Rightarrow (v,es) \text{-state-scheme} \\
\text{where}
\end{align*}
\]
\[
\begin{align*}
\text{new-root} \ v0 \ s \equiv \text{let} \\
&c = \text{Suc} \ (\text{counter} \ s); \\
&d = (\text{discovered} \ s)(v0 \mapsto \text{counter} \ s); \\
&p = \{v0\}\times E'' \{v0\}; \\
&st = [v0] \\
\text{in} \ s \{\text{counter} := c, \text{discovered} := d, \text{pending} := p, \text{stack} := st\} \\
\end{align*}
\]

**lemma** \( \text{new-root-simps[simp]} \):
\[
\begin{align*}
\text{counter} \ (\text{new-root} \ v0 \ s) &= \text{Suc} \ (\text{counter} \ s) \\
\text{discovered} \ (\text{new-root} \ v0 \ s) &= (\text{discovered} \ s)(v0 \mapsto \text{counter} \ s) \\
\text{finished} \ (\text{new-root} \ v0 \ s) &= \text{finished} \ s
\end{align*}
\]

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

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

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

Effects on Whole State

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

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

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

definition do-known-edge
:: 'v ⇒ 'v ⇒ ('v,es) state-scheme ⇒ ('v,es) state-scheme nres
where
do-known-edge u v s ≡
  if is-finished v s then
    do-cross-edge u v s
  else
do-back-edge u v s

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

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

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

definition do-new-root where
do-new-root v₀ s ≡
do{
  let s = new-root v₀ s;
e ← on-new-root param v₀ s;
RETURN (s[state.more := e])
}

lemmas op-defs = discover-def finish-def back-edge-def cross-edge-def new-root-def
lemmas do-defs = do-discover-def do-finish-def do-known-edge-def
do-cross-edge-def do-back-edge-def do-new-root-def
lemmas pred-defs = is-discovered-def is-finished-def is-empty-stack-def

definition init ≡
do{
e ← on-init param;
RETURN (empty-state e)
}
\end{verbatim}
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
cond s ←→ (V₀ ⊆ {v. is-discovered v s} → ¬is-empty-stack s)
∧ ¬is-break param s

**lemma** cond-alt:
cond = (λs. (V₀ ⊆ dom (discovered s) → stack s ≠ []) ∧ ¬is-break param s)
apply (rule ext)
unfolding cond-def is-discovered-def is-empty-stack-def
by auto

**definition** get-pending ::
('v,'es) state-scheme ⇒ ('v × option × ('v,'es) state-scheme) 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
step s ≡
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
        ...
\begin{verbatim}
do-known-edge u v s
else
do-discover u v s
\end{verbatim}

\textbf{definition} \( \text{it-dfs} \equiv \text{init} \gg \text{WHILE} \ \text{cond} \ \text{step} \)
\textbf{definition} \( \text{it-dfsT} \equiv \text{init} \gg \text{WHILET} \ \text{cond} \ \text{step} \)

\textbf{end}

1.1.4 Invariants

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

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

1. First, we prove the invariant wrt. the parameterization in the \textit{param-DFS} locale.

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

\textbf{locale} \textit{param-DFS} =
\begin{verbatim}
\textit{fb-graph} G + \textit{param-DFS-defs} G \textit{param}
\textbf{for} G :: (′v, ′more) \textit{graph-rec-scheme}
\textbf{and} \textit{param} :: (′v,′es) \textit{parameterization}
\textbf{begin}
\textbf{definition} is-invar :: ((′v, ′es) \textit{state-scheme} ⇒ bool) ⇒ bool
— Predicate that states that \( I \) is an invariant.
\textbf{where} is-invar \( I \) ≡ is-rwof-invar \textit{init} \textit{cond} \textit{step} \( I \)
\textbf{end}
\end{verbatim}

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

\textbf{locale} \textit{DFS-invar} =
\begin{verbatim}
\textit{param-DFS} G \textit{param}
\textbf{for} G :: (′v, ′more) \textit{graph-rec-scheme}
\textbf{and} \textit{param} :: (′v,′es) \textit{parameterization}
+ \textbf{fixes} s :: (′v,′es) \textit{state-scheme}
\textbf{assumes} rwof: \textit{rwof} \textit{init} \textit{cond} \textit{step} \( s \)
\textbf{begin}
\textbf{lemma} make-invar-thm: is-invar \( I \) ⇒ \( I \ s \)
\end{verbatim}

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

end

Establishing Invariants

context param-DFS

begin

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

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

lemma \texttt{rwof-eq-DFS-invar[simp]}:
\begin{align*}
\text{rwof init cond step } = \text{DFS-invar G param}
\end{align*}

— The DFS-invar locale is equivalent to the strongest invariant of the loop.

apply (auto intro: DFS-invar.rwof intro: ext)

by unfold-locales

lemma \texttt{DFS-invar-step['] [\text{nofail it-dfs}; DFS-invar G param s; cond s]}
\begin{align*}
& \Rightarrow \text{step s } \leq \text{SPEC (DFS-invar G param)}
\end{align*}

— A step preserves the (best) invariant.

unfolding it-dfs-def rwof-eq-DFS-invar[symmetric]
by (rule rwof-step)

lemma \texttt{DFS-invar-step['] [\text{nofail (step s)}; DFS-invar G param s; cond s]}
\begin{align*}
& \Rightarrow \text{step s } \leq \text{SPEC (DFS-invar G param)}
\end{align*}

unfolding it-dfs-def rwof-eq-DFS-invar[symmetric]
by (rule rwof-step')

We define symbolic names for the preconditions of certain operations

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

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

definition \texttt{pre-on-finish u s'} \equiv \exists s.
\begin{align*}
& \text{DFS-invar G param s } \land \text{cond s } \land \\
& \text{stack s } \neq [] \land u = \text{hd (stack s)} \land \text{pending s } \{ u \} = \{ \} \land s' = \text{finish u s}
\end{align*}
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 (\( \lambda x. \ I \ (\text{empty-state } x) \))

**assumes** new-root: \( \wedge s \ s' v 0 \).
\[
\begin{align*}
\text{DFS-invar } G \ param \ s & \land \ cond \ s \land \\
\text{stack } s & \neq [] \land u = \text{hd} \ (\text{stack } s) \land (u, v) \in \text{pending } s
\end{align*}
\]

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

**assumes** new-root: \( \wedge s \ s' v 0 \).
\[
\begin{align*}
\text{DFS-invar } G \ param \ s & \land \ cond \ s \land \neg \text{is-break } param \ s; \\
\text{stack } s & = []; v 0 \in V 0 ; v 0 \notin \text{dom} \ (\text{discovered } s); \\
s' & = \text{new-root } v 0 \ s
\end{align*}
\]

**implies** on-new-root param v 0 s' \( \leq_n \)
\[
\begin{align*}
\text{SPEC (} \lambda x. \ \text{DFS-invar } G \ param \ (s'(\text{state.more } := x)) & \\
\to I (s'(\text{state.more } := x))
\end{align*}
\]

**assumes** finish: \( \wedge s \ s' u \).
\[
\begin{align*}
\text{DFS-invar } G \ param \ s & \land \ cond \ s \land \neg \text{is-break } param \ s; \\
\text{stack } s & \neq []; u = \text{hd} \ (\text{stack } s); \\
\text{pending } s & \quad \{u\} = \{\}; \\
s' & = \text{finish } u \ s
\end{align*}
\]

**implies** on-finish param u s' \( \leq_n \)
\[
\begin{align*}
\text{SPEC (} \lambda x. \ \text{DFS-invar } G \ param \ (s'(\text{state.more } := x)) & \\
\to I (s'(\text{state.more } := x))
\end{align*}
\]

**assumes** cross-edge: \( \wedge s \ s' u v \).
\[
\begin{align*}
\text{DFS-invar } G \ param \ s & \land \ cond \ s \land \neg \text{is-break } param \ s; \\
\text{stack } s & \neq []; (u, v) \in \text{pending } s; u = \text{hd} \ (\text{stack } s); \\
v & \in \text{dom} \ (\text{discovered } s); v \notin \text{dom} \ (\text{finished } s); \\
s' & = \text{cross-edge } u v \ (s'(\text{pending } := \text{pending } s - \{(u, v)\}))
\end{align*}
\]

**implies** on-cross-edge param u v s' \( \leq_n \)
\[
\begin{align*}
\text{SPEC (} \lambda x. \ \text{DFS-invar } G \ param \ (s'(\text{state.more } := x)) & \\
\to I (s'(\text{state.more } := x))
\end{align*}
\]

12
assumes back-edge: $\forall s \ s' \ u \ v$.

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

$\text{stack} \ s \neq []; \ (u, v) \in \text{pending} \ s; \ u = \text{hd} \ (\text{stack} \ s)$

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

$s' = \text{back-edge} \ u \ v \ (s[\text{pending} := \text{pending} - \{(u,v)\}])$

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

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

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

assumes discover: $\forall s \ s' \ u \ v$.

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

$\text{stack} \ s \neq []; \ (u, v) \in \text{pending} \ s; \ u = \text{hd} \ (\text{stack} \ s)$

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

$s' = \text{discover} \ u \ v \ (s[\text{pending} := \text{pending} - \{(u,v)\}])$

$\implies \text{on-discover} \ param \ u \ v \ s' \leq n$

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

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

shows is-invar I

unfolding is-invar-def

proof

show init $\leq_n \text{SPEC} \ I$

unfolding init-def

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

next

fix $s$

assume rwof init cond step s and IC: $I \ s \ cond \ s$

hence $DI$: $\text{DFS-invar} \ G \ param \ s$ by unfold-locales

then interpret $\text{DFS-invar} \ G \ param \ s$

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

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

by rule (metis DFS-invar-step DI $\langle \text{cond} \ s \rangle$)

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

— Establish a DFS invariant (symbolic preconditions).

assumes init: on-init param ≤ₙ SPEC (λx. I (empty-state x))
assumes new-root: \( s \vdash v \exists \phi. \text{pre-on-new-root} v \theta s' \)

\[ \implies \text{on-new-root} param v \theta s' \leqₙ \]

SPEC (λx. DFS-invar G param (s'⟨ state.more := x ⟩))

\[ \implies I (s'⟨ state.more := x ⟩) \]

assumes finish: \( s \vdash u \). pre-on-finish u s'

\[ \implies \text{on-finish} param u s' \leqₙ \]

SPEC (λx. DFS-invar G param (s'⟨ state.more := x ⟩))

\[ \implies I (s'⟨ state.more := x ⟩) \]

assumes cross-edge: \( s \vdash u v \). pre-on-cross-edge u v s'

\[ \implies \text{on-cross-edge} param u v s' \leqₙ \]

SPEC (λx. DFS-invar G param (s'⟨ state.more := x ⟩))

\[ \implies I (s'⟨ state.more := x ⟩) \]

assumes back-edge: \( s \vdash u v \). pre-on-back-edge u v s'

\[ \implies \text{on-back-edge} param u v s' \leqₙ \]

SPEC (λx. DFS-invar G param (s'⟨ state.more := x ⟩))

\[ \implies I (s'⟨ state.more := x ⟩) \]

assumes discover: \( s \vdash u v \). pre-on-discover u v s'

\[ \implies \text{on-discover} param u v s' \leqₙ \]

SPEC (λx. DFS-invar G param (s'⟨ state.more := x ⟩))

\[ \implies I (s'⟨ state.more := x ⟩) \]

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

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

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

assumes prereq: \( s \vdash u v s \). on-discover param u v s = on-new-root param v s
assumes init: on-init param ≤ₙ SPEC (λx. I (empty-state x))
assumes new-discover: \( s \vdash s' v \)

\[ \text{DFS-invar} G \ param s; I s; \ \text{cond} \ s; \ \neg \ \text{is-break} \ param \ s; \ v \notin \ \text{dom} \ (\text{discovered} \ s); \]

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

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

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

\[ \text{tree-edges} \ s' \supseteq \text{tree-edges} \ s; \]

\[ \text{state.more} \ s' = \text{state.more} \ s \]

\[ \implies \text{on-new-root} param v s' \leqₙ \]

SPEC (λx. DFS-invar G param (s'⟨ state.more := x ⟩))

\[ \implies I (s'⟨ state.more := x ⟩) \]
assumes finish: \( \forall s s' u. \)
\[
\langle \text{DFS-invar } G \text{ param } s; I \text{ s; cond } s; \neg \text{-break param } s; \\
\text{stack } s \neq []; u = \text{hd} (\text{stack } s); \\
\text{pending } s' \{u\} = \{\}; \\
s' = \neg \text{finish } u s \rangle 
\]
\( \Longrightarrow \text{on-finish } \text{param } u s' \leq_n 
\)
SPEC (\( \lambda x. \text{DFS-invar } G \text{ param } (s'(\text{state.more } := x)) \)
\( \rightarrow I (s'(\text{state.more } := x)) \)
assumes cross-edge: \( \forall s s' u v. \)
\[
\langle \text{DFS-invar } G \text{ param } s; I \text{ s; cond } s; \neg \text{-break param } s; \\
\text{stack } s \neq []; (u, v) \in \text{pending } s; u = \text{hd} (\text{stack } s); \\
v \in \text{dom } (\text{discovered } s); v \in \text{dom } (\text{finished } s); \\
s' = \text{cross-edge } u v (s'(\text{pending } := \text{pending } s - \{(u,v)\})]) 
\]
\( \Longrightarrow \text{on-cross-edge } \text{param } u v s' \leq_n 
\)
SPEC (\( \lambda x. \text{DFS-invar } G \text{ param } (s'(\text{state.more } := x)) \)
\( \rightarrow I (s'(\text{state.more } := x)) \)
assumes back-edge: \( \forall s s' u v. \)
\[
\langle \text{DFS-invar } G \text{ param } s; I \text{ s; cond } s; \neg \text{-break param } s; \\
\text{stack } s \neq []; (u, v) \in \text{pending } s; u = \text{hd} (\text{stack } s); \\
v \in \text{dom } (\text{discovered } s); v \notin \text{dom } (\text{finished } s); \\
s' = \text{back-edge } u v (s'(\text{pending } := \text{pending } s - \{(u,v)\})]) 
\]
\( \Longrightarrow \text{on-back-edge } \text{param } u v s' \leq_n 
\)
SPEC (\( \lambda x. \text{DFS-invar } G \text{ param } (s'(\text{state.more } := x)) \)
\( \rightarrow I (s'(\text{state.more } := x)) \)
shows is-invar I
proof (induct rule: establish-invarI)
case (new-root s) thus \( \text{?case by (auto intro; new-discover)} \)
next
case (discover s s' u v) hence
on-new-root param v s' \leq_n 
SPEC (\( \lambda x. \text{DFS-invar } G \text{ param } (s'(\text{state.more } := x)) \)
\( \rightarrow I (s'(\text{state.more } := x)) \)
by (auto intro; new-discover)
with prereq show \( \text{?case by simp} \)
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: on-init param \( \leq_n \) SPEC (\( \lambda x. I (\text{empty-state } x) \))
assumes new-root: \( \forall s s' v0. \)
\[
\langle \text{DFS-invar } G \text{ param } s; I \text{ s; cond } s; \neg \text{-break param } s; \\
\text{stack } s = []; v0 \in V0; v0 \notin \text{dom } (\text{discovered } s); \\
s' = \text{new-root } v0 s \rangle 
\]
\( \Longrightarrow \text{on-new-root } \text{param } v0 s' \leq_n 
\)
SPEC (\( \lambda x. \text{DFS-invar } G \text{ param } (s'(\text{state.more } := x)) \)
\( \rightarrow I (s'(\text{state.more } := x)) \))
assumes finish: \( \forall s \, s' \, u. \)

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

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

pending \( s' = \{ u \} \);

\( s' = \text{finish} \ u \ s' \)

\( \implies \text{on-finish param} \ u \ s' \leq n \)

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

\( \implies I \ (s'[\text{state}.\text{more} := x])) \)

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

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

stack \( s \neq [] \): \( (u, v) \in \text{pending} \ s; \ u = \text{hd} \ (\text{stack} \ s) \);

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

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

stack \( s' = \text{stack} \ s; \ \text{tree-edges} \ s' = \text{tree-edges} \ s; \ \text{counter} \ s' = \text{counter} \ s; \)

pending \( s' = \text{pending} \ s - \{(u,v)\} \);

cross-edges \( s' \cup \text{back-edges} \ s' = \text{cross-edges} \ s \cup \text{back-edges} \ s \cup \{(u,v)\} \);

state.\text{more} \( s' = \text{state.\text{more}} \ s \)

\( \implies \text{on-cross-edge param} \ u \ v \ s' \leq n \)

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

\( \implies I \ (s'[\text{state}.\text{more} := x])) \)

assumes discover: \( \forall s \, s' \, u \, v. \)

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

stack \( s \neq [] \): \( (u, v) \in \text{pending} \ s; \ u = \text{hd} \ (\text{stack} \ s) \);

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

discovered \( s' = \text{discover} \ u \ v \ (s[\text{pending} := \text{pending} \ s - \{(u,v)\}]) \]

\( \implies \text{on-discover param} \ u \ v \ s' \leq n \)

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

\( \implies I \ (s'[\text{state}.\text{more} := x])) \)

shows is-invar I

proof (induct rule: establish-invarI)

case cross-edge thus \( \exists \text{case by} \ (\text{auto intro!}: \text{cross-back-edge}) \)

next

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

on-cross-edge param u v s' \leq n

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

\( \implies I \ (s'[\text{state}.\text{more} := x])) \)

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

with \( \text{prereq show \ \exists \text{case by simp}} \)

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:

\( \forall u \, v \, s. \ \text{on-discover param} \ u \ v \ s = \text{on-new-root param} \ v \ s \)

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

assumes init: \( \text{on-init param} \leq n \)

\( \text{SPEC} \ (\forall x. \ I \ (\text{empty-state} \ x)) \)

16
assumes new-discover: $\forall s \; s' \; v$.

$[\text{DFS-invar } G \; \text{param } s; \; I \; s; \; \text{cond } s; \; \neg \; \text{is-break } \text{param } s; \; 
v \notin \text{dom } (\text{discovered } s); \; $

$\text{discovered } s' = (\text{discovered } s)(u \mapsto \text{counter } s); \; \text{finished } s' = \text{finished } s; \; 
\text{counter } s' = \text{Suc } (\text{counter } s); \; \text{stack } s' = v \# \text{stack } s; \; 
\text{back-edges } s' = \text{back-edges } s; \; \text{cross-edges } s' = \text{cross-edges } s; \; 
\text{tree-edges } s' \supseteq \text{tree-edges } s; \; 
\text{state } \text{more } s' = \text{state } \text{more } s]$

$\implies \text{on-new-root } \text{param } v \; s' \leq_n \; 
\text{SPEC } (\lambda x. \; \text{DFS-invar } G \; \text{param } (s'(\text{state } \text{more } := x)))$ \n
$\implies \text{on-new-root } \text{param } v \; s' \leq_n \; 
\text{SPEC } (\lambda x. \; \text{DFS-invar } G \; \text{param } (s'(\text{state } \text{more } := x)))$

assumes finish: $\forall s \; s' \; u$.

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

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

assumes cross-back-edge: $\forall s \; s' \; u \; v$.

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

shows is-invar I

proof (induct rule: establish-invar-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' \leq_n

$\text{SPEC } (\lambda x. \; \text{DFS-invar } G \; \text{param } (s'(\text{state } \text{more } := x)))$ \n
$\implies \text{on-cross-edge } \text{param } u \; v \; s' \leq_n \; 
\text{SPEC } (\lambda x. \; \text{DFS-invar } G \; \text{param } (s'(\text{state } \text{more } := x)))$

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

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: $\forall e. \; I \; (\text{empty-state } e)$

assumes new-root: $\forall s \; s' \; v0 \; e$. 

17
\[ I \text{ s; cond s; DFS-invar G param s; DFS-invar G param s'};\]
\[ \text{stack s} = []; \ \text{v0} \notin \text{dom (discovered s)}; \ \text{v0} \in \text{V0};\]
\[ s' = \text{new-root v0 s} \{ \text{state.more := e} \}\]
\[ \implies I \ s' \]

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

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

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

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

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

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

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

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

\textbf{and visited}: \(\forall s \ s' \ u \ v \ e \ c \ b.\)
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 ∉ dom (finished s)
using stack-set-def
by auto

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

lemma finished-discovered:
dom (finished s) ⊆ dom (discovered s)
using basic-invar
by (auto simp add: basic-invar-def)

lemma finished-no-pending:
v ∈ dom (finished s) ⇒ pending s "{v} = {}
using basic-invar
by (auto simp add: basic-invar-def)

lemma discovered-eq-finished-un-stack:
dom (discovered s) = dom (finished s) ∪ set (stack s)
using stack-set-def finished-discovered by auto

lemma pending-on-stack:
(v,w) ∈ pending s ⇒ v ∈ set (stack s)
using basic-invar
by (auto simp add: basic-invar-def)

lemma empty-stack-imp-empty-pending:
stack s = [] ⇒ pending s = {}
using pending-on-stack
by auto
end

context param-DFS begin

lemma i-discovered-reachable:
is-invar (λs. dom (discovered s) ⊆ reachable)
proof (induct rule: is-invarI)
case (discover s) then interpret i: DFS-invar where s=s by simp
from discover show ?case
apply (clarsimp dest: i.pendingD)
by (metis contra-subsetD list.set.sel(1) rtrancl-image-advance i.stack-discovered)
qed auto

definition discovered-closed s ≡
E"dom (finished s) ⊆ dom (discovered s)
∧ (E − pending s) "set (stack s) ⊆ dom (discovered s)

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

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

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

end

context \texttt{DFS-invar}
begin

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

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

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

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

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

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

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

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

lemma pending-finite[simp, intro!]: \text{finite (pending s)}
proof
have \text{pending s} \(\subseteq\) \((\text{SIGMA u:dom (discovered s)}. E''\{u\})\)
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 ∈ dom (discovered s)  
and pending s "\{u\} = {}"  
and v ∈ succ u
shows v ∈ dom (discovered s)
proof (cases u ∈ 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 ∈ 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: ¬cond s ¬is-break param s
shows dom (finished s) = reachable
proof
dom (discovered s) = dom (finished s)  
and SS: V0 ⊆ dom (discovered s)
unfolding basic-invar-def cond-alt by auto

show dom (finished s) = reachable
proof
from discovered-reachable show dom (finished s) ⊆ reachable
by simp
next
from discovered-closed have E"{dom (finished s)} ⊆ dom (finished s)
unfolding discovered-closed-def by auto
with SS show reachable ⊆ dom (finished s)
  by (simp, metis rtrancl-reachable-induct)
qed

lemma nc-V0-finished:
assumes NC: ¬cond s ¬is-break param s
shows V0 ⊆ 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
  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

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

  show thesis
    unfolding it-dfsT-def it-dfs-def
    apply (rule bind-mono)
apply simp
apply (subst WHLET-eq-WHILE-tproof[where I=DFS-invar G param
and V=param-dfs-variant])
apply auto []
apply (subst rwof-eq-DFS-invar[symmetric])
using rwof-init[OF True[unfolded it-dfs-def]]
apply (fastforce dest: order-trans) []
apply (rule SPEC-rule-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 ⇒ nofail (on-discover param u v s)
begin

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

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

lemma it-dfs-eq-spec: it-dfs = SPEC (λs. DFS-invar G param s ∧ ¬cond s)
unfolding rwof-eq-DFS-invar[symmetric] it-dfs-def
apply (rule nofail-WHILE-eq-rwof)
apply (subst WHILE-eq-I-rwof)
unfolding rwof-eq-DFS-invar
apply (rule SPEC-nofail[where Φ=λ-. True])
apply (refine-vcg leafD[OF - init-leaf-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
getpending-def getnew-root-def pred-defs
split: option.split)
apply (intro allI conjI impI nofails)
apply (auto simp add: pre-on-defs)
done

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

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

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

lemma it-dfsT-SPEC:
  assumes D : finite reachable
  assumes D : ∀s. \[∀\ s . [DFS-invar G param s ; ¬cond s] ⟹ P s\]
  shows it-dfsT ≤ SPEC P
apply (subst it-dfsT-eq-it-dfs[OF assms(1)])
using assms(2)
by (rule it-dfs-SPEC)

27
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). v ≠ w → δ s v ≠ δ s w)
∧ (∀ v ∈ 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 \(*\): \(hd\ (stack\ s) \not\in\ dom\ (finished\ s)\)
using stack-not-finished stack-discovered hd-in-set\(\{OF\ \text{NE}\}\)
by blast+

from discovered-closed have
\((E - pending\ s) \cap \{hd\ (stack\ s)\} \subseteq dom\ (discovered\ s)\)
using hd-in-set\(\{OF\ \text{NE}\}\)
by (auto simp add: discovered-closed-def)

hence succ-hd: pending\ s \cap \{hd\ (stack\ s)\} = \{}
\implies succ\ (hd\ (stack\ s)) \subseteq dom\ (discovered\ s)
by blast

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

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

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

qed (simp-all add: timing-common-inv-def)
lemmas s-timing-common-inv =
  timing-common-inv[THEN make-invar-thm]

lemma timing-less-counter:
  \( v \in \text{dom} (\text{discovered} s) \Rightarrow \delta s v < \text{counter} s \)
  \( v \in \text{dom} (\text{finished} s) \Rightarrow \phi s v < \text{counter} s \)
  by (auto simp add: timing-common-inv-def)

lemma disc-lt-fin:
  \( v \in \text{dom} (\text{finished} s) \Rightarrow \delta s v < \phi s v \)
  by (auto simp add: timing-common-inv-def)

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

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

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

end end

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

lemma i-prev-stack-discover-all:
  \( \text{is-invar } (\lambda s. \forall n < \text{length} (\text{stack} s). \forall v \in \text{set} (\text{drop} (\text{Suc} n) (\text{stack} s)). \delta s (\text{stack} s ! n) > \delta s v) \)
  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-nil: v ∉ set (stack s) by auto

from stack-discovered timing-less-counter have
  \∀ w. w ∈ set (stack s) ⇒ δ s w < counter s
  by blast
with v-nil have \∀ w. w ∈ set (stack s) ⇒ δ s' w < δ s' v by auto
hence \∀ w. w ∈ set (drop (Suc 0) (stack s')) ⇒ δ s' w < δ s' (stack s' ! 0)
  by auto

moreover
from v-nil have
  \∀ n. [n < (length (stack s')) ; n > 0] \implies \delta s' (stack s' ! n) = \delta s (stack s' ! n)
  by auto
with discover(1) v-nil have \∀ n. [n < (length (stack s')) - 1 ; n > 0] \implies \forall w ∈ set (drop (Suc n) (stack s')). \delta s' (stack s' ! n) > \delta s' w
  by (auto dest: in-set-dropD)
ultimately show ?case
  by (metis drop-Suc-Cons length-drop length-pos-if-in-set length-tl
      list.sel(3) neq0-conv nth-Cons-0 EQ(2) zero-less-diff)
qed simp-all

end end

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

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

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

lemma Suc-stack-discover:
  assumes n: n < (length (stack s)) - 1
  shows \delta s (stack s ! n) > \delta s (stack s ! Suc n)
proof =
  from prev-stack-discover assumes have
    \∀ v. v ∈ set (drop (Suc n) (stack s)) ⇒ \delta s (stack s ! n) > \delta s v 
    by fastforce
moreover from n have stack s ! Suc n ∈ set (drop (Suc n) (stack s))
  using in-set-conv-nth by fastforce
ultimately show \( \exists \text{thesis} \).

\[\text{qed}\]

**Lemma** \( tl-lt-stack-hd-discover \):

**Assumes**
- \( \text{notempty: stack } s \neq [] \)
- \( x \in \text{set (tl (stack } s)) \)
- \( \delta s x < \delta s (hd (stack s)) \)

**Proof**
- from \( \text{notempty obtain y ys where stack } s = y\#ys by (metis list.exhaust) \)
  with \( \text{assms show } \exists \text{thesis} \)
  using \( \text{prev-stack-discover} \)
  by \( \text{(cases } ys) \) force+

\[\text{qed}\]

**Lemma** \( stack-nth-order \):

**Assumes**
- \( l: i < \text{length (stack } s) \) \( j < \text{length (stack } s) \)

**Shows**\( \delta s (stack s ! i) < \delta s (stack s ! j) \iff i > j \) (Is \( \text{\delta s } \forall i < \delta s \exists j \iff \) -)

**Proof**
- assume \( \delta: \delta s \forall i < \delta s \exists j \)

from \( l \) stack-set-def have\( \text{disc: } \forall i \in \text{dom (discovered } s) \) \( \forall j \in \text{dom (discovered } s) \)

by \( \text{auto} \)
with \( \text{disc-unequal[OF disc]} \) \( \delta \) have \( i \neq j \) by \( \text{auto} \)

moreover

\[
\{ \\
\hspace{1cm} \text{assume } i < j \\
\hspace{1cm} \text{with } l \text{ have stack } s ! j \in \text{set (drop (Suc } i) \text{ (stack } s)) \\
\hspace{1cm} \text{using in-set-drop-cone-nth[of stack } s ! j \text{ Suc } i \text{ stack } s] \\
\hspace{1cm} \text{by fastforce} \\
\hspace{1cm} \text{with prev-stack-discover } l \text{ have } \delta s (stack s ! j) < \delta s (stack s ! i) \\
\hspace{1cm} \text{by simp} \\
\hspace{1cm} \text{with } \delta \text{ have } False \text{ by simp} \\
\}
\]

ultimately show \( i > j \) by \( \text{force} \)

next

\[
\{ \\
\hspace{1cm} \text{assume } i > j \\
\hspace{1cm} \text{with } l \text{ have stack } s ! i \in \text{set (drop (Suc } j) \text{ (stack } s)) \\
\hspace{1cm} \text{using in-set-drop-cone-nth[of stack } s ! i \text{ Suc } j \text{ stack } s] \\
\hspace{1cm} \text{by fastforce} \\
\hspace{1cm} \text{with prev-stack-discover } l \text{ show } \delta s \forall i < \delta s \exists j \text{ by simp} \\
\}
\]

\[\text{qed}\]

end end

1.2.2 Paranthesis Theorem

context param-DFS begin context begin interpretation timing-syntax.
\textbf{definition} parenthesis \( s \equiv \)
\[
\forall v \in \text{dom (discovered } s) \land \forall w \in \text{dom (discovered } s). \\
\delta s v < \delta s w \land v \in \text{dom (finished } s) \rightarrow ( \\
\not\exists s w < \delta s w \land \text{disjoint} \\
\lor (\not\exists s s v > \delta s w \land w \in \text{dom (finished } s) \land \not\exists s s w < \varphi s v))
\]

\textbf{lemma} \( i\text{-parenthesis: is-invar parenthesis} \)
\textbf{proof} (induct rule: \textit{is-invar}I)
\begin{itemize}
\item case \( \text{finish } s s' \)
\end{itemize}
\begin{itemize}
\item hence EQ[simp]: discovered \( s' = \text{discovered } s \)
\end{itemize}
\begin{itemize}
\item counter \( s' = \text{Suc } (\text{counter } s) \)
\end{itemize}
\begin{itemize}
\item finished \( s' = \text{finished } s(\text{hd } (\text{stack } s) \rightarrow \text{counter } s) \)
\end{itemize}
\begin{itemize}
\item by simp-all
\end{itemize}

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

\{
\begin{itemize}
\item fix \( x y \)
\item assume \( \text{dom: } x \in \text{dom (discovered } s') \land y \in \text{dom (discovered } s') \)
\item and \( \delta s' x < \delta s' y \)
\item and \( f: x \in \text{dom (finished } s') \)
\item hence neq: \( x \neq y \text{ by force} \)
\end{itemize}
\begin{itemize}
\item note \( \text{assms = dom } \delta f \text{ EQ} \)
\end{itemize}

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

\textbf{have} \( ?\text{DISJ } \lor ?\text{IN} \)
\textbf{proof} (cases \( x = \text{hd } (\text{stack } s) \))
\begin{itemize}
\item case \( \text{True note } x\text{-is-hd = this} \)
\end{itemize}
\begin{itemize}
\item hence \( \varphi x: \varphi s' x = \text{counter } s \text{ by simp} \)
\item from \( x\text{-is-hd neg have } y\text{-not-hd: } y \neq \text{hd } (\text{stack } s) \text{ by simp} \)
\end{itemize}
\begin{itemize}
\item have \( \delta s y < \varphi s' x \land y \in \text{dom (finished } s) \land \varphi s y < \varphi s' x \)
\item proof (cases \( y \in \text{set } (\text{stack } s) \))
\item y on stack is not possible: According to
\item \( \delta s' x < \delta s' y \)
\end{itemize}

\end{itemize}

\begin{itemize}
\item it is discovered after \( x (= \text{hd } (\text{stack } s)) \)
\item case \( \text{True with } y\text{-not-hd have } y \in \text{set } (\text{tl } (\text{stack } s)) \)
\item by (cases \( \text{stack } s \) simp-all)
\item with \( \text{tl-tl-stack-hd-discover[OF NE]} \delta x\text{-is-hd have } \delta s y < \delta s x \)
\item by simp
\item with \( \delta \) have \( \text{False by simp} \)
\item thus \( ?\text{thesis } .. \)
\item next
\end{itemize}
case False — y must be a successor of x (= (hd (stack s)))
from dom have y ∈ dom (discovered s) by simp
with False discovered-not-stack-imp-finished have *:
y ∈ dom (finished s)
by simp
moreover with timing-less-counter φx have φ s y < φ s' x by simp
moreover with * disc-lt-fin φx have δ s y < φ s' x
by (metis less-trans)
ultimately show ?thesis by simp
qed
with y-not-hd show ?thesis by simp
next
case False note [simp] = this
show ?thesis
proof (cases y = hd (stack s))
case False with finish assms show ?thesis
  by (simp add: parenthesis-def)
next
case True with stack-not-finished have y /∈ dom (finished s)
  using hd-in-set[OF NE]
  by auto
with finish assms have φ s x < δ s y
  unfolding parenthesis-def
  by auto
hence ?DISJ by simp
thus ?thesis ..
qed
qed
}
thus ?case by (simp add: parenthesis-def)
next
case (discover s s' u v)
hence EQ[simp]: discovered s' = (discovered s)(v ↦→ counter s)
  finished s' = finished s
  counter s' = Suc (counter s)
by simp-all
from discover interpret DFS-invar where s=s by simp
from discover finished-discovered have
  V': v /∈ dom (discovered s) v /∈ dom (finished s)
  by auto

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

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

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

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

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

lemma parenthesis-disjoint:
assumes \( v \in \text{dom} (\text{finished } s) \) \( w \in \text{dom} (\text{discovered } s) \)
and \( \delta s v < \delta s w \varphi s w > \varphi s v \)
shows \( \varphi s v < \delta s w \)

35
using parenthesis assms
by force

lemma finished-succ-contained:
assumes \( v \in \text{dom} \ (\text{finished } s) \)
and \( w \in \text{succ } v \)
and \( \delta s v < \delta s w \)
shows \( w \in \text{dom} \ (\text{finished } s) \land \varphi s w < \varphi s v \)
using finished-succ-fin finished-imp-succ-discovered parenthesis-contained
using assms
by metis

end end

1.2.3 Edge Types

context param-DFS
begin

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

lemma is-invar \((\lambda s. \text{finite} \ (\text{edges } s))\)
by (induction rule: establish-invar1) auto

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

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

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

lemma i-edges-basic:
is-invar edges-basic
unfolding edges-basic-def[abs-def]
proof (induct rule: is-invar1-full)
case (back-edge \( s \))
then interpret DFS-invar where \( s=s \) by simp
from back-edge show \( \text{case } \) by (auto dest: pendingD)
next
case (cross-edge s)
then interpret DFS-invar where s=s by simp
from cross-edge show ?case by (auto dest: pendingD)
next
case (discover s)
then interpret DFS-invar where s=s by simp
from discover show ?case
  apply (simp add: Field-def Range-def Domain-def)
  apply (drule pendingD)
  apply simp
  by (blast)
next
case (new-root s)
thus ?case by (simp add: Field-def) blast
qed auto
lemmas (in DFS-invar) edges-basic = i-edges-basic[THEN make-invar-thm]

lemma i-edges-covered:
  is-invar (λs. (E ∩ dom (discovered s) × UNIV) − pending s = edges s)
proof (induction rule: is-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/∈ dom (discovered s)⟩
  have [simp]: E ∩ insert v0 (dom (discovered s)) × UNIV − {v0} × succ v0
  = E ∩ dom (discovered s) × 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 ∈ dom (discovered s)
  by (cases stack s) auto
  with cross-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 cross-edge by simp
next
case (back-edge s s’ u v)
  interpret DFS-invar G param s by fact

37
from back-edge stack-discovered have \( u \in \text{dom} \text{(discovered s)} \)
by (cases stack s) auto

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

thus ?case using back-edge by simp

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

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

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

thus ?case using discover by simp
qed simp-all

context DFS-invar begin

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

lemma edges-ss-reachable-edges:
edges s \( \subseteq E \cap \text{reachable} \times \text{UNIV} \)
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 \( \cap \) reachable \( \times \) 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
qed

38
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)

  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) ∈ edges s ⇒ w ∈ dom (discovered s)
  using tree-edge-imp-discovered cross-edge-imp-discovered back-edge-imp-discovered
by blast+

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

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

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

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

end

Properties of the DFS Tree

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

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

category 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
        tree-edges s' = insert (hd (stack s), v) (tree-edges s)
      by simp-all
    from discover have NE[simp]: stack s \not= [] by simp
    from discover obtain v0 where
      v0 \in V0
      path (tree-edges s) v0 (\text{rev (tl (stack } s))) (\text{hd (stack } s))
by blast
with path-mono[OF - this(2)] EQ have
  path (tree-edges s') v0 (rev (tl (stack s))) (hd (stack s))
by blast
with v0 ∈ V0 show ?case
by (cases stack s) (auto simp: path-simps)

next
  case (finish s s')
  hence EQ[simp]: stack s' = tl (stack s)
  tree-edges s' = tree-edges s
  by simp-all

from finish obtain v0 where
  v0 ∈ V0
  path (tree-edges s) v0 (rev (tl (stack s))) (hd (stack s))
by blast
  hence P: path (tree-edges s') v0 (rev (stack s')) (hd (stack s)) by simp

show ?case
proof
  assume A: stack s' ≠ []
  with P have (hd (stack s'), hd (stack s)) ∈ tree-edges s'
    by (auto simp: neq-Nil-conv path-simps)
  moreover from P A have
    path (tree-edges s') v0 (rev (tl (stack s'))) @ (hd (stack s')) (hd (stack s'))
    by (simp)
  moreover note (v0 ∈ V0)
  ultimately show ∃ v0 ∈ V0. path (tree-edges s') v0 (rev (tl (stack s'))) (hd (stack s'))
    by (auto simp add: path-append-conv)
  qed
  qed simp-all
end

context DFS-invar begin

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

lemma stack-is-path:
  stack s ≠ [] ⇒ ∃ v0 ∈ V0. path E v0 (rev (tl (stack s))) (hd (stack s))
using stack-is-tree-path path-mono[OF tree-edges-ssE]
by blast

lemma hd-succ-stack-is-path:
  assumes ne: stack s ≠ []
  and succ: v ∈ succ (hd (stack s))
  shows ∃ v0 ∈ V0. path E v0 (rev (stack s)) v

41
proof  
  from stack-is-path[OF ne] succ obtain v0 where  
v0 ∈ V0  
  path E v0 (rev (tl (stack s)) @ [hd (stack s)]) v  
  by (auto simp add: path-append-conv)  
thus ?thesis using ne  
  by (cases stack s) auto  
qed  

lemma tl-stack-hd-tree-path:  
  assumes stack s ≠ []  
  and v ∈ set (tl (stack s))  
  shows (v, hd (stack s)) ∈ (tree-edges s)⁺  
proof  
  from stack-is-tree-path assms obtain v0 where  
  path (tree-edges s) v0 (rev (tl (stack s))) (hd (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 ≡  
  (tree-edges s = {} −→ dom (discovered s) ⊆ V0 ∧ (stack s = [])  
  ∨ (∃ v0∈V0. stack s = [v0]))  
  ∧ (tree-edges s ≠ {} −→ (tree-edges s)⁺ " V0 ∪ V0 = dom  
  (discovered s) ∪ V0)  

lemma i-tree-discovered-inv:  
is-invar tree-discovered-inv  
proof (induct rule: is-invarI)  
case (discover s s’ u v)  
hence EQ[simp]: stack s’ = v # stack s  
  tree-edges s’ = insert (hd (stack s), v) (tree-edges s)  
  discovered s’ = (discovered s)(v ↦ counter s)  
  by simp-all  
  from discover interpret DFS-invar where s=s by simp  

from discover have NE[simp]: stack s ≠ [] by simp  
note TDI = (tree-discovered-inv s)[unfolded tree-discovered-inv-def]  
  have tree-edges s’ = {} −→ dom (discovered s’) ⊆ V0 ∧ (stack s’ = []) ∨  
  (∃ v0∈V0. stack s’ = [v0]))  
  by simp — tree-edges s’ ≠ {}  
  moreover {  
    fix x
assume \( A: x \in (\text{tree-edges } s')^+ \quad \vdash \quad V_0 \cup V_0 \not\ni V_0 \) then obtain \( y \) where \( y: (y,x) \in (\text{tree-edges } s')^+ \quad y \in V_0 \) by auto

have \( x \in \text{dom (discovered } s') \cup V_0 \)
proof (cases \( \text{tree-edges } s = \{\} \))
  case True with discover \( A \) have \( (\text{tree-edges } s')^+ = \{(\text{hd (stack } s), v)\} \)
    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 \( (y,x) \in (\text{tree-edges } s')^+ \quad V_0 \) by auto
  with \( t-ne \) TDI show \(?thesis\) by auto
qed simp
qed

\{
fix \( x \)
assume \( x \in \text{dom (discovered } s') \cup V_0 \not\ni V_0 \)
hence \( A: x \in \text{dom (discovered } s') \) by simp

have \( x \in (\text{tree-edges } s')^+ \quad \vdash \quad V_0 \cup V_0 \)
proof (cases \( \text{tree-edges } s = \{\} \))
  case True with \( \text{trancl-single} \) have \( (\text{tree-edges } s')^+ = \{(\text{hd (stack } s), v)\} \) by simp
  moreover from \( \text{True TDI} \) have \( \text{hd (stack } s) \in V_0 \quad \text{dom (discovered } s) \subseteq V_0 \) by auto
  ultimately show \(?thesis\) using \( A \) \( x \not\in V_0 \) by auto
next
  case False note \( t-ne = \) this
show \(?thesis\)
proof (cases \( x = v \))
  case False with \( A \) have \( x \in \text{dom (discovered } s) \) by simp
  with \( \text{TDI } t-ne \) \( x \not\in V_0 \) have \( x \in (\text{tree-edges } s')^+ \quad \vdash \quad V_0 \) by auto
  with \( \text{trancl-sub-insert-trancl} \) show \(?thesis\) by simp blast
\}
next
case True
from i-ne TDI have dom (discovered s) ∪ V0 = (tree-edges s)⁺ ⋂ V0 ∪

V0
  by simp

moreover from stack-is-tree-path[of NE] obtain v0 where v0 ∈ V0

and
(v0, hd (stack s)) ∈ (tree-edges s)⁺
by (blast intro!: path-is-rtrancl)

with EQ have (v0, hd (stack s)) ∈ (tree-edges s)⁺ by (auto intro:

rtrancl-mono-mp)

ultimately show ?thesis using (v0 ∈ 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 ∉ V0 ⟷ v ∈ dom (discovered s) ⟷ (∃v0∈V0. (v0,v) ∈ (tree-edges s)⁺)

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

lemma i-tree-one-predecessor:
is-invar (λs. ∀(v,v′) ∈ tree-edges s. ∀y. y ≠ v → (y,v′) ∉ tree-edges s)

proof (induct rule: is-invarI)
case (discover s s' u v)
hence EQ[simp]: tree-edges s' = insert (hd (stack s),v) (tree-edges s) by simp

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

{ fix w w' y
  assume *: (w,w') ∈ tree-edges s'
  and  y ≠ w

  from discover stack-discovered have v-hd: hd (stack s) ≠ v
  using hd-in-set[of NE] by blast
  from discover tree-edges-discovered have
  v-notin-tree: ∀(x,x') ∈ tree-edges s. x ≠ v ∧ x' ≠ v
  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 \(v\text{-notin-tree}\) show False by auto
  qed
  with True \(\ast\) \(y \neq w\) \(v\text{-hd}\) show \(?\text{thesis}\)
    apply (cases \(w = v\))
    apply simp
    using discover apply simp apply blast
    done
next
  case False
    with \(v\text{-notin-tree}\) \(\ast\) \(y \neq w\) \(v\text{-hd}\)
    show \(?\text{thesis}\)
    apply (cases \(w' = v\))
    apply simp apply blast
    using discover apply simp apply blast
    done
  qed

thus \(?\text{case}\) by blast
qed simp-all

lemma (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:
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
\} thus ?case by blast
next
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) ∉ tree-edges s
  using tree-edge-disc by auto

lemma tree-edge-unequal:
  (v,w) ∈ tree-edges s → v ≠ w
  by (metis no-self-loop-in-tree)

lemma tree-path-unequal:
  (v,w) ∈ (tree-edges s)⁺ → v ≠ w
  by (metis no-loop-in-tree)

lemma tree-subpath':
  assumes x: (x,v) ∈ (tree-edges s)⁺
  and y: (y,v) ∈ (tree-edges s)⁺
  and x ≠ y
  shows (x,y) ∈ (tree-edges s)⁺ ∨ (y,x) ∈ (tree-edges s)⁺
  proof
    from x obtain px where px: path (tree-edges s) x px v and px ≠ []
      using trancl-is-path by metis
    from y obtain py where py: path (tree-edges s) y py v and py ≠ []
      using trancl-is-path by metis
    from ⟨px ≠ []⟩ ⟨py ≠ []⟩ px py
    show ?thesis
    proof (induction arbitrary: v rule: rev-nonempty-induct2')
      case (single)
      hence (x,v) ∈ tree-edges s (y,v) ∈ tree-edges s
        by (simp-all add: path-simps)
      with tree-eq-rule have x=y by simp
      with x≠y show ?case by contradiction
    next
      case (snoc a as) hence (y,v) ∈ tree-edges s by (simp add: path-simps)
      moreover from snocl have path (tree-edges s) x as a (a,v) ∈ 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-trancl ⟨as ≠ []⟩ show ?case by metis
    next
      case (snocr - a as) hence (x,v) ∈ tree-edges s by (simp add: path-simps)
      moreover from snocr have path (tree-edges s) y as a (a,v) ∈ tree-edges s
        by (simp-all add: path-simps)
      ultimately have path (tree-edges s) y as x
        using tree-eq-rule
by auto
with path-is-trancl (as ≠ []): show ?case by metis
next
case (sncr a as b bs) hence
  path (tree-edges s) x as a (a,v) ∈ tree-edges s
  path (tree-edges s) y bs b (b,v) ∈ tree-edges s
  by (simp-all add: path-simp's)
moreover hence a=b using tree-eq-rule by simp
ultimately show ?thesis using sncr.IH by metis
qed
qed

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

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

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

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

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

lemma \text{tree-path-impl-parenthesis}:
assumes \((v, w) \in (\text{tree-edges } s)^+\)
and \(v \in \text{dom } (\text{finished } s)\)
shows \(w \in \text{dom } (\text{finished } s)\)
\(\land \delta s v < \delta s w\)
\(\land \varphi s w < \varphi s v\)
using assms
by (auto elim!: trancl-induct dest: tree-edge-impl-parenthesis)

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

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

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

lemma paren-imp-tree-reach:
is-invar paren-imp-tree-reach
unfolding paren-imp-tree-reach-def[abs-def]
proof (induct rule: is-invarI)
case (discover s s' u v)
  hence EQ[simp]: tree-edges s' = insert (hd (stack s), v) (tree-edges s)
  finished s' = finished s
  discovered s' = (discovered s)(v \mapsto \text{counter } s)
  by simp-all

from discover interpret DFS-invar where s=s by simp
from discover have NE[simp]: stack s \neq [] by (simp add: cond-alt)
show \(?\text{case}\)
proof (intro ballI impI)
fix $a \ b$

assume $F : a \in \text{dom} (\text{discovered } s') \ b \in \text{dom} (\text{finished } s')$
and $D : \delta' s' a < \delta' s' b \land (a \notin \text{dom} (\text{finished } s') \lor \varphi s' a > \varphi s' b)$

from $F$ finished-discovered discover have $b \neq v$ by auto
show $(a, b) \in (\text{tree-edges } s')^+$
proof (cases $a = v$)
  case True with $D : b \neq v$ have $\text{counter } s < \delta s b$ by simp
  also from $F$ have $b \in \text{dom} (\text{discovered } s)$
    using finished-discovered by auto
  with timing-less-counter have $\delta s b < \text{counter } s$ by simp
  finally have False .
thus ?thesis .
next
  case False with $\langle b \neq v \rangle$ $F \ D$ discover have $(a, b) \in (\text{tree-edges } s')^+$ by simp
thus ?thesis by (auto intro: trancl-mono-mp)
qed

next
  case $(\text{finish } s \ s' \ u)$
hence $\text{EQ}[\text{simp}]: \text{tree-edges } s' = \text{tree-edges } s$
    finished $s' = (\text{finished } s)(\text{hd } (\text{stack } s) \mapsto \text{counter } s)$
    discovered $s' = \text{discovered } s$
    stack $s' = \text{tl } (\text{stack } s)$
by simp-all

from finish interpret $\text{DFS-invar}$ where $s=s$ by simp
from finish have $\text{NE}[\text{simp}]: \text{stack } s \neq [\ ]$ by (simp add: cond-alt)
show ?case
proof (intro ballI impI)
  fix $a \ b$
  assume $F : a \in \text{dom} (\text{discovered } s') \ b \in \text{dom} (\text{finished } s')$
  and paren: $\delta' s' a < \delta' s' b \land (a \notin \text{dom} (\text{finished } s') \lor \varphi s' a > \varphi s' b)$
  hence $a \neq b$ by auto

  show $(a, b) \in (\text{tree-edges } s')^+$
  proof (cases $b = \text{hd } (\text{stack } s)$)
    case True hence $\varphi b : \varphi s' b = \text{counter } s$ by simp
    have $a \in \text{set } (\text{stack } s)$
      unfolding stack-set-def
    proof
      from $F$ show $a \in \text{dom} (\text{discovered } s)$ by simp
      from True $(a \neq b) \varphi b$ paren have $a \in \text{dom} (\text{finished } s) \longrightarrow \varphi s a > \text{counter } s$ by simp
      with timing-less-counter show $a \notin \text{dom} (\text{finished } s)$ by force
    qed
    with paren True on-stack-is-tree-path have $(a, b) \in (\text{tree-edges } s')^+$ by auto
    thus ?thesis by (auto intro: trancl-mono-mp)
  qed
next
case False note b-not-hd = this
show \( ?thesis \)
proof (cases a = hd (stack s))
case False with b-not-hd F paren finish show \( ?thesis \) by simp
next
case True with paren b-not-hd F have
\( a \in \text{dom} (\text{discovered } s) \) \( b \in \text{dom} (\text{finished } s) \)
\( \delta s a < \delta s b \)
by simp-all
moreover from True stack-not-finished have \( a \notin \text{dom} (\text{finished } s) \)
by simp
ultimately show \( ?thesis \) by (simp add: finish)
qed
qed
qed
next
case (new-root s s' v0) then interpret DFS-invar where \( s=s \) by simp
from new-root finished-discovered have \( v0 \notin \text{dom} (\text{finished } s) \) by auto
moreover note timing-less-counter finished-discovered
ultimately show \( ?case \) using new-root by clarsimp force
qed simp-all
end end

class DFS-invar begin context 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 \( v \in \text{dom} (\text{discovered } s) \)
and \( w \in \text{dom} (\text{finished } s) \)
and \( \delta s v < \delta s w \)
and \( v \notin \text{dom} (\text{finished } s) \)
sows \( (v,w) \in (\text{tree-edges } s)^+ \)
using s-paren-imp-tree-reach assms
by (auto simp add: paren-imp-tree-reach-def)

lemma parenthesis-impl-tree-path:
assumes \( v \in \text{dom} (\text{finished } s) \) \( w \in \text{dom} (\text{finished } s) \)
and \( \delta s v < \delta s w \phi s v > \phi s w \)
sows \( (v,w) \in (\text{tree-edges } s)^+ \)
proof –
from assms(1) have \( v \in \text{dom} (\text{discovered } s) \)
using finished-discovered by blast
with assms show \( ?thesis \)
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 ∈ dom (finished s) w ∈ dom (finished s)
shows (v, w) ∈ (tree-edges s)^+ ←→ δ s v < δ s w ∧ ϕ s v > ϕ s w
using assms
by (metis parenthesis-impl-tree-path tree-path-impl-parenthesis)

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

lemma finished-succ-path-in-tree:
assumes f: v ∈ dom (finished s)
and s: w ∈ succ v
and δ: δ s v < δ s w
shows (v, w) ∈ (tree-edges s)^+
using no-pending-succ-path-in-tree finished-no-pending finished-discovered
using assms
by blast
end

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

lemma i-cross-edges-finished: is-invar (λs. ∀ (u,v)∈cross-edges s.
 v ∈ dom (finished s) ∧ (u ∈ dom (finished s) → v s v < ϕ s w))
proof (induction rule: is-invarI-full)
case (finish s s' u e)
  interpret DFS-invar G param s by fact
  from finish stack-not-finished have u \notin \text{dom}(\text{finished s}) by auto
  with finish show \text{?case by (auto intro: timing-less-counter)}
next
  case (cross-edge s s' u v e)
  interpret DFS-invar G param s by fact
  from cross-edge stack-not-finished have u \notin \text{dom}(\text{finished s}) by auto
  with cross-edge show \text{?case by (auto intro: timing-less-counter)}
qed simp-all

end end

context DFS-invar begin context begin interpretation timing-syntax.
lemmas cross-edges-finished = i-cross-edges-finished[THEN make-invar-thm]

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

lemma cross-edges-finished-decr:
  \([\((u,v)\in\text{cross-edges s}; u\in\text{dom}(\text{finished s})\)] \Longrightarrow \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: w \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 param-DFS begin context begin interpretation timing-syntax.

lemma i-back-edge-impl-tree-path:
  is-invar \((\lambda s. \forall (v,w) \in \text{back-edges s.} (w,v) \in (\text{tree-edges s})^\top \lor w = v)\)
proof (induct rule: is-invarI-full)
  case (back-edge s s' u v) then interpret DFS-invar where s=s by simp

  from back-edge have st: v ∈ set (stack s) u ∈ set (stack s)
    using stack-set-def
    by auto

  have (v,u) ∈ (tree-edges s)⁺ ∨ u = v
  proof (rule disjCI)
    assume u ≠ v
    with st back-edge have v ∈ set (tl (stack s)) by (metis not-hd-in-tl)
    with tl-tl-stack-hd-discover st back-edge have δ s v < δ s u by simp
    with on-stack-is-tree-path st show (v,u) ∈ (tree-edges s)⁺ by simp
    qed
    with back-edge show ?case by auto
  next
    case discover thus ?case using trancl-sub-insert-trancl by force
  qed simp-all

end end

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

lemma back-edge-impl-tree-path:
  [((v,w) ∈ back-edges s; v ≠ w] → (w,v) ∈ (tree-edges s)⁺
  using i-back-edge-impl-tree-path[THEN make-invar-thm]
  by blast

lemma back-edge-disc:
  assumes (v,w) ∈ back-edges s
  shows δ s w ≤ δ s v
  proof cases
    assume v≠w
    with assms back-edge-impl-tree-path have (w,v) ∈ (tree-edges s)⁺ by simp
    with tree-path-disc show ?thesis by force
  qed simp

lemma back-edges-tree-disjoint:
  back-edges s ∩ tree-edges s = {}
  using back-edge-disc tree-edge-disc
  by force

lemma back-edges-tree-pathes-disjoint:
  back-edges s ∩ (tree-edges s)⁺ = {}
  using back-edge-disc tree-path-disc
  by force

lemma back-edge-finished:
  assumes (v,w) ∈ back-edges s
and $w \in \text{dom} (\text{finished } s)
\text{shows } v \in \text{dom} (\text{finished } s) \land \varphi \ s v \leq \varphi \ s w
\text{proof (cases } v = w)\n\text{case True with } \text{assms show } \varphi s v \leq \varphi s w\n\text{next}\n\text{case False with } \text{back-edge-impl-tree-path } \text{assms have } (w, v) \in (\text{tree-edges } s)^+\n\text{by simp}\n\text{with } \text{tree-path-impl-parenthesis } \text{assms show } \varphi s v \leq \varphi s w\n\text{qed}\nend end
\text{context } \text{param-DFS begin context begin interpretation } \text{timing-syntax .}
\text{lemma i-disc-imp-back-edge-or-pending: }\n\text{is-invar } (\lambda s. \forall (v, w) \in E.
\text{v } \in \text{dom} (\text{discovered } s) \land w \in \text{dom} (\text{discovered } s)
\land \delta s v \geq \delta s w
\land (w \in \text{dom} (\text{finished } s) \implies v \in \text{dom} (\text{finished } s) \land \varphi s w \geq \varphi s v)
\implies (v, w) \in \text{back-edges } s \lor (v, w) \in \text{pending } s)
\text{proof (induct rule: is-invarI-full)}\n\text{case (cross-edge } s s' u v) \text{ then interpret } \text{DFS-invar where } s= s \text{ by simp}\n\text{from cross-edge stack-not-finished[of u] have } u \notin \text{dom} (\text{finished } s)\n\text{using hd-in-set}\n\text{by (auto simp add: cond-alt)}\n\text{with cross-edge show } \varphi s v \leq \varphi s w\n\text{next}\n\text{case (finish } s s' u v) \text{ then interpret } \text{DFS-invar where } s= s \text{ by simp}\n\text{from finish have } \text{IH: } \forall v. v \in \text{succ } v; v \in \text{dom} (\text{discovered } s); w \in \text{dom} (\text{discovered } s);
\delta s v \leq \delta s w;
(w \in \text{dom} (\text{finished } s) \implies v \in \text{dom} (\text{finished } s) \land \varphi s w \leq \varphi s v)
\implies (v, w) \in \text{back-edges } s \lor (w, v) \in \text{pending } s\n\text{by blast}\n\text{from finish have } ne[simp]: \text{stack } s \neq []\n\text{and p[simp]: pending } s \mapsto \{\text{hd } (\text{stack } s)\} = \{\}
\text{by (simp-all)}\n\text{from hd-in-set[of ne] have } \text{disc: } \text{hd } (\text{stack } s) \in \text{dom} (\text{discovered } s)\n\text{and } \text{not-fin: } \text{hd } (\text{stack } s) \notin \text{dom} (\text{finished } s)\n\text{using stack-discovered stack-not-finished}\n\text{by blast}+\n\{\text{fix } w\n\text{assume w: } w \in \text{succ } (\text{hd } (\text{stack } s)) \neq \text{hd } (\text{stack } s) \in \text{dom} (\text{discovered } s)\n\text{and } f: w \in \text{dom} (\text{finished } s) \implies \text{counter } s \leq \varphi s w\n\text{by blast};\n\text{qed}\nend end
\[\delta: \delta s w \leq \delta s (hd (stack s))\]

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

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

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

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

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

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

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

lemma finished-imp-back-edge:
  \[w \in succ v; v \in dom (finished s); w \in dom (finished s);\]
  \[\delta s w \leq \delta s v; \varphi s v \leq \varphi s w\]
  \implies \((v, w) \in back-edges s\]
using disc-imp-back-edge-or-pending finished-discovered finished-no-pending
by fast

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

lemma finished-self-loop-in-back-edges:
assumes \( v \in \text{dom (finished } s) \)
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 (tree-edges s \cup cross-edges s) (is acyclic \(?E\))
proof (rule ccontr)
{ fix u v
  assume \(*: u \in \text{dom (finished } s) \) and \((u, v) \in \text{?E}^+\)
  from this(2) have \(\varphi s v < \varphi s u \land v \in \text{dom (finished } s)\)
  proof induct
    case base thus ?case
    next case (step v w)
    hence \(\varphi s w < \varphi s v \land w \in \text{dom (finished } s)\)
    by (metis Un-iff * cross-edges-finished-decr cross-edges-target-finished
    tree-edge-impl-parenthesis)
  qed
} note aux = this

assume \(\neg \text{acyclic } ?E\)
then obtain u where path: \((u, u) \in \text{?E}^+\) by (auto simp add: acyclic-def)
show False

58
proof cases
  assume u ∈ dom (finished s)
  with aux path show False by blast
next
  assume *: u ∉ dom (finished s)
  moreover
  from no-loop-in-tree have (u,u) ∉ (tree-edges s)⁺.
  with trancl-union-outside[OF path] obtain x y where (u,x) ∈ ?E⁺ (x,y) ∈
  cross-edges s (y,u) ∈ ?E⁺ by auto
  with cross-edges-target-finished have y ∈ dom (finished s) by simp
  moreover with * ⟨(y,u) ∈ ?E⁺⟩ have (y,u) ∈ ?E⁺ by (auto simp add: rtrancl-eq-or-trancl)
  ultimately show False by (metis aux)
qed
qed end

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

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

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

\[\text{qed}\]

**lemma** back-edge-closes-reachable-cycle:
\[
\text{back-edges } s \neq \{\} \implies \neg \text{acyclic } (E \cap \text{reachable } \times \text{UNIV})
\]
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)

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:
\[
\text{it-dfs} \leq \text{SPEC} (\lambda s. \forall x \in \text{reachable} \quad \forall y \in \text{reachable} \quad \neg \text{is-break param } s \rightarrow \text{white-path } s x y \leftrightarrow (x,y) \in (\text{tree-edges } s)^*)
\]
proof (rule it-dfs-SPEC, intro ballI impI)

fix \(s x y\)

assume \(DI: \text{DFS-invar } G \text{ param } s\)

and \(C: \neg \text{cond } s \quad \neg \text{is-break param } s\)

and \(reach: x \in \text{reachable} y \in \text{reachable}\)

from \(DI\) interpret \(\text{DFS-invar where } s=s\).

**note** fin-eq-reach = nc-finished-eq-reachable[\(OF C\)]

show white-path \(s x y \leftrightarrow (x,y) \in (\text{tree-edges } s)^*\)

proof (cases \(x\equiv y\))

case True thus \(\text{thesis by (simp add: white-path-def)}\)

next

case False

show \(\text{thesis}\)

proof

assume \((x,y) \in (\text{tree-edges } s)^*\)

with \(\langle x\neq y\rangle\) have \(T: (x,y) \in (\text{tree-edges } s)^+\) by (metis rtranclD)

then obtain \(P\) where \(P: \text{path } (\text{tree-edges } s) x p y\) by (metis trancl-is-path)

with \(\text{tree-edges-ssE}\) have \(\text{path } E x p y\) using path-mono[\(\text{where } E=\text{tree-edges } s\)]

by simp

60
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 \( x \neq y \).

proof (induct rule: path-tl-induct)
  case (single \( u \)) thus ?case by (fact tree-edge-disc)
next
  case (step \( u \) \( v \)) note \( \delta s x < \delta s u \)
also from step have \( \delta s u < \delta s v \) by (metis tree-edge-disc)
finally show ?case.
qed
ultimately show white-path \( s x y \)
by (auto simp add: \( x \neq y \) white-path-def)

next
assume white-path \( s x y \)
with \( x \neq y \) obtain \( p \) where
\[ P : \text{path } E x p y \text{ and } \]
white: \( \delta s x < \delta s y \land (\forall v \in \text{set } (tl p), \delta s x < \delta s v) \)
unfolding white-path-def
by blast
hence \( p \neq [] \) by auto
thus \( (x,y) \in (\text{tree-edges } s)^* \) using \( P \) white reach(2)

proof (induction \( p \) arbitrary: \( y \) rule: rev-nonempty-induct)
  case single hence \( y \in \text{succ } x \) by (simp add: path-cons-conv)
  with reach single show ?case using fin-eq-reach finished-succ-impl-path-in-tree[of \( x \) \( y \)]
  by simp
next
  case (snoc \( u \) \( us \)) hence path \( E x u p y \) by (simp add: path-append-conv)
moreover hence \( (x,u) \in E^* \) by (simp add: path-is-rtrancl)
  with reach have ureach: \( u \in \text{reachable} \)
  by (metis rtrancl-image-advance-rtrancl)
moreover from snoc have \( \delta s x < \delta s u \land (\forall v \in \text{set } (tl us), \delta s x < \delta s v) \)
  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(\( \delta \)) fin-eq-reach finished-discovered have
\( y-f-d \): \( y \in \text{dom } (\text{finished } s) \) \( y \in \text{dom } (\text{discovered } s) \)
by auto

from \( y \in \text{succ } w \) ureach fin-eq-reach have \( \delta s y < \varphi s u \)
using finished-succ-fin by simp
also from \( \delta s x < \delta s w \) have \( x \neq u \) by auto
with \( x-u \) have \( (x,u) \in (\text{tree-edges } s)^* \) by (metis rtrancl-eq-or-trancl)
with fin-eq-reach 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

61
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
end end

1.3 Invariants for SCCs

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

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

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

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

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

context DFS-invar begin
lemma \(\text{scc-root-is-discovered:}\)
lemma \textit{scc-root-scc-tree-rtrancl}:
\begin{itemize}
\item \textbf{assumes} \( \text{scc-root } s \ v \ scc \) 
\item \textbf{and} \( x \in scc \ x \in \text{dom (discovered } s \) 
\item \textbf{shows} \( (v,x) \in (\text{tree-edges } s)^* \)
\item \textbf{using} \text{assms}
\item \textbf{by} \( \text{(auto simp add: scc-root-def)} \)
\end{itemize}

lemma \textit{scc-root-scc-reach}:
\begin{itemize}
\item \textbf{assumes} \( \text{scc-root } s \ r \ scc \) 
\item \textbf{shows} \( (r,v) \in E^* \)
\item \textbf{proof} 
\item \textbf{from} \text{assms} \textbf{have} \( \text{is-scc } E \ scc \ r \in scc \)
\item \textbf{with} \text{is-scc-connected} \text{assms} \textbf{show} \( \text{?thesis by metis} \)
\end{itemize}

lemma \textit{scc-reach-scc-root}:
\begin{itemize}
\item \textbf{assumes} \( \text{scc-root } s \ r \ scc \) 
\item \textbf{shows} \( (v,r) \in E^* \)
\item \textbf{proof} 
\item \textbf{from} \text{assms} \textbf{have} \( \text{is-scc } E \ scc \ r \in scc \)
\item \textbf{with} \text{is-scc-connected} \text{assms} \textbf{show} \( \text{?thesis by metis} \)
\end{itemize}

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

lemma \textit{scc-root-unique-scc}:
\begin{itemize}
\item \( \text{scc-root } s \ v \ scc \Rightarrow scc-root \ s \ v \ scc' \Rightarrow scc = scc' \)
\item \textbf{unfolding} \text{scc-root-def} 
\item \textbf{by} \( \text{metis is-scc-unique} \)
\end{itemize}

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

lemma scc-root-unique-is-scc:
assumes scc-root s v scc
shows scc-root s v (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-of 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 by metis
thus x \in \text{dom} \) (finished s)
proof (induction p arbitrary: x rule: rev-induct)
case Nil hence v = x by simp
with assms show \( ?\)case by simp
next
case (snoc y ys) hence path \?E v y s 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 \( (y,x) \in \?E \) have \( (y,x) \in E x \in scc \) by auto
ultimately have x \in \text{dom} \) (discovered s)
using finished-imp-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 \( x \in \text{scc} \cap \text{dom} (\text{discovered s}) \)
  shows \( \delta s v \leq \delta s x \)
proof (cases \( x = v \))
  case False with assms scc-root-tree-trancl tree-path-disc have
  \( \delta s v < \delta s x \)
  by blast
  thus \( \text{thesis by simp} \)
qed simp

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

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

lemma Min-disc-is-scc-root:
  assumes \( v \in \text{scc} \cap \text{dom} (\text{discovered s}) \)
  and \( \text{is-scc E scc} \)
  and \( \text{min: } \delta s v = \text{Min} (\delta s' (\text{scc} \cap \text{dom} (\text{discovered s}))) \)
  shows scc-root s v scc
proof
{ fix \( y \)
  assume A: \( y \in \text{scc} \cap \text{dom} (\text{discovered s}) \) \( y \neq v \)
with min have δ s v ≤ δ s y by auto
with assms disc-unequal A have δ s v < δ s y by fastforce
} note scc-disc = this

{ fix x
  assume A: x ∈ scc ∩ dom (discovered s)
  have x ∈ (tree-edges s)* " {v}
  proof (cases v = x)
    case False with A scc-disc have δ: δ s v < δ s x by simp
    have (v,x) ∈ (tree-edges s)+
    proof (cases v ∈ dom (finished s))
      case False with stack-set-def assms have
      v-stack: v ∈ set (stack s) by auto
      show ?thesis
      proof (cases x ∈ dom (finished s))
        case True with parenthesis-impl-tree-path-not-finished[of v x] assms δ False show ?thesis by auto
        next
        case False with stack-set-def have x ∈ set (stack s) by auto
        with v-stack δ show ?thesis
        using on-stack-is-tree-path
        by simp
      qed
    next
    case True note v-fin = this
    let ?E = Restr E scc
    { fix y
      assume (v, y) ∈ ?E and v ≠ y
      hence *: y ∈ succ v y ∈ scc by auto
      with finished-imp-succ-discovered v-fin have
      y ∈ dom (discovered s) by simp
      with scc-disc ⟨v ≠ y⟩ * have δ s v < δ s y by simp
      with * finished-succ-impl-path-in-tree v-fin have (v,y) ∈ (tree-edges s)+
      by simp
    } note trancl-base = this
    from A have x ∈ scc by simp
    with assms have (v,x) ∈ ?E*
    by (simp add: is-scc-connected')
    with ⟨v≠x⟩ have (v,x) ∈ ?E+ by (metis rtrancl-eq-or-trancl)
    thus ?thesis using ⟨v≠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 ∈ scc 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
qed
qed
thus ?thesis by auto
qed simp

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

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

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

lemma scc-root-exists:
assumes is-scc E scc and \(\text{scc} \cap \text{dom} (\text{discovered} \ s) \neq \{}\)
shows \(\exists r. \text{scc-root} s r scc\)
proof
let \(?S = \text{scc} \cap \text{dom} (\text{discovered} \ s)\)

from discovered-finite have finite (\(\delta s'??S\)) by auto
moreover from scc have \(\delta s ??S \neq \{}\) by auto
moreover have \(\exists x. f A. x \neq f' A \lor (\exists y. x = f y \land y \in A)\) by blast
— autogenerated by sledgehammer
ultimately have \(\exists x \in ?S. \delta s x = \text{Min} (\delta s ??S)\) by (metis Min-in)
with Min-disc-is-scc-root (is-scc E scc) show ?thesis by auto
qed

lemma scc-root-of-node-exists:
assumes \(v \in \text{dom} (\text{discovered} \ s)\)
shows \(\exists r. \text{scc-root} s r (\text{scc-of} E v)\)
proof
have is-scc E (\text{scc-of} E v) by simp
moreover have \(v \in \text{scc-of} E v\) by simp
with assms have \(\text{scc-of} E v \cap \text{dom} (\text{discovered} \ s) \neq \{}\) by blast
ultimately show ?thesis using scc-root-exists by metis
qed

lemma scc-root-transfer' :
assumes discovered s = discovered s' tree-edges s = tree-edges s'
shows scc-root s r scc = scc-root s' r scc
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')
    ∀ x ∈ dom (discovered s). δ s x = δ s' x
    ∀ x ∈ dom (discovered s') - dom (discovered s). δ s' x ≥ counter s
  and t: tree-edges s ⊆ tree-edges s'
  shows scc-root s r scc ←→ scc-root s' r scc
proof –
  interpret s': DFS-invar where s=s' by fact

let ?sd = scc ∩ dom (discovered s)
let ?sd' = scc ∩ dom (discovered s')
let ?sdd = scc ∩ (dom (discovered s') - dom (discovered s))

{ Assume r-s: r ∈ scc is-scc E scc
  with r-d have ne: δ s' ?sd' ≠ {} by blast
  from discovered-finite have fin: finite (δ s' ?sd) by simp

  from timing-less-counter d have ∃ x. x ∈ δ s' ?sd' x < counter s by auto
  hence Min: Min (δ s' ?sd) < counter s
  using Min-less-iff [OF fin ne] by blast

  from d have Min (δ s' ?sd) = Min (δ s' ?sd') by (auto simp: image-def)
  also from d have ?sd' = ?sd ∪ ?sdd by auto
  hence s': δ s' ?sd' = δ s' ?sd ∪ δ s' ?sdd by auto
  hence Min (δ s' ?sd') = Min (δ s' ?sd') by auto
  proof (cases ?sdd = {})
    case False
    from d have ∃ x. x ∈ δ s' ?sdd x ≥ counter s by auto
    moreover from False have ne': δ s' ?sdd ≠ {} by blast
    moreover from s'.discovered-finite have fin': finite (δ s' ?sdd) by blast
    ultimately have Min (δ s' ?sdd) ≥ counter s
    using Min-ge-iff by metis
    with Min Min-Un [OF fin ne fin' ne'] show ?thesis by simp
  qed simp
  finally have Min (δ s' ?sd) = Min (δ s' ?sd')
  }
  note aux = this

show ?thesis
proof
  assume r: scc-root s r scc
  from r-d have δ s' r = δ s r by simp
  also from r scc-root-is-Min-disc have δ s r = Min (δ s ?sd) by simp
  also from r aux have Min (δ s ?sd) = Min (δ s' ?sd') by (simp add: scc-root-def)
  finally show scc-root s' r scc
using r-d d r[unfolded scc-root-def]
by (blast intro: s'.Min-disc-is-scc-root)
next
assume r': scc-root s' r sce
from r-d d have δ s r = δ s' r by simp
also from r' s', scc-root-is-Min-disc have δ s' r = Min (δ s' ' ?sd') by simp
also from r' aux have Min (δ s' ' ?sd') = Min (δ s ' ?sd) by (simp add: scc-root-def)
finally show scc-root s r sce
  using r-d d r'[unfolded scc-root-def]
  by (blast intro: Min-disc-is-scc-root)
qed
qed

end end

1.4 Generic DFS and Refinement

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

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

named-theorems DFS-code-unfold (DFS framework: Unfolding theorems to prepare term for automatic refinement):

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

1.4.1 Generic DFS Algorithm

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


\[
gds\text{-discover} :: \ 'v \Rightarrow \ 'v \Rightarrow \ 's \Rightarrow \ 's \text{ nres}
\]

\textbf{locale} gen\text{-}dfs\text{-}defs =
\begin{itemize}
\item \textbf{fixes} gds :: ('v,'s) gen\text{-}dfs\text{-}struct
\item \textbf{fixes} V0 :: 'v set
\end{itemize}

\textbf{begin}

\textbf{definition} gen\text{-}step s ≡
\begin{itemize}
\item if \text{gds\text{-}is\text{-}empty\text{-}stack} gds s then do 
\begin{itemize}
\item \text{v0 } \leftarrow \text{SPEC } (\lambda v0. \text{v0} \in V0 \land \neg \text{gds\text{-}is\text{-}discovered} gds v0 s);
\item \text{gds\text{-}new\text{-}root} gds v0 s
\end{itemize}
\item else do 
\begin{itemize}
\item (u, Vs, s) \leftarrow \text{gds\text{-}get\text{-}pending} gds s;
\item case Vs of
\begin{itemize}
\item None ⇒ \text{gds\text{-}finish} gds u s
\item Some v ⇒ do 
\begin{itemize}
\item if \text{gds\text{-}is\text{-}discovered} gds v s then ( 
\begin{itemize}
\item if \text{gds\text{-}is\text{-}finished} gds v s then 
\text{gds\text{-}cross\text{-}edge} gds u v s 
\item else 
\text{gds\text{-}back\text{-}edge} gds u v s 
\end{itemize}
\item else 
\text{gds\text{-}discover} gds u v s
\end{itemize}
\end{itemize}
\end{itemize}
\end{itemize}
\end{itemize}

\textbf{definition} gen\text{-}cond s ≡
\begin{itemize}
\item \text{V0} \subseteq \{ v. \text{gds\text{-}is\text{-}discovered} gds v s \} \rightarrow \neg \text{gds\text{-}is\text{-}empty\text{-}stack} gds s
\item \neg \text{gds\text{-}is\text{-}break} gds s
\end{itemize}

\textbf{definition} gen\text{-}dfs ≡ gds\text{-}init gds \gg \text{WHILE gen\text{-}cond gen\text{-}step}

\textbf{definition} gen\text{-}dfsT ≡ gds\text{-}init gds \gg \text{WHILET gen\text{-}cond gen\text{-}step}

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

\textbf{abbreviation} gen\text{-}rwof ≡ \text{rwof (gds\text{-}init} gds) gen\text{-}cond gen\text{-}step

\textbf{definition} pre\text{-}new\text{-}root v0 s ≡
\begin{itemize}
\item gen\text{-}rwof s \land \text{gds\text{-}is\text{-}empty\text{-}stack} gds s \land \neg \text{gds\text{-}is\text{-}break} gds s
\item v0 \in V0 \land \neg \text{gen\text{-}discovered} s
\end{itemize}

\textbf{definition} pre\text{-}get\text{-}pending s ≡
\begin{itemize}
\item gen\text{-}rwof s \land \neg \text{gds\text{-}is\text{-}empty\text{-}stack} gds s \land \neg \text{gds\text{-}is\text{-}break} gds s
\end{itemize}

71
**definition** post-get-pending v Vs s0 s ≡ pre-get-pending s0 ∧ inres (gds-get-pending gds s0) (u, Vs, s)

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

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

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

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

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

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

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

**definition** gen-dfsT-assert ≡ gds-init gds ≫ WHILET gen-cond gen-step-assert
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)
apply (refine-rcg simp-all)
by (auto simp: pre-defs gen-cond-def)
done

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

subgoal
  apply (subst (2) WHLET-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', 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-asm) 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-asm) only: gen-step-def[abs-def] gen-step-assert-def[abs-def]) []
  apply (unfold GHOST-elim-Let) []
  apply (rule leofI)
apply (rule refine-IdD)
  by (refine-rcg bind-refine', refine-dref-type,
      auto simp: pre-defs gen-cond-def) []
done

lemma gen-dfs-le-gen-dfsT: gen-dfs ≤ gen-dfsT
  unfolding gen-dfs-def gen-dfsT-def
  apply (rule bind-mono)
  apply simp
  unfolding WHILET-def WHILE-def
  apply (rule WHILEI-le-WHILEIT)
done

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

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

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

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

75
\begin{verbatim}

\textbf{definition} \textit{do-init} ≡ do { \\
  e ← \textit{on-init} \textit{param} ; \\
  \textit{gbs-init} \textit{gbs} \, e \\
}

\textbf{definition} \textit{do-new-root} \, v0 \\
  ≡ \textit{do-action} (\textit{gbs-new-root} \, \textit{gbs} \, v0) \, (\textit{on-new-root} \textit{param} \, v0)

\textbf{definition} \textit{do-finish} \, u \\
  ≡ \textit{do-action} (\textit{gbs-finish} \, \textit{gbs} \, u) \, (\textit{on-finish} \textit{param} \, u)

\textbf{definition} \textit{do-back-edge} \, u \, v \\
  ≡ \textit{do-action} (\textit{gbs-back-edge} \, \textit{gbs} \, u \, v) \, (\textit{on-back-edge} \textit{param} \, u \, v)

\textbf{definition} \textit{do-cross-edge} \, u \, v \\
  ≡ \textit{do-action} (\textit{gbs-cross-edge} \, \textit{gbs} \, u \, v) \, (\textit{on-cross-edge} \textit{param} \, u \, v)

\textbf{definition} \textit{do-discover} \, u \, v \\
  ≡ \textit{do-action} (\textit{gbs-discover} \, \textit{gbs} \, u \, v) \, (\textit{on-discover} \textit{param} \, u \, v)

  \textit{do-finish-def} \, \textit{do-back-edge-def} \, \textit{do-cross-edge-def} \, \textit{do-discover-def}

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


\textbf{sublocale} \textit{gen-dfs-defs} \textit{gds} \, V0 . \\
\end{verbatim}
and \( V_0 :: 'v \text{ set} \)

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

definition \( gbs \equiv \{} \\
gbs-init = \text{RETURN} o \text{ empty-state} , \\
gbs-is-empty-stack = \text{is-empty-stack} , \\
gbs-new-root = \text{RETURN} oo \text{ new-root} , \\
gbs-get-pending = \text{get-pending} , \\
gbs-finish = \text{RETURN} oo \text{ finish} , \\
gbs-is-discovered = \text{is-discovered} , \\
gbs-is-finished = \text{is-finished} , \\
gbs-back-edge = \text{RETURN} ooo \text{ back-edge} , \\
gbs-cross-edge = \text{RETURN} ooo \text{ cross-edge} , \\
gbs-discover = \text{RETURN} ooo \text{ discover} \\
\}

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

sublocale \( \text{gen-dfs: gen-param-dfs-defs gbs param state more-update V0} . \)

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

lemma \( \text{gen-step-simp}[\text{simp}]:\text{gen-dfs.gen-step} = \text{step} \) apply (intro ext)
unfolding \( \text{gen-dfs.gen-step-def}[\text{abs-def}] \)
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 \( \text{gen-init-simp}[\text{simp}]:\text{gen-dfs.do-init} = \text{init} \)
unfolding init-def
apply (simp add: gen-dfs.do-action-defs[abs-def])
done

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

lemma \( \text{gen-dfsT-simp}[\text{simp}]:\text{gen-dfs.gen-dfsT} = \text{it-dfsT} \)

77
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
for gds V0
locale gen-dfs-refine =
c: gen-dfs gdsi V0i + a: gen-dfs gds V0
for gdsi V0i gds V0
fixes V S
assumes BIJV[relator-props]: bijective V
assumes V0-param[param]: (V0i, V0) ∈ (V set-rel
assumes is-discovered-param[param]:
  (gds-is-discovered gdsi, gds-is-discovered gds) ∈ V → S → bool-rel
assumes is-finished-param[param]:
  (gds-is-finished gdsi, gds-is-finished gds) ∈ V → S → bool-rel
assumes is-empty-stack-param[param]:
  (gds-is-empty-stack gdsi, gds-is-empty-stack gds) ∈ S → bool-rel
assumes is-break-param[param]:
  (gds-is-break gdsi, gds-is-break gds) ∈ S → bool-rel
assumes init-refine[refine]:
gds-init gdsi ≤⇓ S (gds-init gds)
assumes new-root-refine[refine]:
[a.pre-new-root v0 s; (v0i, v0) ∈ V; (si, s) ∈ S]
⇒ gds-new-root gdsi v0i s i ≤⇓ 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 s i ≤⇓ 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 s i ≤⇓ 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 s i ≤⇓ S (gds-back-edge gds u v s)
assumes discover-refine[refine]:

[\text{pre-discover} u \: v \: s0 \: s \: (u, i) \in V; \: (v, i) \in V; \: (s, i) \in S] 
\implies \text{gds-discover} \: gdsi \: u \: i \: \leq \nexists S \: (\text{gds-discover} \: gds \: u \: v \: s)

\text{begin}
\text{term} \ gds\text{-is-discovered} \ gdsi

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

\text{lemma} \ \text{gen-rwof-refine}:
\text{assumes} \ NF: \ \text{nofail} \ (a.\text{gen-dfs})
\text{assumes} \ RW: \ c.\text{gen-rwof} \ s
\text{obtains} \ s' \ where \ (s, s') \in S \land a.\text{gen-rwof} \ s'
\text{proof} –
\text{from} \ NF \ \text{have} \ NFa: \ \text{nofail} \ (a.\text{gen-dfs-assert})
\text{unfolding} \ a.\text{gen-dfs-eq-assert} .
\text{have} \ \exists s'. \ (s, s') \in S \land a.\text{gen-rwof-assert} \ s'
\text{apply} (\text{rule} \ rwof\text{-refine}[OF RW NFa[unfolded a.\text{gen-dfs-assert-def}]])
\text{apply} (\text{rule} \ leofI, \text{rule} \ init\text{-refine})
\text{unfolding} \ c.\text{gen-cond-def} \ a.\text{gen-cond-def}
\text{apply} (\text{rule} \ IdD)
\text{apply} (\text{simp only: subset-Collect-conv})
\text{apply} \ \text{parametricity}
\text{unfolding} \ c.\text{gen-step-def} \ a.\text{gen-step-assert-def} \ GHOST\text{-elim-Let}
\text{apply} (\text{rule} \ leofI)
\text{apply} (\text{refine-reg} \ IdD)
\text{apply} \ \text{simp-all}
\text{apply} ((\text{rule} \ IdD, \text{parametricity}) \ | \ (\text{auto}) [])+
\text{done}
\text{thus} \ \text{thesis}
\text{unfolding} \ a.\text{gen-rwof-eq-assert}[OF NF, \text{symmetric}]
\text{by} \ (\text{blast intro: that})
\text{qed}
lemma gen-step-refine[refine]: (si,s) ∈ S \implies 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-reg 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-reg
  unfolding c.gen-cond-def a.gen-cond-def
  apply (rule IdD)
  apply (simp only: subset-Collect-come)
  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-reg
  unfolding c.gen-cond-def a.gen-cond-def
  apply (rule IdD)
  apply (simp only: subset-Collect-come)
  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 BIJV: bijective V
  assumes V0-param[param]: (V0i, V0) ∈ (V) set-rel

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

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

  assumes is-empty-stack-param[param]:
  (gbs-is-empty-stack gbsi, gbs-is-empty-stack gbs) ∈ S → bool-rel
assumes is-break-param[param]:
(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 ≤ \downarrow (V ×_r (V_{option-rel} ×_r 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 si ≤ 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)

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

assumes on-new-root-refine[refine]:
[a.pre-new-root v0 s; (v0i, v0) ∈ V; (si, s) ∈ S;
(si′, s′) ∈ S; nf-inres (gbs-new-root gbs v0 s) s′]
⇒ on-new-root parami v0i si′ ≤ S (on-new-root param v0 s′)
assumes on-finish-refine[refine]:
\[ a.\text{pre-finish} v s0 s; (vi, v) \in V; (si, s) \in S; (si', s') \in S; \\
\text{nf-inres} (gbs-finish gbs v s) s' \]
\[ \implies \text{on-finish parami vi si}' \leq \downarrow ES \text{ (on-finish param v s')} \]

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

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

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

locale gen-param-dfs-refine-defs =
c: gen-param-dfs-defs gbsi parami upd-exti V0i +
a: gen-param-dfs-defs gbs param upd-ext V0
for gbsi parami upd-exti V0i gbs param upd-ext V0
begin
  sublocale gen-dfs-refine-defs c.gds V0i a.gds V0 .
end

locale gen-param-dfs-refine =
gbs-refinement where V=V and S=S and ES=ES +
param-refinement where V=V and S=S and ES=ES +
gen-param-dfs-refine-defs
for V :: ('vi'v) set and S:: ('si's) set and ES :: ('esix'es) set
begin
  sublocale gen-dfs-refine c.gds V0i a.gds V0 V S
  apply unfold-locales
  apply (simp-all add: BUJV V0-param a.do-action-defs c.do-action-defs)
  apply (parametricity+) [4]
apply refine-rcg
apply (refine-rcg bind-refine-abs', assumption+, parametricity) []
apply refine-rcg
apply (refine-rcg bind-refine-abs', assumption+, parametricity) []
apply refine-rcg
apply (refine-rcg bind-refine-abs', assumption+, parametricity) []
apply refine-rcg
apply (refine-rcg bind-refine-abs', assumption+, parametricity) []
apply refine-rcg
apply (refine-rcg bind-refine-abs', assumption+, parametricity) []
done
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-is-empty-stack gds s)
  ∧ V0−it ⊆ gen-discovered s)
  V0
  (Not o gds-is-break gds)
  (λv0 s. do {
    let — ghost: s0 = s;
    if gds-is-discovered gds v0 s then
      RETURN s
    else do {
      s ← gds-new-root gds v0 s;
      WHILEIT
  }
  end
  end

end

end

end
\[
\begin{align*}
(\lambda s. \text{gen-\text{rwof} } s \land \text{insert } v0 \ (\text{gen-discovered } s0) \subseteq \text{gen-discovered } s) \\
(\lambda s. \neg \text{gds-is-break} \ \text{gds} \ s \land \neg \text{gds-is-empty-stack} \ \text{gds} \ s) \\
\text{tr-\text{impl-while-body} } s
\end{align*}
\]

\[
\}
\]

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

\[
\text{tailrec-impl} \equiv \text{do} \{ \\
\text{s} \leftarrow \text{gds-init gds;}
\}
\]

**FOREACH**

\[
(\lambda it s. \\
\text{gen-\text{rwof} } s \\
\land (\neg \text{gds-is-break} \ \text{gds} \ s \rightarrow \text{gds-is-empty-stack} \ \text{gds} \ s) \\
\land V0 \leftarrow \text{it} \subseteq \text{gen-discovered} \ s)
\]

\[
\text{V0}
\]

\[
(\text{Not o gds-is-break gds})
\]

\[
(\lambda v0 s. \text{do} \{ \\
\text{let \text{ghost:} s0 = s;}
\}
\]

\[
\text{if gds-is-discovered gds v0 s then}
\]

\[
\text{RETURN s}
\]

\[
\text{else do} \{ \\
\text{s} \leftarrow \text{gds-new-root gds v0 s;}
\}
\]

\[
\text{WHILEI}
\]

\[
(\lambda s. \text{gen-\text{rwof} } s \land \text{insert } v0 \ (\text{gen-discovered } s0) \subseteq \text{gen-discovered } s) \\
(\lambda s. \neg \text{gds-is-break} \ \text{gds} \ s \land \neg \text{gds-is-empty-stack} \ \text{gds} \ s) \\
(\lambda s. \text{do} \{ \\
\text{(u, Vs, s)} \leftarrow \text{gds-get-pending gds s;}
\}
\]

\[
\text{case Vs of}
\]

\[
\text{None} \Rightarrow \text{gds-finish gds u s}
\]

\[
\text{| Some v} \Rightarrow \text{do} \{ \\
\text{if gds-is-discovered gds v s then do} \{ \\
\text{if gds-is-finished gds v s then}
\]

\[
\text{gds-cross-edge gds u v s}
\]

\[
\text{else}
\]

\[
\text{gds-back-edge gds u v s}
\]

\[
\}
\text{else}
\]

\[
\text{gds-discover gds u v s}
\]

\[
\}
\}
\}
\]

\[
\}
\]

end

Implementation of general DFS with outer foreach-loop

**locale** tailrec-impl =
fb-graph $G$ + gen-dfs gds $V0$ + tailrec-impl-defs $G$ gds
for $G$ :: (′v, ′more) graph-rec-scheme
and gds :: (′v,s)gen-dfs-struct +
assumes init-empty-stack:
gds-init gds $\leq_n$ SPEC (gds-is-empty-stack gds)
assumes new-root-discovered:
\[
\begin{align*}
\text{[pre-new-root $v0$ s]} & \implies \text{gds-new-root gds $v0$ s} \leq_n \text{SPEC (λs'}. \\
& \text{insert $v0$ (gen-discovered s) $\subseteq$ gen-discovered s')}
\end{align*}
\]
assumes get-pending-incr:
\[
\begin{align*}
\text{[pre-get-pending s]} & \implies \text{gds-get-pending gds s} \leq_n \text{SPEC (λ(\cdot,s').} \\
& \text{gen-discovered s $\subseteq$ gen-discovered s')}
\end{align*}
\]
assumes finish-incr: [pre-finish $u$ $s0$ s]
\[
\begin{align*}
& \implies \text{gds-finish gds u s} \leq_n \text{SPEC (λs'.} \\
& \text{gen-discovered s $\subseteq$ gen-discovered s')}
\end{align*}
\]
assumes cross-edge-incr: pre-cross-edge $u$ $v$ $s0$ s
\[
\begin{align*}
& \implies \text{gds-cross-edge gds u v s} \leq_n \text{SPEC (λs'.} \\
& \text{gen-discovered s $\subseteq$ gen-discovered s')}
\end{align*}
\]
assumes back-edge-incr: pre-back-edge $u$ $v$ $s0$ s
\[
\begin{align*}
& \implies \text{gds-back-edge gds u v s} \leq_n \text{SPEC (λs'.} \\
& \text{gen-discovered s $\subseteq$ gen-discovered s')}
\end{align*}
\]
assumes discover-incr: pre-discover $u$ $v$ $s0$ s
\[
\begin{align*}
& \implies \text{gds-discover gds u v s} \leq_n \text{SPEC (λs'.} \\
& \text{gen-discovered s $\subseteq$ gen-discovered s')}
\end{align*}
\]
begin

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

lemma gds-init-refine: gds-init gds
\[
\leq \text{SPEC (λs. gen-rwof s $\land$ gds-is-empty-stack gds s)}
\]
apply (rule SPEC-rule-conj-leofI1)
apply (rule order-trans [OF - rwof-step [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
\[
\leq \text{SPEC (λs'. gen-rwof s' $\land$ insert $v0$ (gen-discovered s) $\subseteq$ gen-discovered s'}
\]
apply (rule SPEC-rule-conj-leofI1)
apply (rule order-trans [OF - rwof-step [OF nofail]])
using PNR apply (unfold gen-step-def gen-cond-def pre-new-root-def) [3]

85
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-leofI1)

qed
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-increment[OF PRE])
done

lemma gds-cross-edge-refine:
assumes PRE: pre-cross-edge u v s0 s
shows gds-cross-edge gds u v s \leq SPEC (λs'. gen-rwof s'
∧ gen-discovered s \subseteq gen-discovered s')
apply (rule SPEC-rule-conj-leofI1)

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

apply (rule cross-edge-increment[OF PRE])
done

lemma gds-back-edge-refine:
assumes PRE: pre-back-edge u v s0 s
shows gds-back-edge gds u v s \leq SPEC (λs'. gen-rwof s'
∧ gen-discovered s \subseteq gen-discovered s')
apply (rule SPEC-rule-conj-leofI1)

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

apply (rule back-edge-increment[OF PRE])
done

lemma gds-discover-refine:
assumes PRE: pre-discover u v s0 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-leafII)

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

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

end

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

theorem tailrec-impl: tailrec-impl $\leq$ gen-dfs
unfolding gen-dfs-def
apply (rule WHILE-refine-rwof)
unfolding tailrec-impl-def
apply (refine-req refine-veg
order-trans[OF gds-init-refine]
order-trans[OF gds-new-root-refine]
order-trans[OF gds-get-pending-refine]
order-trans[OF gds-discover-refine]

88
order-trans[OF gds-finish-refine]
order-trans[OF gds-cross-edge-refine]
order-trans[OF gds-back-edge-refine]
order-trans[OF gds-discover-refine]

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

done

lemma tr-impl-while-body-gen-step:
  assumes [simp]: ¬gds-is-empty-stack gds s
  shows tr-impl-while-body s ≤ 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: nofail gen-dfsT
  from nofail-WHILEIT-wf-rel[of gds-init gds λ-. True gen-cond gen-step]
  and this[unfolded gen-dfsT-def WHILET-def]
  have WF: wf (?V⁻¹) by simp

    from NF have NF': nofail gen-dfs using gen-dfs-le-gen-dfsT
      by (auto simp: pw-le-iff)

    from rwof-rel-spec[of gds-init gds gen-cond gen-step] have
      \( \forall s. \left[ \langle \text{gen-rwof } s; \text{gen-cond } s \rangle \right] \Rightarrow \text{gen-step } s \leq_n \text{SPEC } (\lambda s'. (s,s') \in ?V) \).
    hence aux: \( \forall s. \left[ \langle \text{gen-rwof } s; \text{gen-cond } s \rangle \right] \Rightarrow \text{gen-step } s \leq_n \text{SPEC } (\lambda s'. (s,s') \in ?V) \)
      apply (rule leofD[rotated])
      apply assumption
      apply assumption
      using NF[unfolded gen-dfsT-def]
    by (drule (1) WHILEET-nofail-imp-rwof-nofail)

show theorem
  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-rec bind-refine inj-on-id)
  apply refine-dref-type
  apply simp-all
  apply (subst WHILEET-eq-WHILEI-tproof[where V=?V⁻¹])

89
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
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 ≡ \{ 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'
end
**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)
        )
        (λ D (u, s)). do {
          s ← FOREACHci
          (λit s′. gen-rwof′ s′
            ∧ (¬gds-is-break gds s′ →
                stack s′ = stack s
                ∧ pending s′ = (pending s − {u}×(E′(u) − it))
                ∧ gen-discovered s′ ≥ gen-discovered s ∪ (E′(u) − it))
            )
          (E′(u)) (λs. ¬gds-is-break gds s)
        (λv s. do {
          s ← choose-pending u (Some v) s;
          if gds-is-discovered gds v s then do {
            if gds-is-finished gds v s then
              gds-cross-edge gds u v s
            else
              gds-back-edge gds u v s
          } else do {
            s ← gds-discover gds u v s;
          }
        } else {
          s ← gds-discover gds u v s;
        })
    })
  })

91
if gds-is-break gds s then RETURN s else D (v,s)
}
}}
s;
if gds-is-break gds s then
 RETURN s
else do {
 s ← choose-pending u (None) s;
 s ← gds-finish gds u s;
 RETURN s
}
}) (v0,s)
}
}
s
}) s
}
}
definition rec-impl-for-paper where rec-impl-for-paper ≡ do {

 s ← gds-init gds;
 FOREACHc V0 (Not o gds-is-break gds) (\v0 s. do {
 if gds-is-discovered gds v0 s then RETURN s
 else do {
 s ← gds-new-root gds v0 s;
 if gds-is-break gds s then RETURN s
 else do {
 REC (\D (\u,s). do {
 s ← FOREACHc (E''\{u\}) (\ls. ¬gds-is-break gds s) (\v s. do {
 s ← choose-pending u (Some v) s;
 if gds-is-discovered gds v s then do {
 if gds-is-finished gds v s then gds-cross-edge gds u v s
 else gds-back-edge gds u v s
 } else do {
 s ← gds-discover gds u v s;
 if gds-is-break gds s then RETURN s else D (v,s)
 }
}));
if gds-is-break gds s then RETURN s
else do {
 s ← choose-pending u (None) s;
 gds-finish gds u s
}
}) (v0,s)
}
}) s
}
end
locale rec-impl =

fb-graph $G + \text{gen-dfs} \ gds \ V0 + \text{rec-impl-defs} \ G \ \text{pending} \ \text{stack} \ \text{choose-pending} \n$

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

and $gds :: (v, 's) \ \text{gen-dfs-struct}$

and $pending :: 's \Rightarrow \ 'v \ \text{rel}$

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

and $\text{choose-pending} :: 'v \Rightarrow \ 'v \ \text{option} \Rightarrow \ 's \ \text{res}$

+ assumes [simp]: $\text{gds-is-empty-stack} \ gds \ s \leftarrow stack \ s = []$

assumes init-spec:

$\text{gds-init} \ gds \ SPEC (\lambda s. stack \ s = [] \land pending \ s = {})$

assumes new-root-spec:

$\left[\left[\text{pre-new-root} \ v0 \ s\right]\right] \Rightarrow \text{gds-new-root} \ gds \ v0 \ s \ SPEC (\lambda s'.$

$\text{stack} \ s' = [v0] \land pending \ s' = \{\text{v0}\} \times E''{\{\text{v0}\}} \land$

$\text{gen-discovered} \ s' = \text{insert} \ v0 \ (\text{gen-discovered} \ s)$

assumes get-pending-fmt: $\left[\left[\text{pre-get-pending} \ s\right]\right] \Rightarrow$

\[\begin{array}{l}
\text{do} \\
\quad \text{let} \ u = \text{hd} (\text{stack} \ s); \\
\quad vo \leftarrow \text{SELECT} (\lambda v. (u,v) \in \text{pending} \ s); \\
\quad s \leftarrow \text{choose-pending} \ u \ vo \ s; \\
\quad \text{RETURN} \ (u,vo,s)
\end{array}\]

\[\leq \text{gds-get-pending} \ gds \ s\]

assumes choose-pending-spec: $\left[\left[\text{pre-get-pending} \ s; \ u = \text{hd} \ (\text{stack} \ s); \ \text{case} \ vo \ \text{of} \right]\right]$

$\begin{cases}
\text{None} \Rightarrow \text{pending} \ s '' \{u\} = {} & \\
\text{| Some} \ v \Rightarrow v \in \text{pending} \ s '' \{u\}
\end{cases}$

$\Rightarrow$

\[\text{choose-pending} \ u \ vo \ s \ SPEC (\lambda s'.$

$\text{(case} \ vo \ \text{of} \$

$\text{| None} \Rightarrow \text{pending} \ s' = \text{pending} \ s$

$\text{| Some} \ v \Rightarrow \text{pending} \ s' = \text{pending} \ s - \{(u,v)\} \land$

$\text{stack} \ s' = \text{stack} \ s \land$

$(\forall x. \text{gds-is-discovered} \ gds \ x \ s' = \text{gds-is-discovered} \ gds \ x \ s)$

$\end{cases}$

$\text{finish-spec: } \left[\left[\text{pre-finish} \ u \ s0 \ s\right]\right]$

$\Rightarrow \text{gds-finish} \ gds \ u \ s \ SPEC (\lambda s'.$

$\text{pending} \ s' = \text{pending} \ s \land$

$\text{stack} \ s' = \text{tl} (\text{stack} \ s) \land$

$(\forall x. \text{gds-is-discovered} \ gds \ x \ s' = \text{gds-is-discovered} \ gds \ x \ s))$

assumes cross-edge-spec: $\text{pre-cross-edge} \ u \ v \ s0 \ s$$\Rightarrow \text{gds-cross-edge} \ gds \ u \ v \ s \ SPEC (\lambda s'$. pending $s' = \text{pending} \ s \land \text{stack} \ s' = \text{stack} \ s \land$

93
(∀ x. gds-is-discovered gds x s' = gds-is-discovered gds x s))

assumes back-edge-spec: pre-back-edge u v s0 s
⇒ gds-back-edge gds u v s ≤_n SPEC (λs'. pending s' = pending s ∧ stack s' = stack s ∧
(∀ x. gds-is-discovered gds x s' = gds-is-discovered gds x s))

assumes discover-spec: pre-discover u v s0 s
⇒ gds-discover gds u v s ≤_n SPEC (λs'. pending s' = pending s ∪ (v × E'\{v\}) ∧ stack s' = v#stack s ∧
gen-discovered s' = insert v (gen-discovered s))

begin

lemma gen-step'-refine:
[ gen-rwof s; gen-cond s ] ⇒ gen-step' s ≤ gen-step s
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' ≤ gen-dfs
unfolding gen-dfs'-def gen-dfs-def WHILE-eq-I-rwof[where f=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:
gen-rwof' s ⇒ set (stack s) ⊆ reachable ∧ pending s ⊆ E
∧ set (stack s) ⊆ gen-discovered s ∧ distinct (stack s)
∧ pending s ⊆ set (stack s) × UNIV
apply (erule establish-rwof-invar [rotated – l])
apply (rule leof-trans [OF init-spec], auto)
apply (subst gen-step′-def)
apply (refine-reg refine-rcg refine-vcg
  leaf-trans [OF new-root-spec]
  SELECT-rule [THEN leaf-lift]
  leaf-trans [OF choose-pending-spec [THEN leaf-strengthen-SPEC]]
  leaf-trans [OF finish-spec]
  leaf-trans [OF cross-edge-spec]
  leaf-trans [OF back-edge-spec]
  leaf-trans [OF discover-spec])

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

apply ((unfold pre-defs, intro conjI); assumption?)
subgoal by (clarsimp simp: gen-cond-def)
subgoal by (clarsimp simp: gen-cond-def)
subgoal
  apply (rule pwD2 [OF get-pending-fmt!])
  subgoal by (clarsimp simp: 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)

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

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

lemma mk-spec-aux:
[\[ m \leq n \text{ SPEC } \Phi; m \leq \text{SPEC gen-rwof'} \] ] \rightarrow m \leq \text{SPEC} (\lambda s. \text{gen-rwof'} s \wedge \Phi s)
by (rule SPEC-rule-conj-leofII)

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

96
None ⇒ pending s0''{u} = {} ∧ pending s = pending s0
| Some v ⇒ v ∈ pending s0''{u} ∧ pending s = pending s0 − {(u,v)}

context
assumes nofail:
  nofail (gds-init gds ⇒ WHILE gen-cond gen-step')
assumes nofail2:
  nofail (gen-dfs)
begin

lemma pcp-imp-pgp:
post-choose-pending u vo s0 s0 s ⇒ 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:
[ [pre-new-root v0 s; gen-rwof'' s] ⇒ gds-new-root gds v0 s ≤ SPEC ?Φ
apply (rule mk-spec-aux[OF new-root-spec], assumption)
apply (rule order-trans[OF - rwof-step[OF nofail, where s=s]], assumption)
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 ⇒ pending s ++ {u} = {}
  | Some v ⇒ v ∈ pending s ++ {u}
shows choose-pending u vo s ≤ SPEC (post-choose-pending u vo s)
proof –
from WHILE-nofail-imp-rwof-nofail[OF nofail 2] 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 \( \text{choose-pending } u \; \text{vo } s \leq_n \text{SPEC} \) (post-choose-pending \( u \; \text{vo } s \))

apply (rule leof-trans[OF choose-pending-spec[OF 1 - 3, THEN leof-strengthen-SPEC]])
  apply simp
  apply (rule leof-RES-rule)
  using 1
  apply (simp add: post-choose-pending-def 2 pre-defs gen-cond-def split: option.splits)
  using 3
  apply auto
  done
finally (leofD) show \(?\text{thesis}\).

qed

schematic-goal gds-finish-refine:
[[\text{pre-finish } u \; s_0 \; s; \text{post-choose-pending } u \; \text{None } s_0 \; s] \implies \text{gds-finish } gds \; u \; s \leq SPEC \?\Phi

apply (rule mk-spec-aux[OF finish-spec], assumption)
apply (rule order-trans[OF - rwof-step[OF nofail, where \( s=s_0\)])]
unfolding gen-step'-def pre-defs gen-cond-def post-choose-pending-def
apply (auto simp: pw-le-iff refine-pw-simps split: option.split)
done

schematic-goal gds-cross-edge-refine:
[[\text{pre-cross-edge } u \; v \; s_0 \; s; \text{post-choose-pending } u \; (\text{Some } v) \; s_0 \; s] \implies \text{gds-cross-edge}
\text{gds } u \; v \; s \leq SPEC \?\Phi

apply (rule mk-spec-aux[OF cross-edge-spec], assumption)
apply (rule order-trans[OF - rwof-step[OF nofail, where \( s=s_0\)])]
unfolding gen-step'-def pre-defs gen-cond-def post-choose-pending-def
apply (auto simp: pw-le-iff refine-pw-simps select-def split: option.split, blast)
apply simp
apply blast
done

schematic-goal gds-back-edge-refine:
[[\text{pre-back-edge } u \; v \; s_0 \; s; \text{post-choose-pending } u \; (\text{Some } v) \; s_0 \; s] \implies \text{gds-back-edge}
\text{gds } u \; v \; s \leq SPEC \?\Phi

apply (rule mk-spec-aux[OF back-edge-spec], assumption)
apply (rule order-trans[OF - rwof-step[OF nofail, where \( s=s_0\)])]
unfolding gen-step'-def pre-defs gen-cond-def post-choose-pending-def
apply (simp add: pw-le-iff refine-pw-simps select-def split: option.split, blast)
apply simp
apply blast
done

schematic-goal gds-discover-refine:
[[\text{pre-discover } u \; v \; s_0 \; s; \text{post-choose-pending } u \; (\text{Some } v) \; s_0 \; s] \implies \text{gds-discover}
\text{gds } u \; v \; s \leq SPEC \?\Phi

apply (rule mk-spec-aux[OF discover-spec], assumption)
apply (rule order-trans[OF - rwof-step[OF nofail, where \( s=s_0\)])]

98
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
done
end

lemma rec-impl-aux: \[ xd \notin \text{Domain} \ P \ \Rightarrow \ P - \{y\} \times (\text{succ} \ y - \text{ita}) - \{(y, xd)\} - \{xd\} \times \text{UNIV} = \]
\[ P - \text{insert} (y, xd) (\{y\} \times (\text{succ} \ y - \text{ita})) \]
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-r cg refine-rv)
order-trans[OF gds-init-refine]
order-trans[OF gds-choose-pending-refine]
order-trans[OF gds-new-root-refine]
order-trans[OF gds-finish-refine]
order-trans[OF gds-back-edge-refine]
order-trans[OF gds-cross-edge-refine]
order-trans[OF gds-discover-refine]
)
apply (simp-all split: if-split-asm)
using [[goals-limit = 1]]
apply (auto simp add: pre-defs; fail)
apply (auto simp add: pre-defs gen-rwof′-imp-rwof; fail)
apply (auto; fail)
apply (auto dest: reachable-invar; fail)
apply (auto simp add: pre-defs gen-rwof′-imp-rwof; fail)
apply (auto; fail)
apply (auto; fail)
apply ((drule pcp-imp-pgp, auto simp add: pre-defs gen-rwof′-imp-rwof); fail)
apply (auto simp: post-choose-pending-def; fail)
apply (auto simp: post-choose-pending-def; fail)
apply (auto simp: post-choose-pending-def; fail)
apply ((drule pcp-imp-pgp, auto simp 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-conv)
  apply (rule conjI)
subgoal
  apply (rule rec-impl-aux)
  apply (drule reachable-invar)+
  apply (metis Domain.cases SigmaD1 mem-Collect-eq rev-subsetD)
done
subgoal
  apply (rule conjI)
  apply auto []
  apply (metis order-trans)
done
done
apply (auto simp add: pre-defs gen-rwof'-imp-rwof; fail)
apply (auto; fail)
apply (auto dest: reachable-invar; fail)
apply ((drule pcp-imp-pgp, auto simp add: pre-defs gen-rwof'-imp-rwof); fail)
apply (auto simp: post-choose-pending-def; fail)
apply (auto simp: post-choose-pending-def; fail)
apply (auto simp: post-choose-pending-def; fail)
apply (auto; fail)
apply (auto simp: gen-cond-def; fail)
apply (auto simp: gen-cond-def; fail)
done
end
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' " {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 ([] 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

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 ≡ \text{return} (| ss-stack = [], on-stack = {}, visited = {}, \ldots = 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 ≡ \{ \text{assert} (ss-stack s \neq [] \land u \in on-stack s); \text{let } s = s([\text{ss-stack} := \text{tl} (ss-stack s)]); \text{let } s = s([\text{on-stack} := \text{on-stack s} - \{u\}]); \text{return } s \}

definition get-pending-impl s ≡ \{ \text{assert} (ss-stack s \neq []); \text{let } (u,Vs) = \text{hd} (ss-stack s); \text{if } Vs = \{\} \text{ then } \text{return } (u,\text{None}, s) \text{ else do } \{ \text{let } v = \text{spec} (\lambda v. \text{v} \in Vs); \text{let } Vs = Vs - \{v\}; \text{let } s = s([\text{ss-stack} := (u,Vs) \# \text{tl} (ss-stack s) \]); \text{return } (u, \text{Some } v, s) \} \}

definition discover-impl u v s ≡ \{ \text{assert} (v \notin on-stack s \land v \in visited s); \text{let } s = s([ss-stack := (v,E''\{v\}) \# ss-stack s]); \text{let } s = s([on-stack := insert v (on-stack s)]); \text{let } s = s([visited := insert v (visited s)]); \text{return } s \}

definition new-root-impl v0 s ≡ \{ \text{assert} (v0 \notin visited s); \text{let } s = s([ss-stack := [(v0,E''\{v0\})]]); \text{let } s = s([on-stack := \{v0\}]); \text{let } s = s([visited := insert v0 (visited s)]); \text{return } s \}
definition \( \text{gbs} \equiv \{ \) 
\( \text{gbs-init} = \text{init-impl} , \)
\( \text{gbs-is-empty-stack} = \text{is-empty-stack-impl} , \)
\( \text{gbs-new-root} = \text{new-root-impl} , \)
\( \text{gbs-get-pending} = \text{get-pending-impl} , \)
\( \text{gbs-finish} = \text{finish-impl} , \)
\( \text{gbs-is-discovered} = \text{is-discovered-impl} , \)
\( \text{gbs-is-finished} = \text{is-finished-impl} , \)
\( \text{gbs-back-edge} = (\lambda u v s. \text{RETURN } s) , \)
\( \text{gbs-cross-edge} = (\lambda u v s. \text{RETURN } s) , \)
\( \text{gbs-discover} = \text{discover-impl} \)\( \} \)

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

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

end

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

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

begin

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

definition \( \text{get-pending } s \equiv \bigcup (\{ \text{map } (\lambda (u, Vs). \{u\} \times Vs) (\text{ss-stack } s) \})\)
definition \( \text{get-stack } s \equiv \text{map } \text{fst} (\text{ss-stack } s) \)
definition \( \text{choose-pending} :: \text{'}v \Rightarrow \text{'}v \text{ option } \Rightarrow (\text{'}v,\text{'d})\text{ simple-state-scheme } \Rightarrow (\text{'}v,\text{'d})\text{ simple-state-scheme} \)
nres
\( \text{where } [\text{DFS-code-unfold}]: \)
\( \text{choose-pending } u v s \equiv \)
case vo of
    None ⇒ RETURN s
    | Some v ⇒ do
        ASSERT (ss-stack s ≠ []);
        let (u,Vs) = hd (ss-stack s);
        RETURN (s\ [v]\ ss-stack := (u, Vs - {v}) # tl (ss-stack s))
    }

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

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

lemma init-impl: (ei, e) ∈ ES ⇒
    c.init-impl ei ≤\(\downarrow\)⟨⟨ES⟩simple-state-rel⟩ (RETURN (a.empty-state e))
unfolding c.init-impl-def a.empty-state-def simple-state-rel-def
by (auto)

lemma new-root-impl:
[a.gen-dfs.pre-new-root v0 s;
  (v0i, v0) ∈ Id; (si, s) ∈ ⟨ES⟩simple-state-rel]
⇒ c.new-root-impl v0 si ≤\(\downarrow\)⟨⟨ES⟩simple-state-rel⟩ (RETURN (a.new-root v0 s))
unfolding simple-state-rel-def a.gen-dfs.pre-new-root-def c.new-root-impl-def
by (auto simp add: a.pred-defs)

lemma get-pending-impl:
[a.gen-dfs.pre-get-pending s; (si, s) ∈ ⟨ES⟩simple-state-rel]
⇒ c.get-pending-impl si
  ≤\(\downarrow\) (Id ×, Id ×, ⟨ES⟩simple-state-rel) (a.get-pending s)
apply (unfold a.get-pending-def c.get-pending-impl-def) []
apply (refine-reg bind-refine’ Let-refine’ IdI)
apply (refine-dref-type)
apply (auto
  simp: simple-state-rel-def a.gen-dfs.pre-defs a.pred-defs neq-Nil-conv
  dest: DFS-invar.stack-distinct
done

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

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

lemma finish-impl:
  [\[ a.gen-dfs.pre-finish v s0 s; (vi, v)\in Id; (si, s) \in \langle ES \rangle simple-state-rel \] \]
  ⇒ c.finish-impl v si \leq⇓ (\langle ES \rangle 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.map-tl)
apply (clarsimp simp: neq-Nil-conv)
apply blast
done

lemma cross-edge-impl:
  [\[ a.gen-dfs.pre-cross-edge u v s0 s; (ui, u)\in Id; (vi, v)\in Id; (si, s) \in \langle ES \rangle simple-state-rel \] \]
  ⇒ (si, a.cross-edge u v s) \in \langle ES \rangle simple-state-rel
unfolding simple-state-rel-def a.gen-dfs.pre-defs
by simp

lemma back-edge-impl:
  [\[ a.gen-dfs.pre-back-edge u v s0 s; (ui, u)\in Id; (vi, v)\in Id; (si, s) \in \langle ES \rangle simple-state-rel \] \]
  ⇒ (si, a.back-edge u v s) \in \langle ES \rangle simple-state-rel
unfolding simple-state-rel-def a.gen-dfs.pre-defs
by simp

lemma discover-impl:
  [\[ a.gen-dfs.pre-discover u v s0 s; (ui, u)\in Id; (vi, v)\in Id; (si, s) \in \langle ES \rangle simple-state-rel \] \]
  ⇒ c.discover-impl ui vi si \leq⇓ (\langle ES \rangle simple-state-rel) (RETURN (a.discover u v s))
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.gen-dfs ≤ ⪯⟨⟨ES⟩simple-state-rel⟩ a.it-dfs
using gen-dfs-refine
by simp

lemma simple-refineT[refine]: c.gen-dfsT \leq \psi((ES)\ simple-state-rel) a.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]: tailrec-impl \leq \psi((ES)\ simple-state-rel) a.it-dfs
proof -
  note tailrec-impl also note simple-refine finally show \?thesis .
qed

lemma simple-tailrecT-refine[refine]: tailrec-implT \leq \psi((ES)\ simple-state-rel) a.it-dfsT
proof -
  note tailrecT-impl also note simple-refineT finally show \?thesis .
qed

Link to recursive implementation

lemma reachable-invar:
  assumes c.gen-rwof s
  shows set (map fst (ss-stack s)) \subseteq visited s
  \land 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-rec refine-vcg)
apply simp

apply (refine-rec refine-vcg)
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

sublocale rec-impl G c.gds get-pending get-stack choose-pending
apply unfold-locales
unfolding get-pending-def get-stack-def choose-pending-def
apply (simp-all add: c.do-action-defs c.impl-defs[abs-def])
apply (auto simp: pw-le-of-iff refine-pw-simps pw-le-iff select-def
split: prod.split) []
apply (auto simp: pw-le-of-iff refine-pw-simps pw-le-iff
split: prod.split) []
apply (rule le-ASSERTI)
apply (unfold c.pre-defs, clarify) []
apply (frule reachable-invar)
apply (fastforce simp add: pw-le-of-iff refine-pw-simps pw-le-iff
split: prod.split option.split) []
apply (auto simp: pw-le-of-iff refine-pw-simps pw-le-iff
split: prod.split if-split-as) []
apply (auto simp: pw-le-of-iff refine-pw-simps pw-le-iff split: prod.split) []
apply (auto simp: pw-le-of-iff refine-pw-simps pw-le-iff split: prod.split) []
done

lemma simple-rec-refine[refine]: rec-impl ≤ \(\langle ES \rangle\) 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, \ldots = mi),
| (ss-stack = s, on-stack = os, visited = vis, \ldots = m)|
si osi visi mi s os vis m,
| (si, s) ∈ s-rel ∧
| (osi, os) ∈ vis-rel ∧
(visi, vis) ∈ vis-rel ∧ (mi, m) ∈ erel
\}

consts
i-simple-state :: interface ⇒ interface ⇒ interface ⇒ interface

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

term simple-state-ext

lemma [autoref-rules, param]:
 fixes s-rel ps-rel vis-rel erel
 defines R ≡ (s-rel, vis-rel, erel) ss-impl-rel
 shows (ss-stack-impl, ss-stack) ∈ R → s-rel
 (ss-on-stack-impl, on-stack) ∈ R → vis-rel
 (ss-visited-impl, visited) ∈ R → vis-rel
 (simple-state-impl.more, simple-state.more) ∈ R → erel
 (ss-stack-impl-update, ss-stack-update) ∈ (s-rel → s-rel) → R → R
 (ss-on-stack-impl-update, on-stack-update) ∈ (vis-rel → vis-rel) → R → R
 (ss-visited-impl-update, visited-update) ∈ (vis-rel → vis-rel) → R → R
 (simple-state-impl-update, simple-state-update) ∈ (erel → erel) → R → R
 (simple-state-impl-ext, simple-state-ext) ∈ s-rel → vis-rel → vis-rel → erel → R
 unfolding ss-impl-rel-def R-def
 apply auto
 apply parametricity+
done

1.7.2 Simple state without on-stack

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

record ('si,'ksi)simple-state-nos-impl =
 ssnos-stack-impl :: 'si
 ssnos-visited-impl :: 'ksi

definition [to-relAPP]: ssnos-impl-rel s-rel vis-rel erel ≡
 { (ss-stack-impl = si, ssnos-visited-impl = visi, ... = mi),
   (ss-stack = s, on-stack = os, visited = vis, ... = m)) |
   si visi mi s os vis m.
   (si, s) ∈ s-rel ∧
   (visi, vis) ∈ vis-rel ∧
   (mi, m) ∈ erel
 }

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

definition op-nos-on-stack-update
\[ (- \text{ set} \Rightarrow - \text{ set}) \Rightarrow (-, -)\text{simple-state-scheme} \Rightarrow - \]

where \( \text{op-nos-on-stack-update} \equiv \text{on-stack-update} \)

context begin interpretation autoref-syn .

\textbf{lemma} \[ \text{op-nos-on-stack-update} f \ s \equiv \text{OP} (\text{op-nos-on-stack-update} f) \$ s \text{ by simp} \]

end

lemmas ssnos-unfolds — To be unfolded before autoref when using \( \text{ssnos-impl-rel} \)

\[ \equiv \text{op-nos-on-stack-update-def} \]

\[ \equiv \text{symmetric} \]

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

\textbf{fixes} \( s\text{-rel} \ vis\text{-rel} \ erel \)

\textbf{defines} \( R \equiv \langle s\text{-rel}, vis\text{-rel}, erel \rangle \text{ssnos-impl-rel} \)

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

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

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

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

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

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

\( (\lambda ns - ps \ vs. \text{simple-state-nos-impl-ext} ns ps vs, \text{simple-state-ext}) \in s\text{-rel} \rightarrow \text{ANY-rel} \rightarrow \text{vis-rel} \rightarrow \text{erel} \rightarrow R \)

\textbf{unfolding} \( \text{ssnos-impl-rel-def} R\text{-def op-nos-on-stack-update-def} \)

\textbf{apply} auto

\textbf{apply} parametricity+

\textbf{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)!

\textbf{record} \( ('si, 'nsi)\text{simple-state-ns-impl} = \)

\( \text{ssns-visited-impl} :: 'nsi \)

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

\( \{ ((\text{ssns-visited-impl} = \text{visi, ... = mi}), \)

\( (\text{ss-stack} = s, \text{on-stack} = os, \text{visited} = \text{vis}, \ldots = m)) \mid \)

\( \text{visi mi s os vis m}, \)

\( (\text{visi, vis}) \in \text{vis-rel} \land \)

\( (mi, m) \in \text{erel} \}

\textbf{lemmas} [autoref-rel-intf] = \text{REL-INTFI[of ssns-impl-rel i-simple-state]}
\textbf{definition} \textit{op-ns-on-stack-update}\\:: (- set \Rightarrow - set) \Rightarrow (-,-)\text{simple-state-scheme} \Rightarrow -\\where \textit{op-ns-on-stack-update} \equiv \text{on-stack-update}\\

\textbf{definition} \textit{op-ns-stack-update}\\:: (- list \Rightarrow - list) \Rightarrow (-,-)\text{simple-state-scheme} \Rightarrow -\\where \textit{op-ns-stack-update} \equiv \text{ss-stack-update}\\

\textbf{context} begin \textit{interpretation autoref-sgn} .\\\textbf{lemma} [\textit{autoref-op-pat-def}]: \textit{op-ns-on-stack-update} f s \equiv \text{OP} (\textit{op-ns-on-stack-update} f)\$s \textbf{by simp}\\

\textbf{lemma} [\textit{autoref-op-pat-def}]: \textit{op-ns-stack-update} f s \equiv \text{OP} (\textit{op-ns-stack-update} f)\$s \textbf{by simp}\\

end\\

\textbf{context} \textit{simple-impl-defs} begin\\\textbf{thm} \textit{choose-pending-def[unfolded \textit{op-ns-stack-update-def[symmetric]}, no-vars]}\\\textbf{lemma} \textit{choose-pending-ns-unfold}: \textit{choose-pending} u vo s = (\\\textit{case} vo of \textit{None} \Rightarrow \text{RETURN} s\\\mid \text{Some} v \Rightarrow \text{do} \{\\\hspace{1em} - \leftarrow \text{ASSERT} (\text{ss-stack} s \neq []);\\\hspace{2em} \text{RETURN} (\textit{op-ns-stack-update}\\\hspace{3em} (\textit{let} (u, Vs) = \text{hd} (\text{ss-stack} s)\\\hspace{4em} \text{in} (\lambda-. (u, Vs - \{v\}) \neq \text{tl} (\text{ss-stack} s))\\\hspace{3em}) s)\\\}\}\textbf{unfolding} \textit{choose-pending-def \textit{op-ns-stack-update-def}}\\\textbf{by} (\textit{auto split: option.split prod.split})\\\begin{itemize}\\item \textbf{lemmas} \textit{ssns-unfolds} — To be unfolded before \textit{autoref} when using \textit{ssns-impl-rel}.\\\textbf{Attention}: This lemma conflicts with the standard unfolding lemma in \textit{DFS-code-unfold}, so has to be placed first in an \textit{unfold-statement}!\\\hspace{1em} = \textit{op-ns-on-stack-update-def[symmetric]} \textit{op-ns-stack-update-def[symmetric]}\\\hspace{1em} \textit{choose-pending-ns-unfold}\\\end{itemize}\\

end\\

\textbf{lemma} [\textit{autoref-rules, param}]:\\\textbf{fixes} \textit{s}-rel \textit{vis-rel erel ANY-rel}\\\textbf{defines} \textit{R} \equiv (\textit{ANY-rel,vis-rel,ereleq})\textit{ssns-impl-rel}\\\textbf{shows}
(ssns-visited-impl, visited) ∈ R → vis-rel
(simple-state-ns-impl.more, simple-state.more) ∈ R → erel
\[ (∀ x. \text{op-ns-stack-update } f) \in R → R \]
\[ (∀ x. \text{op-ns-on-stack-update } f) \in R → R \]
(ssns-visited-impl-update, visited-update) ∈ (vis-rel → vis-rel) → R → R
(simple-state-ns-impl.more-update, simple-state.more-update) ∈ (erel → erel) → R → R

(λ- ps vs, simple-state-ns-impl-ext ps vs, simple-state-ext)
∈ ANY1-rel → ANY2-rel → vis-rel → erel → R

unfolding ssns-impl-rel-def R-def op-ns-on-stack-update-def op-ns-stack-update-def
apply auto
apply parametricity+
by auto

end

1.8 Restricting Nodes by Pre-Initializing Visited Set

theory Restr-Impl
imports Simple-Impl
begin
Implementation of node and edge restriction via pre-initialized visited set.
We now further refine the simple implementation in case that the graph has the form
\( G'=(rel-restrict E R, V0-R) \) for some fb-graph \( G=(E,V0) \). If, additionally, the parameterization is not "too sensitive" to the visited set, we can pre-initialize the visited set with \( R \), and use the \( V0 \) and \( E \) of \( G \). This may be a more efficient implementation than 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 ≡ \{ (s,s') \}
\((ss-stack s, ss-stack s')\in(Id ×r \{(U,U') \}. U-R = U')\)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 \((s,s')\in restr-rel R\)
 shows \(visited s = visited s' \cup R\)
 and \(simple-state.more s = simple-state.more s'\)
 using assms unfolding restr-rel-def by auto

lemma
 assumes \((s,s')\in restr-rel R\)
 shows \(restr-rel-stackD: (ss-stack s, ss-stack s') \in (Id \times_r \{(U,U'). U-R = U'\})\) \text{list-rel}\)
 and \(restr-rel-vis-djD: visited s' \cap R = \{\}\)
 using assms unfolding restr-rel-def by auto

category fixes \(R :: 'v set\) begin

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

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

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

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

113
definition gbs' ≡ gbs ∘
  gbs-init := \lambda e. RETURN
  ( ss-stack= [], on-stack= {}, visited = R, \ldots e [] )

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

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

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

assumes is-break-refine:
  [ (s,s')\in restr-rel R ]
  \Rightarrow is-break parami s \leftrightarrow is-break parami s'

assumes on-new-root-refine:
  [ (s,s')\in restr-rel R ]
  \Rightarrow on-new-root parami v0 s \leq on-new-root parami v0 s'

assumes on-finish-refine:
  [ (s,s')\in restr-rel R ]
  \Rightarrow on-finish parami u s \leq on-finish parami u s'

assumes on-discover-refine:
  [ (s,s')\in restr-rel R ]
  \Rightarrow on-discover parami u v s \leq 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)\in rel \Rightarrow 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 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)∈Id; (si,s)∈rel; v0∉R ]
⇒ do-new-root v0i si ≤ ⇓ rel (a.c.do-new-root v0 s)
  unfolding do-action-defs a.c.do-action-defs
  apply refine-rcg

115
apply (rule intro-prgR[where R=rel])
apply (simp add: a.c.new-root-impl-def new-root-impl-def)
apply (refine-reg, auto simp: rel-def rel-restrict-def) []
apply (rule intro-prgR[where R=Id])
apply (simp add: on-new-root-refine)
apply (simp add: rel-def)
done

lemma do-finish-refine[refine]:
\[(s, s') \in rel; \ (u,u')\in Id\] 
\[\implies do-finish u s \leq \downarrow rel \ (a.c.do-finish u' s')\]
unfolding do-action-defs a.c.do-action-defs
apply refine-reg
apply (rule intro-prgR[where R=rel])
apply (simp add: finish-impl-def is-empty-stack-impl-def)
apply (refine-reg, auto simp: rel-def rel-restrict-def) []
apply parametricity
apply (rule intro-prgR[where R=Id])
apply (simp add: on-finish-refine)
apply (simp add: rel-def)
done

lemma aux-cnv-pending: 
\[(s, s') \in rel; \ \neg is-empty-stack-impl s; \ vs\in Vs; \ vs\notin R; \]
\[hd (ss-stack s) = (u, Vs) \implies hd (ss-stack s') = (u, insert vs (Vs\setminus R))\]
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 \((s, s') \in rel \ gen-cond s \ \neg is-empty-stack-impl s\)
shows \(get-pending-impl s \leq (sup \ (\downarrow (Id \times_r (Id) \ option-rel \times_r rel) \ (inf \ (get-pending-impl s'))) \ (SPEC \ (\lambda (\cdot, Vs, \cdot). \ case Vs \ of \ None \Rightarrow True \ | \ Some v \Rightarrow v \notin R))))\)
\((\downarrow (Id \times_r (Id) \ option-rel \times_r rel) \ (\ SPEC \ (\lambda (u, Vs, s'). \ \exists v. \ Vs=Some v \land v \in R \land s''=s'))\))
proof -
from assms have \[simp]: ss-stack s' \neq []\]
and \([\text{simp}]: \text{ss-stack } s \neq []\)
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 (split prod.split, intro allI impI)
apply (rule lhs-step-If)

apply (rule le-supI1)
apply (simp add: pred-rel-simps no-pending-refine restr-rel-simps
RETURN-RES-refine-iff)

apply (rule rhs-step-bind, simp)
apply (simp split del: if-split)
apply (rename-tac v)
apply (case-tac v ∈ R)

apply (rule le-supI2)
apply (rule RETURN-SPEC-refine)
apply (auto simp: rel-def is-empty-stack-impl-def neq-Nil-conv) []
apply (cases ss-stack s′, simp) apply (auto elim: list-relE) []

apply (rule le-supI1)
apply (frule (4) aux-cnv-pending)
apply (simp add: no-pending-refine pred-rel-simps memb-imp-not-empty)
apply (subst nofail-inf-serialize,
(simp-all add: refine-pw-simps split: prod.splits) [2])
apply simp
apply (rule rhs-step-bind-RES, blast)
apply (simp add: rel-def is-empty-stack-impl-def) []
apply (cases ss-stack s′, simp)
apply (auto elim: list-relE) []
done

qed

lemma do-discover-refine[refine]:
\[
\begin{array}{l}
(s, s') \in \text{rel}; (u, u') \in \text{Id}; (v, v') \in \text{Id}; v' \notin R
\end{array}
\]

\(\implies\) do-discover \(u\ v\ s\ \leq_{R} \) rel \(\langle a.\ c.\ \text{do-discover} \ a' \ v' \ s' \rangle\)

unfolding do-action-defs a.c.do-action-defs
apply refine-rvc
apply (rule intro-prgR[where \(R=\text{rel}\)])
apply (simp add: discover-impl-def a.c.discover-impl-def)
apply (refine-rcg,auto simp: rel-def rel-restrict-def)

apply (rule intro-prgR[where \( R = \text{Id} \)])
apply (simp add: on-discover-refine)

apply (auto simp: rel-def)
done

lemma aux-R-node-discovered: \( [((s,s') \in \text{rel}; v \in R] \implies \text{is-discovered-impl } v \text{ s} \)
by (auto simp: pred-rel-simps)

lemma re-refine-aux: gen-dfs \( \leq \downarrow \text{rel a.c.gen-dfs} \)
unfolding a.c.gen-dfs-def gen-dfs-def
apply (simp del: graph-restrict-simps)
apply (rule bind-refine)
apply (refine-rcg)
apply (erule WHILE-invisible-refine)

apply (frule gen-cond-param[param-fo], fastforce)

apply (frule (1) gen-cond-param[param-fo, THEN IdD, THEN iffD1])
apply (simp del: graph-restrict-simps)
unfolding gen-step-def
apply (simp del: graph-restrict-simps cong: if-cong cong: option.cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong cong con-
tion.case-cong if-cong)
  apply (rule bind-refine'[OF conc-fan-mono[THEN monoD]], simp)
  apply (clarsimp simp: refine-pw-simps)
  apply (refine-rcg, refine-dref-type, simp-all add: pred-rel-simps) []

  apply (rule le-supI2)
  apply (rule RETURN-as-SPEC-refine)
  apply (clarsimp simp add: conc-fan-SPEC)
  apply (refine-rcg refine-vcg bind-refine', simp-all) []
  apply (clarsimp)
  apply (frule (1) aux-R-node-discovered, blast)
  done

theorem re-refine-aux2: gen-dfs ≤⇓(rel O (ES) simple-state-rel) a.a.it-dfs
proof –
  note re-refine-aux
  also note a.gen-dfs-refine
  finally show ?thesis by (simp add: conc-fan-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 (simp-all 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

end

1.9 Basic DFS Framework

theory DFS-Framework
imports
  Param-DFS
  Invars/DFS-Invars-Basic

119
Impl/Structural/Tailrec-Impl
Impl/Structural/Rec-Impl
Impl/Data/Simple-Impl
Impl/Data/Restr-Impl

\textbf{begin}

Entry point for the DFS framework, with basic invariants, tail-recursive and recursive implementation, and basic state data structures.

\textbf{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-cycc-state = 'v state +
  break :: bool

Some utility lemmas for the simplifier, to handle idiosyncrasies of the record package.
lemma break-more-cong: state.more s = state.more s' \implies break s = break s'
  by (cases s, cases s', simp)

lemma [simp]: s() state.more := () break = foo () = s () break := foo ()
  by (cases s) simp

Defining the parameterization. We start at a default parameterization, where
all operations default to skip, and just add the operations we are interested
in: Initially, the break flag is false, it is set if we encounter a back-edge, and
once set, the algorithm shall terminate immediately.

definition cycc-params :: ('v,unit cycc-state-ext) parameterization
  where cycc-params ≡ dflt-parametrization state.more
    (RETURN (break = False )) (on-back-edge := λ . . . . . RETURN (break = True ),
    is-break := break )
lemmas cycc-params-simp[simp] =
  gen-parameterization.simps[mk-record-simp, OF cycc-params-def[simplified]]

interpretation cycc: param-DFS-defs where param=cycc-params for G .

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

definition cyc-checker G ≡ do {
  ASSERT (fb-graph G);
  s ← cycc.it-dfs TYPE(′a) G;
  RETURN (break s)
}

The total correct variant asserts finitely many reachable nodes.

definition cyc-checkerT G ≡ do {
  ASSERT (graph G ∧ finite (graph-defs.reachable G));
  s ← cycc.it-dfsT TYPE(′a) G;
  RETURN (break s)
}

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

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

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

sublocale DFS G cycc-params
  apply unfold-locales
  apply (simp-all add: cycc-params-def)
  done

122
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 straightforward from our parameterization, and we can use the \texttt{establish-invarI} rule provided by the DFS framework.

We use this example to illustrate the general proof scheme:

\textbf{lemma (in cycc) i-brk-eg-back: is-invar (λs. break s \iff back-edges s \neq \{\})}

\textbf{proof (induct rule: establish-invarI)}

The \texttt{[on-init cycc-params ≤_n SPEC (λx. ?I (empty-state x))]}; \texttt{∃s s’ v 0. [DFS-invar G cycc-params s; ?I s; cond s; \neg is-break cycc-params s; stack s = []; v 0 \in V 0; v 0 \notin dom (discovered s); s’ = new-root v 0 s] \rightarrow on-new-root cycc-params v 0 s’ ≤_n SPEC (λx. DFS-invar G cycc-params (s’[state := x]))} → \texttt{?I (s’[state := x])}]) ∨ \texttt{∃s s’ u v. [DFS-invar G cycc-params s; ?I s; cond s; \neg is-break cycc-params s; stack s \neq []; (u, v) \in pending s; u = hd (stack s); v \in dom (discovered s); v \in dom (finished s); s’ = cross-edge u v (s[pending := pending s \setminus \{(u, v)\}])]} → \texttt{on-cross-edge cycc-params u v s’ ≤_n SPEC (λx. DFS-invar G cycc-params (s’[state := x]))} → \texttt{?I (s’[state := x])})\]

\textbf{print-cases}

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

\textbf{case init thus ?case by (simp add: empty-state-def)}

\textbf{next}

\textbf{case (back-edge s s’ u v)}

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

\textbf{then interpret cycc-invar G s by simp}

However, here we do not need them:

\textbf{from back-edge show ?case by simp}

\textbf{qed (simp-all cong: break-more-cong)}

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

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

lemma (in cycc) i-brk-eq-back-short-proof:
  is-invar (\lambda s. break s ←→ 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:
  assumes NC: \neg cond s
  shows break s ←→ \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:
  assumes NC: \neg cond s
  shows break s ←→ \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 ≡ do {
  ASSERT (fb-graph G);
  SPEC (\lambda r. r ←→ \neg acyclic (g-E G \cap ((g-E G)^* \cap g-V0 G) \times UNIV))}

theorem cyc-checker-correct: cyc-checker G ≤ cyc-checker-spec G
unfolding cyc-checker-def cyc-checker-spec-def
proof (refine-vcg le-ASSERTI order-trans[OF DFS.it-dfs-correct], clarsimp-all)
assume fb-graph G
then interpret fb-graph G.
interpret cycc by unfold-locales
show DFS G cycc-params by unfold-locales
next
fix s
assume cycc-invar G s
then interpret cycc-invar G s.
assume ¬cycc.cond TYPE'(b) G s
thus break s = (¬ acyclic (g-E G ∩ cycc.reachable TYPE'(b) G × UNIV))
  by (rule cycc-correct-aux)
qed

The same for the total correct variant:
definition cyc-checkerT-spec G ≡ do {
  ASSERT (graph G ∧ finite (graph-defsreachable G));
  SPEC (λr. r ←→ ¬acyclic (g-E G ∩ (g-E G)∗ " g.V0 G) × UNIV))
}

theorem cyc-checkerT-correct: cyc-checkerT G ≤ cyc-checkerT-spec G
unfolding cyc-checkerT-def cyc-checkerT-spec-def
proof (refine-vcg le-ASSERTI order-trans[OF DFS.it-dfsT-correct], clarsimp-all)
assume graph G then interpret graph G .
assume finite (graph-defsreachable G)
then interpret fb-graph G by (rule fb-graphI-fr)
interpret cycc by unfold-locales
show DFS G cycc-params by unfold-locales
next
fix s
assume cycc-invar G s
then interpret cycc-invar G s.
assume ¬cycc.cond TYPE'(b) G s
thus break s = (¬ acyclic (g-E G ∩ cycc.reachable TYPE'(b) G × UNIV))
  by (rule cycc-correct-aux)
qed

2.1.3 Implementation

The implementation has two aspects: Structural implementation and data implementation. The framework provides recursive and tail-recursive implementations, as well as a variety of data structures for the state.

We will choose the simple-state implementation, which provides a stack, an on-stack and a visited set, but no timing information.

Note that it is common for state implementations to omit details from the very detailed abstract state. This means, that the algorithm’s operations must not access these details (e.g. timing). However, the algorithm’s correctness proofs may still use them.
We extend the state template to add a break flag

record 'v cycc-state-impl = 'v simple-state +
  break :: bool

Definition of refinement relation: The break-flag is refined by identity.

definition cycc-erel ≡ {
  (\(\|\) cycc-state-impl.break = b \|, \(\|\) cycc-state.break = b\|) | b. True }

abbreviation cycc-rel ≡ (cycc-erel) simple-state-rel

Implementation of the parameters

definition cycc-params-impl :: ('v, 'v cycc-state-impl, unit cycc-state-impl-ext) gen-parameterization
where cycc-params-impl
  ≡ dflt-parametrization simple-state.more (RETURN (\ break = False \)) (\
on-back-edge := \(\lambda\) u v s. RETURN (\ break = True \), is-break := break \)

lemmas cycc-params-impl-simp[simp,DFS-code-unfold] =
  gen-parameterization.simps[mk-record-simp, OF cycc-params-impl-def[simplified]]

Note: In this simple case, the reformulation of the extension state and parameterization is just redundant, However, in general the refinement will also affect the parameterization.

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)

interpretation cycc-impl: simple-impl-defs G cycc-params-impl cycc-params
  for G .

The above interpretation creates an iterative and a recursive implementation

term cycc-impl.tailrec-impl term cycc-impl.rec-impl
term cycc-impl.tailrec-implT — Note, for total correctness we currently only support tail-recursive implementations.

We use both to derive a tail-recursive and a recursive cyclicity checker:

definition [DFS-code-unfold]: cyc-checker-impl G ≡ do
  ASSERT (fb-graph G);
  s ← cycc-impl.tailrec-impl TYPE('a) G;
  RETURN (break s)
}
definition [DFS-code-unfold]: cyc-checker-rec-impl G ≡ do
  ASSERT (fb-graph G);
  s ← cycc-impl.rec-impl TYPE('a) G;
  RETURN (break s)
}
\textbf{definition} \hspace*{0.1em} [DFS-code-unfold]: \texttt{cyc-checker-implT} \texttt{G} \equiv \texttt{do} \{ \\
\hspace*{1em} \texttt{ASSERT} (\texttt{graph} \texttt{G} \land \texttt{finite} (\texttt{graph-defs.reachable} \texttt{G})); \\
\hspace*{1em} \texttt{s} \leftarrow \texttt{cyc-impl.tailrec-implT} \texttt{TYPE('a)} \texttt{G}; \\
\hspace*{1em} \texttt{RETURN} (\texttt{break} \texttt{s}) \\
\}

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

\textbf{context} \texttt{cyc} \textbf{begin}

\textbf{sublocale} \texttt{simple-impl} \texttt{G} \texttt{cycc-params} \texttt{cycc-params-impl} \texttt{cycc-erel}
\hspace*{1em} \textbf{apply} \hspace*{1em} \texttt{unfold-locales}
\hspace*{1em} \textbf{apply} \hspace*{1em} (\texttt{intro fun-relI, clarsimp simp: simple-state-rel-def, parametricity}) []
\hspace*{1em} \textbf{apply} \hspace*{1em} (\texttt{auto simp: cycc-erel-def break-impl simple-state-rel-def})
\hspace*{1em} \textbf{done}

We get that our implementation refines the abstract DFS algorithm.

\textbf{lemmas} \texttt{impl-refine} = \texttt{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:

\textbf{lemma}
\hspace*{1em} \texttt{cycc-impl.tailrec-impl TYPE('more)} \texttt{G} \leq \downarrow \hspace*{0.5em} \texttt{cycc-rel it-dfs}
\hspace*{1em} \texttt{cycc-impl.rec-impl TYPE('more)} \texttt{G} \leq \downarrow \hspace*{0.5em} \texttt{cycc-rel it-dfs}
\hspace*{1em} \texttt{cycc-impl.tailrec-implT TYPE('more)} \texttt{G} \leq \downarrow \hspace*{0.5em} \texttt{cycc-rel it-dfsT}
\hspace*{1em} \textbf{using} \hspace*{1em} \texttt{impl-refine} .

\textbf{end}

Finally, we get correctness of our cyclicity checker implementations

\textbf{lemma} \texttt{cyc-checker-impl-refine}: \texttt{cyc-checker-impl} \texttt{G} \leq \downarrow \hspace*{0.5em} \texttt{Id} (\texttt{cyc-checker G})
\hspace*{1em} \textbf{unfolding} \hspace*{1em} \texttt{cyc-checker-impl-def} \texttt{cyc-checker-def}
\hspace*{1em} \textbf{apply} \hspace*{1em} (\texttt{refine-vcg cyc.impl-refine})
\hspace*{1em} \textbf{apply} \hspace*{1em} (\texttt{simp-all add: break-impl cycI})
\hspace*{1em} \textbf{done}

\textbf{lemma} \texttt{cyc-checker-rec-impl-refine}:
\hspace*{1em} \texttt{cyc-checker-rec-impl} \texttt{G} \leq \downarrow \hspace*{0.5em} \texttt{Id} (\texttt{cyc-checker G})
\hspace*{1em} \textbf{unfolding} \hspace*{1em} \texttt{cyc-checker-rec-impl-def} \texttt{cyc-checker-def}
\hspace*{1em} \textbf{apply} \hspace*{1em} (\texttt{refine-vcg cyc.impl-refine})
\hspace*{1em} \textbf{apply} \hspace*{1em} (\texttt{simp-all add: break-impl cycI})
\hspace*{1em} \textbf{done}

\textbf{lemma} \texttt{cyc-checker-implT-refine}: \texttt{cyc-checker-implT} \texttt{G} \leq \downarrow \hspace*{0.5em} \texttt{Id} (\texttt{cyc-checkerT G})
\hspace*{1em} \textbf{unfolding} \hspace*{1em} \texttt{cyc-checker-implT-def} \texttt{cyc-checkerT-def}
\hspace*{1em} \textbf{apply} \hspace*{1em} (\texttt{refine-vcg cyc.impl-refine})
\hspace*{1em} \textbf{apply} \hspace*{1em} (\texttt{simp-all add: break-impl cycI'})
\hspace*{1em} \textbf{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.

\[
\text{record } ('s_i,'n_i,'p_i) \text{cycc-state-impl} = ('s_i,'n_i)\text{simple-state-impl} + \text{break-impl :: bool}
\]

We define the refinement relation for the state extension

\[
\text{definition } [\text{to-relAPP}]: \text{cycc-state-erel erel} \equiv \{ (b,\ldots,m) | (b\ldots,m) \in \text{bool-} \land (m\in\text{erel}) \}
\]

And register it with the Autoref tool:

\[
\text{consts }
\text{i-cycc-state-ext :: interface } \Rightarrow \text{interface}
\]

\[
\text{lemmas } [\text{autoref-rel-intf}] = \text{REL-INTFI[of cycc-state-erel i-cycc-state-ext]}
\]

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

\[
\text{lemma } [\text{autoref-rules}]:
\text{fixes } ns-rel \text{ vis-rel erel}
\text{defines } R \equiv \langle ns-rel,\text{vis-rel},\langle erel\rangle \text{cycc-state-erel}\rangle \text{ss-impl-rel}
\text{shows }
(cycc-state-impl'-ext, cycc-state-impl-ext) \in \text{bool-} \rightarrow \text{erel} \rightarrow (erel)\text{cycc-state-erel}
(break-impl, cycc-state-impl, break) \in R \rightarrow \text{bool-}
\text{unfolding cycc-state-erel-def ss-impl-rel-def R-def}
\text{by auto}
\]

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

\[
\text{schematic-goal cyc-checker-impl:}
\text{defines } V \equiv \text{Id :: ('v x 'v::hashable) set}
\text{assumes } [\text{unfolded V-def,autoref-rules}]:
(G_i, G) \in (Rm, V)\text{g-impl-rel-ext}
\text{notes } [\text{unfolded V-def,autoref-tyrel}]
\text{TYRELI[where } R=(V)\text{dflt-ahs-rel]}
\text{TYRELI[where } R=(V \times_r (V)\text{list-set-rel})\text{ras-rel]}
\text{shows } nres-of (?c::'?c dres) \leq⇓ _?R (cyc-checker-impl G)
\text{unfolding DFS-code-unfold}
\text{using } [[\text{autoref-trace-failed-id, goals-limit}=1]]
\text{apply (autoref-monadic (trace))}
\]
Combining the refinement steps yields a correctness theorem for the cyclicity checker implementation:

**Theorem cyc-checker-code-correct:**

- **Assumptions:**
  1. `fb-graph G`  
  2. `(Gi, G) ∈ ⟨Rm, Id⟩`  
  3. `cyc-checker-code Gi = dRETURN x`

- **Shows:**
  
  \( x ⇔ (¬acyclic (g-E G ∩ ((g-E G) * "g-V0 G) × UNIV)) \)

**Proof:**

- **Notes:**
  - `cyc-checker-code-refine[OF 2]`
  - `cyc-checker-impl-refine`
  - `cyc-checker-correct`

**Finally show:**

- **Unfolding:**
  - `cyc-checker-spec-def by auto`

**QED**

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

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

- **Defines:**
  
  \( V ≡ \text{Id} :: (′v × ′v::hashable) \)

- **Assumes:**
  
  \( (Gi, G) ∈ ⟨Rm, V⟩ \)

- **Notes:**
  
  \( V::\text{def, autoref-rules} \)

**Shows:**

\( nres-of (?c :: ?′c dres) ≤⇓ R (cyc-checker-rec-impl G) \)

**Unfolding:**

- `DFS-code-unfold`

**Using:**

- `autoref-trace-failed-id, goals-limit=1]`

**Apply:**

- `autoref-monadic (trace)`

**QED**

**Concrete-definition cyc-checker-rec-code uses cyc-checker-rec-impl**

**Export-code cyc-checker-rec-code checking SML**

**Lemma cyc-checker-rec-code-correct:**

- **Assumptions:**
  
  \( (fi, G) ∈ ⟨Rm, Id⟩ \)

- **Assumes:**
  
  \( cyc-checker-rec-code Gi = dRETURN x \)

**Shows:**

\( x ⇔ (¬acyclic (g-E G ∩ ((g-E G) * "g-V0 G) × UNIV)) \)

**Proof:**

- **Notes:**
  
  \( cyc-checker-rec-code-refine[OF 2] \)

**Finally show:**

- **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) dlIt-ahs-rel]
  TYRELI[where R=(V ×r (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 ≡ [| ppath = Some (p,v) |]

definition fp0-params :: ('v ⇒ bool) ⇒ 'v fp0-param
        where
          fp0-params P ≡ |
            on-init = RETURN no-path,
            on-new-root = λv0 s. if P v0 then RETURN (a-path [] v0) else RETURN no-path,
            on-discover = λu v s. if P v
              then — v is already on the stack, so we need to pop it again
                  RETURN (a-path (rev (tl (stack s))) v)
              else RETURN no-path,
            on-finish = λu s. RETURN (state.more s),
            on-back-edge = λu v s. RETURN (state.more s),
            on-cross-edge = λu v s. RETURN (state.more s),
            is-break = λs. ppath s ≠ None \)

lemmas fp0-params-simps[simp]
  = gen-parameterization.simps[mk-record-simp, OF fp0-params-def]

interpretation fp0: param-DFS-defs where param = fp0-params P
    for G P.

decl 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 (cases s, cases s') auto

sublocale DFS where param = fp0-params P
        by unfold-locales simp-all

132
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 (λs. ppath s = None −→ dom (discovered s) ∩ Collect P = { })
by (rule establish-invarI) simp-all

lemma i-path-to-P:
  is-invar (λs. ppath s = Some (vs, v) −→ P v)
by (rule establish-invarI) auto

lemma i-path-invar:
  is-invar (λs. ppath s = Some (vs, v) −→
    (vs ≠ [] −→ hd vs ∈ V0 ∧ path E (hd vs) vs v)
    ∧ (vs = [] −→ v ∈ V0 ∧ path E v vs v)
    ∧ (distinct (vs@[v])))
proof (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 ≠ [] by simp
  from discover have vnis: v ∈ set (stack s) using stack-discovered by auto

  from pendingD discover have v ∈ succ (hd (stack s)) by simp
  with hd-succ-stack-is-path[OF ne] have ∃ v ∈ V0. path E v (rev (stack s)) v.

  moreover from last-stack-in-V0 ne have last (stack s) ∈ V0 by simp
ultimately have path E (hd (rev (stack s))) (rev (stack s)) v hd (rev (stack s)) ∈ V0 using hd-rev[OF ne] path-hd[where p=rev (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[THEN make-invar-thm, rule-format]

lemmas path-to-P
= i-path-to-P[THEN make-invar-thm, rule-format]

lemmas path-invar
= i-path-invar[THEN make-invar-thm, rule-format]

lemma path-invar-nonempty:
  assumes ppath s = Some (vs,v)
  and vs ≠ []
  shows hd vs ∈ V0 path E (hd vs) vs v
  using assms path-invar
  by auto

lemma path-invar-empty:
  assumes ppath s = Some (vs,v)
  and vs = []
  shows v ∈ V0 path E v vs v
  using assms path-invar
  by auto

lemma fp0-correct:
  assumes ¬cond s
  shows 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]))
  proof (cases ppath s)
    case None with assms nc-discovered-eq-reachable no-path-no-P-discovered have reachable ∩ Collect P = {} by auto
    thus ?thesis by (auto simp add: None)
next
    case (Some vvs) then obtain v vs where [simp]: vvs = (vs,v)
      by (cases vvs) auto

    from Some path-invar[of vs] path-to-P[of - v] show ?thesis
      by auto
  qed

134
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 (p0[v])))
  apply (rule weaken-SPEC[OF it-dfs-correct])
  apply clarsimp
  apply (simp add: fp0-invar)
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)

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] =
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

\[
\text{restr-invar } E R P \equiv E^{\sim} R \subseteq R \land R \cap \text{Collect } P = \{\}
\]

**lemma** restr-invar-triv \([\text{simp, intro}]\)

- **unfolding** restr-invar-def \(\text{by simp}\)

**lemma** restr-invar-imp-not-reachable: restr-invar \(E R P \implies E^{\sim} R \cap \text{Collect } P = \{\}\)

- **unfolding** restr-invar-def \(\text{by (simp add: Image-closed-trancl)}\)

**type-synonym** \('v \text{ fpr-result } = 'v \text{ set } + ('v \text{ list } \times 'v)\)

**definition** find-path0-restr-pred \(G P R \equiv \lambda r.\)

\[
\text{case } r \text{ of }
\begin{align*}
\text{Inl } R' & \Rightarrow R' = R \cup (g-E G)^{\sim} g-V0 G \land \text{restr-invar} (g-E G) R' P \\
\text{Inr } (vs, v) & \Rightarrow P v \land (\exists v0 \in g-V0 G - R. \text{path} \text{(rel-restrict} (g-E G) R) v0 vs)
\end{align*}
\]

**definition** find-path0-restr-spec

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

where find-path0-restr-spec \(G P R \equiv \text{do}\)

\[
\begin{align*}
\text{ASSERT} \ (\text{fb-graph } G) & \text{;}
\text{SPEC} \ (\text{find-path0-restr-pred } G P R) \\
\text{SPEC} \ (\text{find-path0-restr-spec } G P R)
\end{align*}
\]

**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]
some \( (v, s) \Rightarrow R E N T U R \ (I n r \ (v, s)) \)

\[
\]

**Lemma** \texttt{find-path0-restr-correct}:

**shows** \( \texttt{find-path0-restr} G P R \leq \texttt{find-path0-restr-spec} G P R \)

**proof** (rule \texttt{le-ASSERT-defI1(OF find-path0-restr-spec-def)}, clarify)

**assume** \texttt{fb-graph G}

**interpret** \( a \cdot \texttt{fb-graph G} \) by fact

**interpret** \( \texttt{fb-graph graph-restrict G R} \) by (rule \texttt{a \cdot fb-graph-restrict})

**assume** \( I : \texttt{restr-invar} (g-E G) R P \)

**define** \( \texttt{reachable where reachable = graph-defs.reachable (graph-restrict G R)} \)

**interpret** \( \texttt{fp0 graph-restrict G R} \) by unfold-locale

**show** \( ?\texttt{thesis unfolding find-path0-restr-def find-path0-restr-spec-def} \)

**apply** (refine-rcg refine-vcg le-ASSERTI order-trans[OF it-dfs-correct])

**apply** unfold-locale

**apply** (clarsimp-all)

**proof** =

**fix** \( s \)

**assume** \( \texttt{fp0-invar (graph-restrict G R) P s} \)

**and** \( \texttt{NC[\texttt{simp}; \sim fp0.condition \ 'b' (graph-restrict G R) P s}} \)

**then interpret** \( \texttt{fp0-invar graph-restrict G R P s by simp} \)

\{

**assume** \( \texttt{[simp]; ppath s = None} \)

**from** \( \texttt{nc-discovered-eq-finished} \)

**show** \( \texttt{dom (discovered s) = dom (finished s) by simp} \)

**from** \( \texttt{nc-finished-eq-reachable} \)

**have** \( \texttt{DFR[\texttt{simp}; dom (finished s) = reachable} \)

**by** (simp add: reachable-def)

**from** \( I \)

**have** \( \texttt{g-E G \sqsubseteq R restr-invar-def by auto} \)

**have** \( \texttt{reachable \subseteq (g-E G)^* \sqsubseteq g-V0 G} \)

**unfolding** reachable-def

**by** (rule \texttt{Image-mono, rule rtrancl-mono}) (auto simp: rel-restrict-def)

**hence** \( \texttt{R \cup dom (finished s) = R \cup (g-E G)^* \sqsubseteq g-V0 G} \)

**apply** =

**apply** (rule equalityI)

**apply** auto []

**unfolding** \( \texttt{DFR reachable-def} \)
apply (auto elim: E-closed-restr-reach-cases[OF - (g-E G " R ⊆ R)]) []
done
moreover from nc-fin-closed I
have g-E G " (R ∪ dom (finished s)) ⊆ R ∪ dom (finished s)
  unfolding restr-invar-def by (simp add: rel-restrict-def) blast
moreover from no-path-no-P-discovered nc-discovered-eq-finished I
have (R ∪ dom (finished s)) ∩ Collect P = {}
  unfolding restr-invar-def by auto
ultimately
show find-path0-restr-pred G P R (Inl (R ∪ dom (finished s)))
  unfolding restr-invar-def find-path0-restr-pred-def by auto
}

{ fix v vs
  assume [simp]; ppath s = Some (vs,v)
  from fp0-correct
  show find-path0-restr-pred G P R (Inr (vs, v))
    unfolding find-path0-restr-pred-def by auto
  }
qed

2.2.3 Path of Minimal Length One, with Restriction

definition find-path1-restr-pred G P R ≡ λr.
case r of
  Inl R' ⇒ R' = R ∪ (g-E G)" " g-V0 G ∧ restr-invar (g-E G) R' P
  | Inr (vs,v) ⇒ P v ∧ vs ≠ [] ∧ (∃ v0 ∈ g-V0 G. path (g-E G ∩ UNIV × −R) v0 vs v)

definition find-path1-restr-spec — Find a path of length at least one to a target node that satisfies P. Takes an
  initial node restriction, and returns an extended node restriction.
  where find-path1-restr-spec G P R ≡ do
    ASSERT (fb-graph G ∧ restr-invar (g-E G) R P);
    SPEC (find-path1-restr-pred G P R))

lemmas find-path1-restr-spec-rule[refine-vcg] =
  ASSERT-le-defI[OF find-path1-restr-spec-def]
  ASSERT-leof-defI[OF find-path1-restr-spec-def]

definition find-path1-restr
  :: ('v, 'more) graph-rec-scheme ⇒ ('v ⇒ bool) ⇒ 'v set ⇒ 'v fpr-result nres
  where find-path1-restr G P R ≡
    FOREACHc (g-V0 G) is-Inl (λv0 s. do {
      ASSERT (is-Inl s); — TODO: Add FOREACH-condition as precondition in
      let R = projl s;
\[ f_0 \leftarrow \text{find-path0-restr-spec} \ (G \parallel g-V0 := g-E G \{ v_0 \}) \ P \ R; \]
\[
\begin{cases}
\text{case } f_0 \text{ of} \\
\quad \text{Inl} \Rightarrow \text{RETURN } f_0 \\
\quad \text{Inr} (\text{vs}, v) \Rightarrow \text{RETURN} (\text{Inr} (v_0 \# \text{vs}, v))
\end{cases}
\]
}

**definition** \text{find-path1-tailrec-invar} \ G P R0 it s \equiv
\[
\begin{cases}
\text{case } s \text{ of} \\
\quad \text{Inl} R \Rightarrow R = R0 \cup (g-E G)^{+} \{ g-V0 G - \text{it} \} \wedge \text{restr-invar} (g-E G) \ R P \\
\quad \text{Inr} (vs, v) \Rightarrow P v \land vs \neq [] \land (\exists v_0 \in g-V0 G \dashv \text{it} \ 	ext{path} (g-E G \cap \text{UNIV} \times -R0) v_0 \text{vs} v)
\end{cases}
\]

**lemma** \text{find-path1-restr-correct};
**shows** \text{find-path1-restr} \ G P R \leq \text{find-path1-restr-spec} \ G P R
**proof**
\[
\text{assume } \text{fb-graph } G
\]
\[
\begin{align*}
\text{interpret } a: & \ \text{fb-graph } G \text{ by fact} \\
\text{interpret } f_0: & \ \text{fb-graph } G \parallel g-E := g-E G \cap \\text{UNIV} \times -R \\
& \text{by (rule a.fb-graph-subset, auto)}
\end{align*}
\]

**assume** \ I: \text{restr-invar} (g-E G) \ R P

**have** \ aux2: \( \forall v_0. \ v_0 \in g-V0 G \Rightarrow \text{fb-graph} (G \parallel g-V0 := g-E G \{ v_0 \}) \)
\[
\text{by (rule a.fb-graph-subset, auto)}
\]

\[
\{ \\
\text{fix } v_0 \ \text{s} \\
\text{assume } IT: \ \text{it} \subseteq g-V0 G v_0 \in \text{it} \\
\text{and is-Inl s} \\
\text{and FPI: } \text{find-path1-tailrec-invar} \ G P R \ \text{it} s \\
\text{and RI: } \text{restr-invar} (g-E G) (\text{projl s} \cup (g-E G)^{+} \{ v_0 \}) \ P
\end{align*}
\]

then obtain \( R' \) where \( \text{simp}: s = \text{Inl} R' \) by (cases s) auto

from FPI have \( \text{simp}: R' = R \cup (g-E G)^{+} \{ g-V0 G - \text{it} \} \)

**unfolding** \text{find-path1-tailrec-invar-def} by simp

**have** \text{find-path1-tailrec-invar} \ G P R \ (\text{it} - \{ v_0 \})
\[
(\text{Inl} (\text{projl s} \cup (g-E G)^{+} \{ v_0 \}))
\]

using RI
\[
\text{by (auto simp: find-path1-tailrec-invar-def it-step-insert-iff[OF IT])}
\]

} note aux4 = this

\[
\{ \\
\text{fix } v_0 \ u \ \text{s} \ v \ p \\
\text{assume } IT: \ \text{it} \subseteq g-V0 G v_0 \in \text{it} \\
\text{and is-Inl s} \\
\text{and FPI: } \text{find-path1-tailrec-invar} \ G P R \ \text{it} s
\}
\]

139
and $PV: P v$
and $PATH: \text{path (rel-restrict (g-E G) (projl s))} u p v (v0,u) \in (g-E G)$
and $PR: u \notin \text{projl s}$

then obtain $R'$ where $[\text{simp}]: s = \text{Inl } R'$ by (cases $s$) auto

from FPI have $[\text{simp}]: R' = R \cup (g-E G)^+ \cap (g-V0 G - \text{it})$
unfolding $\text{find-path1-tailrec-invar-def}$ by simp

have $\text{find-path1-tailrec-invar G P R} (\text{it} - \{v0\}) (\text{Inr} (v0 \neq p, v))$
apply $(\text{simp add: find-path1-tailrec-invar-def PV})$
apply $(\text{rule bexI [where } x=v0])$
using $PR PATH(2)$ $\text{path-mono[OF rel-restrict-mono2[of R] PATH(1)]}$
apply $(\text{auto simp: path1-restr-cone})$ []

using $IT$ apply blast
done

} note $aux5 = this$

show $\text{thesis}$
unfolding $\text{find-path1-restr-def find-path1-restr-spec-def find-path1-restr-pred-def}$
apply $(\text{refine-vcg FOREACHc-rule [where } I=\text{find-path1-tailrec-invar G P R]})$

apply simp
using $I$ apply $(\text{auto simp add: find-path1-tailrec-invar-def restr-invar-def})$ []
apply $(\text{blast intro: aux2})$
apply $(\text{auto simp add: find-path1-tailrec-invar-def split: sum.splits})$ []
apply $(\text{auto simp: find-path0-restr-pred-def aux4 aux5 simp: trancl-Image-unfold-left[symmetric] split: sum.splits})$ []
apply $(\text{auto simp add: find-path1-tailrec-invar-def split: sum.splits})$ [2]
done
qed

definition $\text{find-path1-pred G P} \equiv \lambda r.$
case $r$ of
  None $\Rightarrow (g-E G)^+ \cap (g-V0 G \cap \text{Collect } P = \{\}$
| Some $(vs, v) \Rightarrow P v \land vs \neq [] \land (\exists v0 \in g-V0 G. \text{ path (g-E G) } v0 vs v)$
definition $\text{find-path1-spec}$
  — Find a path of length at least one to a target node that satisfies a given predicate.
where $\text{find-path1-spec G P} \equiv do$
  $\text{ASSERT (fb-graph G);}$
  $\text{SPEC (find-path1-pred G P)}$
lemmas find-path1-spec-rule[refine-vcg] =
  ASSERT-le-defI[OF find-path1-spec-def]
  ASSERT-leof-defI[OF find-path1-spec-def]

2.2.4 Path of Minimal Length One, without Restriction

definition find-path1 :: ('v, 'more) graph-rec-scheme ⇒ ('v ⇒ bool) ⇒ 'v fp-result nres
  where find-path1 G P ≡ do
    r ← find-path1-restr-spec G P {}; case r of
      Inl - ⇒ RETURN None
      | Inr vsv ⇒ RETURN (Some vsv)

lemma find-path1-correct:
  shows find-path1 G P ≤ find-path1-spec G P
  unfolding find-path1-def find-path1-spec-def find-path1-pred-def
  apply (refine-rcg refine-vcg le-ASSERTI order-trans[OF find-path1-restr-correct])
  apply simp
  apply (fastforce simp: find-path1-restr-spec-def find-path1-restr-pred-def
    split: sum.splits
    dest!: restr-invar-imp-not-reachable tranclD)
done

2.2.5 Implementation

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

definition fp0-erel ≡ |
  (fp0-state-impl.ppath = p) , (fp0-state.ppath = p) | p. True |
abbreviation fp0-rel R ≡ ⟨fp0-erel⟩ restr-simple-state-rel R
abbreviation no-path-impl ≡ ⟨fp0-state-impl.ppath = None⟩
abbreviation a-path-impl p v ≡ ⟨fp0-state-impl.ppath = Some (p,v)⟩

lemma fp0-rel-ppath-cong[simp]:
  (s,s') ∈ fp0-rel R ⇒ fp0-state-impl.ppath s = fp0-state.ppath s'
  unfolding restr-simple-state-rel-def fp0-erel-def
  by (cases s, cases s', auto)

lemma fp0-ss-rel-ppath-cong[simp]:
  (s,s') ∈ (fp0-erel) simple-state-rel ⇒ fp0-state-impl.ppath s = fp0-state.ppath s'
  unfolding simple-state-rel-def fp0-erel-def
  by (cases s, cases s', auto)
lemma fp0-cong: simple-state.more s = simple-state.more s'
\[ \Rightarrow fp0-state-impl.ppath s = fp0-state-impl.ppath s' \]
by (cases s, cases s', auto)

lemma fp0-erelI: p = p'
\[ \Rightarrow (\langle fp0-state-impl.ppath = p [], fp0-state-impl.ppath = p' [] \rangle) \in fp0-erel \]

unfolding fp0-erel-def by auto

definition fp0-params-impl
\[ \Rightarrow \langle 'v, 'v fp0-state-impl.('v,unit)fp0-state-impl-ext \rangle \]
gen-parameterization

where fp0-params-impl P \equiv \[
\begin{align*}
on-init & = \text{RETURN no-path-impl}, \\
on-new-root & = \lambda v0 s. \\
\text{if } P v0 \text{ then } \text{RETURN } (a-path-impl [] v0) \text{ else } \text{RETURN } no-path-impl, \\
on-discover & = \lambda u v s. \\
\text{if } P v \text{ then } \text{RETURN } (a-path-impl (map fst (rev (tl (CAST (ss-stack s))))) v) \text{ else } \text{RETURN } no-path-impl, \\
on-finish & = \lambda v s. \text{RETURN } (simple-state.more s), \\
on-back-edge & = \lambda u v s. \text{RETURN } (simple-state.more s), \\
on-cross-edge & = \lambda u v s. \text{RETURN } (simple-state.more s), \\
is-break & = \lambda s. ppath s \neq \text{None} \}
\end{align*}
\]


interpretation fp0-impl:
restricted-impl-defs fp0-params-impl P fp0-params P G R
for G P R.

locale fp0-restr = fb-graph
begin
sublocale fp0?: fp0 graph-restrict G R
apply (rule fp0I)
apply (rule fb-graph-restrict)
done

sublocale impl: restricted-impl G fp0-params P fp0-params-impl P fp0-erel R
apply unfold-locales
apply parametricity
apply (simp add: fp0-erel-def)
apply (auto) [1]

apply (auto
  simp: rev-map[symmetric] map-tl comp-def
  simp: fp0-erel-def simple-state-rel-def) [7]

apply (auto simp: restr-rel-def) [3]
apply (clarsimp simp: restr-rel-def)
apply (rule IdD)
apply (subst list-id-simp[symmetric])
apply parametricity
done
end

definition find-path0-restr-impl G P R ≡ do 
  ASSERT (fb-graph G);
  ASSERT (fp0 (graph-restrict G R));
  s ← fp0-impl.tailrec-impl TYPE('a) G R P;
  case ppath s of
  None ⇒ RETURN (Inl (visited s))
  | Some (vs,v) ⇒ RETURN (Inr (vs,v))
}

lemma find-path0-restr-impl[refine]:
shows find-path0-restr-impl G P R 
  ≤⟨⟨Id,Id×rId⟩⟩\sum-rel (find-path0-restr G P R)
proof (rule refine-ASSERT-defI2[OF find-path0-restr-def])
assume fb-graph G
then interpret fb-graph G .
interpret fp0-restr G by unfold-locales

show ?thesis
  unfolding find-path0-restr-impl-def find-path0-restr-def
  apply (refine-rcg impl.tailrec-refine)
  apply refine-dref-type
  apply (auto simp: restr-simple-state-rel-def)
done
qed

definition find-path0-impl G P ≡ do 
  ASSERT (fp0 G);
  s ← fp0-impl.tailrec-impl TYPE('a) {} P;
  RETURN (ppath s)
}

lemma find-path0-impl[refine]: find-path0-impl G P 
  ≤\parallel (⟨Id×,Id⟩\option-rel) (find-path0 G P)
proof (rule refine-ASSERT-defI1[OF find-path0-def])
assume fp0 G
then interpret fp0 G .
interpret r: fp0-restr G by unfold-locales

show ?thesis
  unfolding find-path0-impl-def find-path0-def
  apply (refine-rcg 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, . . . = mi),(ppath = p, . . . = m)) | pi mi p m.
(p,p)∈((Id)list-rel ×, Id)option-rel ∧ (mi,m)∈erel}

consts
i-fp0-state-ext :: interface ⇒ interface

lemmas [autoref-rel-intf] = REL-INTFI[of fp0-state-erel i-fp0-state-ext]

term fp0-state-impl-ext
lemma [autoref-rules]:
fixes ns-rel vis-rel erel
defines R ≡ (ns-rel,vis-rel, ⟨erel⟩fp0-state-erel)ssnos-impl-rel

shows
(fp0-state-impl'-ext, fp0-state-impl-ext)
∈ ((Id)list-rel ×, Id)option-rel → erel → ⟨erel⟩fp0-state-erel
(ppath-impl, fp0-state-impl,ppath) ∈ R → ((Id)list-rel ×, Id)option-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) ∈ (Rm, Id)g-impl-rel-ext
(Pi, P) ∈ Id → bool-rel
notes [autoref-tyrel] = TYRELI[where R=(Id::('v×'v) set)dflt-abs-ref]

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)
\begin{align*}
&\in \langle Rm, Rv \rangle \text{g-impl-rel-ext} \rightarrow (Rv \rightarrow \text{bool-rel}) \\
&\rightarrow \langle (\langle Rv \rangle \text{list-rel} \times, Rv) \text{option-rel} \rangle \text{nres-rel} \\
\text{apply} & \text{(intro fun-relI nres-relI)} \\
\text{unfolding} & \text{Vid} \\
\text{apply} & \text{(rule order-trans[OF find-path0-code.refine[param-fo, THEN nres-relD]], assumption+)} \\
\text{using} & \text{find-path0-impl find-path0-correct} \\
\text{apply} & \text{(simp add: pw-le-iff refine-pw-simps)} \\
\text{apply} & \text{blast} \\
\text{done} \\
\text{lemmas} & \text{find-path0-autoref[autoref-rules] = find-path0-autoref-aux[of PREFER-id-D]} \\
\text{schematic-goal} & \text{find-path0-restr-code:} \\
\text{fixes} & \text{vis-rel :: (}^{'v} \times {'}^{'v}\text{) set \Rightarrow (}^{'v}\text{visi} \times {'}^{'v}\text{set) set} \\
\text{notes} & \text{[autoref-rel-intf] = REL-INTFI[of vis-rel i-set for I]} \\
\text{assumes} & \text{[autoref-rules]: (op-vis-insert, insert)\in Id \rightarrow (Id)vis-rel \rightarrow (Id)vis-rel} \\
\text{assumes} & \text{[autoref-rules]: (op-vis-memb, (\in))\in Id \rightarrow (Id)vis-rel \rightarrow bool-rel} \\
\text{assumes} & \text{[autoref-rules]:} \\
& (Gi, G) \in \langle Rm, Id \rangle \text{g-impl-rel-ext} \\
& (Pi, P)\in Id \rightarrow bool-rel \\
& (Ri, R)\in (Id)vis-rel \\
\text{shows} & \text{(nres-of (?c::'v c dres),} \\
& \text{find-path0-restr-impl} \\
& \text{G} \\
& \text{P} \\
& (R::, (Id)vis-rel)) \in ?R \\
\text{unfolding} & \text{find-path0-restr-impl-def[abs-def] DFS-code-unfold ssnos-unfolds} \\
\text{unfolding} & \text{if-cancel not-not comp-def nres-monad-laws} \\
\text{using} & \text{[[autoref-trace-failed-id]]} \\
\text{apply} & \text{(autoref-monadic (trace))} \\
\text{done} \\
\text{concrete-definition} & \text{find-path0-restr-code uses find-path0-restr-code} \\
\text{export-code} & \text{find-path0-restr-code checking SML} \\
\text{lemma} & \text{find-path0-restr-autoref-aux:} \\
\text{assumes} & I: (op-vis-insert, insert)\in Rv \rightarrow (Rv)vis-rel \rightarrow (Rv)vis-rel \\
\text{assumes} & 2: (op-vis-memb, (\in))\in Rv \rightarrow (Rv)vis-rel \rightarrow bool-rel \\
\text{assumes} & \text{Vid: } Rv = Id \\
\text{shows} & (\lambda G P R. \text{nres-of (find-path0-restr-code op-vis-insert op-vis-memb G P R)},} \\
& \text{find-path0-restr-spec} \\
& \in \langle Rm, Rv \rangle \text{g-impl-rel-ext} \rightarrow (Rv \rightarrow bool-rel) \rightarrow (Rv)vis-rel \rightarrow}
\end{align*}
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: pw-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)∈⟨ ⟨Id⟩vis-rel, ⟨Id⟩list-rel ×r ⟨Id⟩sum-rel⟩nres-rel
   (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 ×r ⟨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)
   ∈ ⟨⟨⟨V⟩vis-rel, ⟨V⟩list-rel ×r ⟨V⟩sum-rel⟩nres-rel
proof –
note find-path1-restr-code.refine[OF G[simplified], param-fo, THEN nres-relD]
also note find-path1-restr-correct
finally show ?thesis by (force intro!: nres-relI)
qed

146

schematic-goal find-path1-code:
  assumes Vid: V = (Id :: 'a :: hashable rel)
  assumes [unfolded Vid,autoref-rules]:
    (Gi, G) ∈ (Rm, V)g-impl-rel-ext
    (Pi, P) ∈ V → bool-rel
  notes [autoref-tyrel] = TYRELI[where R=((Id::('a×'a::hashable)set))dflt-ahs-rel]
  shows (nres-of ?c_find-path1 G P)
    ∈ (∪{V}list-rel ×, V}option-rel)nres-rel
  unfolding find-path1-def[abs-def] Vid
  using [autoref-trace-failed-id]
  apply (autoref-monadic (trace))
  done
concrete-definition find-path1-code uses find-path1-code

export-code find-path1-code checking SML

lemma find-path1-code-autoref-aux:
  assumes Vid: V = (Id :: 'a :: hashable rel)
  shows (λ G P. nres-of (find-path1-code G P), find-path1-spec)
    ∈ (Rm, V)g-impl-rel-ext → (V → bool-rel) → (∪{V}list-rel ×, V}option-rel)nres-rel
  proof —
    note find-path1-code.refine[OF Vid, param-fo, THEN nres-relD, simplified]
    also note find-path1-correct
    finally show ?thesis by (force intro!: nres-relI)
  qed

lemmas find-path1-autoref[autoref-rules] = find-path1-code-autoref-aux[OF PREFER-id-D]

2.2.7 Conclusion

We have synthesized an efficient implementation for an algorithm to find a path to a reachable node that satisfies a predicate. The algorithm comes in four variants, with and without empty path, and with and without node restriction.

We have set up the Autoref tool, to insert this algorithms for the following specifications:

- find-path0-spec G P — find path to node that satisfies P.
- find-path1-spec G P — find non-empty path to node that satisfies P.
- find-path0-restr-spec G P R — find path, with nodes from R already searched.
• find-path1-restr-spec — find non-empty path, with nodes from $R$ already searched.

```
thm find-path0-autoref
thm find-path1-autoref
thm find-path0-restr-autoref
thm find-path1-restr-autoref
```

end

2.3 Set of Reachable Nodes

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

This theory provides a re-usable algorithm to compute the set of reachable nodes in a graph.

2.3.1 Preliminaries

```
lemma gen-obtain-finite-set:
  assumes F: finite $S$
  assumes E: $(e,\{\})\in\langle(R)Rs\\rangle$
  assumes I: $(i,\text{insert})\in R\rightarrow\langle(R)Rs\rangle$
  assumes EE: $\forall x. \ x\in S \Rightarrow \exists x_i. (x_i,x)\in R$
  shows $\exists Si. (Si,S)\in\langle(R)Rs\\rangle$
proof (induction)
  case empty thus ?case
  using E by (blast)
next
  case (insert $x$ S)
  then obtain $xi \ Si$ where 1: $(Si,S)\in\langle(R)Rs\rangle$ and 2: $(x_i,x)\in R$
  using EE unfolding $S'\text{-def}$ by blast
  from I[THEN fun-relD, OF 2, THEN fun-relD, OF I] show ?case ..
qed
```

```
lemma obtain-finite-ahs: finite $S$ $\Rightarrow \exists x. (x,S)\in\langle(Id)\text{dflt-ahs-rel}\\rangle$
```

148
apply (erule gen-obtain-finite-set)
apply autoref
apply autoref
by blast

2.3.2 Framework Instantiation

definition unit-parametrization ≡ dflt-parametrization (λ. ()) (RETURN ())

lemmas unit-parametrization-simp[simp, DFS-code-unfold] =
  dflt-parametrization-simp[mk-record-simp, OF, OF unit-parametrization-def]

interpretation unit-dfs: param-DFS-defs where param=unit-parametrization for G.

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

2.3.3 Correctness

context unit-DFS begin
  lemma find-reachable-correct: find-reachable G ≤ SPEC (λr. r = reachable)
  unfolding find-reachable-def
  apply (refine-vcg order-trans[OF it-dfs-correct])
  apply unfold-locales
  apply clarify
  apply blast
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

code 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 ×, (V)list-set-rel) ras-rel]
shows nres-of (?c::'?c dres) ≤⇑?R (find-reachable-impl G)
unfolding if-cancel DFS-code-unfold ssnos-unfolds
using [[autoref-trace-failed-id, goals-limit=1]]
apply (autoref-monadic (trace))
done
concrete-definition find-reachable-code uses find-reachable-impl
export-code find-reachable-code checking SML

lemma find-reachable-code-correct:
assumes 1: fb-graph G
assumes 2: (Gi, G) ∈ ⟨Rm, Id⟩ g-impl-rel-ext
assumes 4: find-reachable-code Gi = dRETURN r
shows (r, (g-E G) star '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)
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)r-as-rel]
shows RETURN (¿c::¿c') ≤⇓?R (find-reachable-implT G)
unfolding if-cancel DFS-code-unfold ssnos-unfolds
using [[autoref-trace-failed-id, goals-limit=1]]
apply (autoref-monadic (plain,trace))
done
concrete-definition find-reachable-codeT for eq bhc sz Gi
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.refine[OF BHC EQ VDS 2]
  also note find-reachable-implT-refine
  also note find-reachableT-correct
  finally show thesis using FR by (auto simp: RETURN-RES-refine-iff)
qed

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

lemma all-unit-refine[simp]:
  ((),x)∈all-unit-rel unfolding all-unit-rel-def by simp

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

152
lemma unit-list-rel-refine[simp]: \(((),y)\in (R)\) 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]:
\(((),[])\in (R)\) unit-list-rel
\((\lambda -. ().tl)\in (R)\) unit-list-rel\rightarrow (R)\) unit-list-rel
\((\lambda -. ().(#))\in R \rightarrow (R)\) unit-list-rel\rightarrow (R)\) unit-list-rel

by auto

schematic-goal find-reachable-rec-impl:
defines V \equiv Id :: ('v \times 'v::hasable) set
assumes [unfolded V-def,autoref-rules]:
\((Gi, G)\in (Rm, V)\) g-impl-rel-ext

notes [unfolded V-def,autoref-tyrel] =
TYRELI[where R=\(\langle V\rangle\) dflt-ahs-rel]

shows nres-of \((?c::'?c dres) \leq \nu R\) (find-reachable-rec-impl G)

unfolding unit-simple-impl.ssns-unfolds
  DFS-code-unfold if-cancel if-False option.case
using [[autoref-trace-failed-id, goals-limit=1]]
apply (autoref-monadic (trace))
done

concrete-definition find-reachable-rec-code uses find-reachable-rec-impl
prepare-code-thms find-reachable-rec-code-def
export-code find-reachable-rec-code checking SML

lemma find-reachable-rec-code-correct:
  assumes 1: fb-graph G
  assumes 2: \((Gi, G)\in (Rm, Id)\) g-impl-rel-ext
  assumes 4: find-reachable-rec-code Gi = dRETURN r

shows \((r, (g-E G)\ast ^{\cdot} g-V0 G)\in (\langle Id\rangle) dflt-ahs-rel\)

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 \equiv \((g-E G)\ast ^{\cdot} g-V0 G\)

lemmas [autoref-op-pat] = op-reachable-def[symmetric]

context begin interpretation autoref-syn .

lemma autoref-op-reachable[autoref-rules]:

153
fixes $V :: (\text{'vi} \times \text{'v}) \text{ set}$
assumes $G$: $\text{SIDE-PRECOND (graph } G)$
assumes $FR$: $\text{SIDE-PRECOND (finite } ((g-E G)^+ \times g-V0 G))$
assumes $BHC$: $\text{SIDE-GEN-ALGO (is-bounded-hashcode } V \text{ eq } bhc)$
assumes $EQ$: $\text{GEN-OP eq (\text{'v} \rightarrow \text{'v} \rightarrow \text{bool-rel})}$
assumes $VDS$: $\text{SIDE-GEN-ALGO (is-valid-def-hm-size TYPE} \text{'vi} \text{ sz)}$
assumes $2$: $(G_i, G) \in \langle Rm, V \rangle g-impl-rel-ext$
shows $(\text{find-reachable-codeT eq } bhc \text{ sz } G_i, \langle \text{OP op-reachable :: } (\text{'Rm, V} \rangle g-impl-rel-ext \rightarrow (V) ahs-rel bhc)\text{sz}(V) ahs-rel bhc)$
using asms
by (simp add: find-reachable-codeT-correct)

end

2.3.5 Conclusions

We have defined an efficient DFS-based implementation for $\text{op-reachable}$, 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 $is-fas :: \langle \text{'v}, \text{'more}\rangle \text{ graph-rec-scheme } \Rightarrow \langle \text{'v} \text{ rel} \Rightarrow \text{bool} \rangle$ where
$\text{is-fas } G \text{ EC } \equiv \neg (\exists u \in (g-E G)^+ \times \text{g-V0 G}. (u, u) \in (g-E G - EC)^+)$$
lemma $is-fas-alt$:
$\text{is-fas } G \text{ EC } = \text{acyclic } ((g-E G \cap (g-E G)^+ \times \text{g-V0 G} \times \text{UNIV}) - EC))$
unfolding $\text{is-fas-def} \text{ acyclic-def}$
proof (clarsimp, safe)
  fix $u$
  assume $A: (u, u) \in (g-E G \cap (g-E G)^+ \times \text{g-V0 G} \times \text{UNIV} - EC)^+$
  hence $(u, u) \in (g-E G - EC)^+$ by (rule trancl-mono) blast
  moreover from $A$ have $u \in (g-E G)^+ \times \text{g-V0 G}$ by (cases rule: converse-tranclE)
auto
  moreover assume $\forall u \in (g-E G)^+ \times \text{g-V0 G}. (u, u) \notin (g-E G - EC)^+$
  ultimately show $\text{False}$ by blast
next
  fix $u$ $v$0

154
assume 1: $v_0 \in g-V_0$ and 2: $(v_0, u) \in (g-E)^*$ and 3: $(u, u) \in (g-E - EC)^*$

have $(u, u) \in (Restr (g-E - EC) ((g-E)^* \ " g-V_0)^*)^*

apply (rule trancl-restrict-reachable[of 3, where $S=(g-E)^* \ " g-V_0)$]
apply (rule order-trans[of - rtrancl-image-unfold-right])
using 1 2 by auto

hence $(u, u) \in (Restr (g-E - EC) ((g-E)^* \ " g-V_0)$

apply (rule trancl-restrict-reachable[of 3, where $S=(g-E)^* \ " g-V_0)$]
apply (rule order-trans[of - rtrancl-image-unfold-right])
using 1 2 by auto

moreover assume $\forall x. (x, x) \notin (g-E \cap (g-E)^* \ " g-V_0 \times UNIV - EC)^+$

ultimately show False by blast

 qed

2.4.1 Instantiation of the DFS-Framework

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

lemma fas-more-cong: state.more s = state.more s' $\implies fas s = fas s'$
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 $\equiv$ dflt-parametrization state.more
(RETURN (| fas = {} |)) |
  on-back-edge := $\lambda 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 $G \equiv$ do {
  ASSERT (graph $G$);
  ASSERT (finite ((g-E)^* \ " g-V_0 $G$));
  s $\leftarrow$ fas.it-dfsT TYPE($a$) $G$;
  RETURN (fas-state.fas s)
}

locale fas $=$
param-DFS $G$ fas-params
for $G :: ('v, 'more) graph-rec-scheme$
+
assumes finite-reachable[simp, intro!]: finite ((g-E)^* \ " g-V_0 $G$)
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 find-fas G ≤ SPEC (is-fas G)
  unfolding find-fas-def
proof (refine-vcg le-ASSERTI order-trans[OF DFS.it-dfsT-correct], clarsimp-all)
interpret graph $G$ by fact
assume finite $((g \cdot E) G)^* \cdot \cdot \cdot g \cdot V0 G$
then interpret $fb$-graph $G$ by (rule $fb$-graph1-fr)
interpret $fas$ by unfold-locales fact
show DFS $G$ fas-params by unfold-locales
next
fix s
assume DFS-invar $G$ fas-params $s$
then interpret DFS-invar $G$ fas-params $s$
interpret fas-invar $G$ s by unfold-locales fact
assume $\neg fas$. cond TYPE('b) $G$ s
thus is-fas $G$ (fas-state.fas s)
by (rule find-fas-correct-aux)
qed (rule assms)+

2.4.3 Implementation

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

definition fas-erel ≡ {
  (\( fas-state-impl.fas = f \), \( fas-state.fas = f \)) \& f. True }
abbreviation fas-rel ≡ ⟨fas-erel⟩ simple-state-rel

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

lemma fas-impl: (si,s)∈fas-rel
  ===> fas-state-impl.fas si = fas-state.fas s
  by (cases si, cases s, simp add: simple-state-rel-def fas-erel-def)
interpretation fas-impl: simple-impl-defs G fas-params-impl fas-params
for $G$.

term fas-impl.tailrec-impl term fas-impl.tailrec-implT term fas-impl.rec-impl

definition [DFS-code-unfold]: find-fas-impl G ≡ do {
  ASSERT (graph G);
  ASSERT (finite $((g \cdot E) G)^* \cdot \cdot \cdot g \cdot V0 G)$;
  s ← fas-impl.tailrec-implT 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 ≡{
  ([fas-impl = fi, . . . = mi],([fas = f, . . . = m])) | fi mi f m.
  (fi,f)∈frel ∧ (mi,m)∈erel
}

consts i-fas-state-ext :: interface ⇒ interface ⇒ interface

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

term fas-update
term fas-state-impl',fas-impl-update
lemma [autoref-rules]:
  fixes ns-rel vis-rel frel erel
defines R ≡ (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
by (auto, parametricity)

schematic-goal find-fas-impl:
fixes \( V :: (V \times V) \) set
assumes [autoref-ga-rules]: \( \text{is-bounded-hashcode } V \text{ eq } bhc \)
assumes [autoref-rules]: \( (\text{eq}(=)) \in V \rightarrow V \rightarrow \text{bool-rel} \)
assumes [autoref-ga-rules]: \( \text{is-valid-def-hm-size } \text{TYPE} (\text{\textquote{vi}}) \) sz
assumes [autoref-rules]: \( (G_i, G) \in (\text{\textquote{Rm}}, V) \text{g-impl-rel-ext} \)

notes [autoref-tyrel] =
  \( \text{TYRELI[where} \text{R}=(V)\text{ahs-rel bhc} \)
  \( \text{TYRELI[where} \text{R}=(V \times_r V)\text{ahs-rel (prod-bhc bhc bhc)} \)
  \( \text{TYRELI[where} \text{R}=(V \times_r (V)\text{list-set-rel})\text{ras-rel} \)
shows \( \text{RETURN } (?c::?c) \leq ?R (\text{find-fas-impl } G) \)

unfolding DFS-code-unfold
using [[\text{autoref-trace-failed-id}, \text{goals-limit}=1]]
apply (\text{autoref-monadic} (\text{trace}))
done

concrete-definition find-fas-code for eq bhc sz Gi uses find-fas-impl

export-code find-fas-code checking SML

\text{thm find-fas-code-refine}

\text{lemma find-fas-code-refine[refine]:}
fixes \( V :: (V \times V) \) set
assumes \( \text{is-bounded-hashcode } V \text{ eq } bhc \)
assumes \( (\text{eq}(=)) \in V \rightarrow V \rightarrow \text{bool-rel} \)
assumes \( \text{is-valid-def-hm-size } \text{TYPE} (\text{\textquote{vi}}) \) sz
assumes \( 2: (G_i, G) \in (\text{\textquote{Rm}}, V) \text{g-impl-rel-ext} \)
shows \( \text{RETURN } (\text{find-fas-code eq bhc sz Gi}) \leq \downarrow (\text{\textquote{VxV}}\text{ahs-rel (prod-bhc bhc bhc)}) (\text{find-fas } G) \)

\text{proof}
  note find-fas-code-refine[OF assms]
  also note find-fas-impl-refine
  finally show \( ?\text{thesis} \).
qed

context begin interpretation autoref-syn .

Declare this algorithm to Autoref:

\text{theorem find-fas-code-autoref[autoref-rules]:}
fixes \( V :: (V \times V) \) set and bhc
defines \( RR \equiv ((V \times V)\text{ahs-rel (prod-bhc bhc bhc)})\text{nres-rel} \)
assumes \( BHC: \text{SIDE-GEN-ALGO} (\text{is-bounded-hashcode } V \text{ eq } bhc) \)
assumes \( EQ: \text{GEN-OP eq (=)} (V \rightarrow V \rightarrow \text{bool-rel}) \)
assumes \( VDS: \text{SIDE-GEN-ALGO} (\text{is-valid-def-hm-size } \text{TYPE} (\text{\textquote{vi}}) \) sz \)
assumes \( 2: (G_i, G) \in (\text{\textquote{Rm}}, V) \text{g-impl-rel-ext} \)
shows \( (\text{RETURN } (\text{find-fas-code eq bhc sz Gi}), (\text{OP find-fas} \)
  \( :: (\text{\textquote{Rm}}, V) \text{g-impl-rel-ext} \rightarrow RR)\$G)\in RR \)
unfolding RR-def
apply (rule nres-relI)
using assms

by (simp add: find-fas-code-refine)

end

2.4.5 Feedback Arc Set with Initialization

This algorithm extends a given set to a feedback arc set. It works in two steps:

1. Determine set of reachable nodes

2. Construct feedback arc set nodes for graph without initial set

**Definition** find-fas-init where

\[
\text{find-fas-init } G \ FI \equiv \begin{cases} \\
\text{ASSERT (graph } G); \\
\text{ASSERT (finite (}
\text{(g-E G)}^* \ g-V0 G); \\
\text{let nodes } = (g-E G)^* \ g-V0 G; \\
\text{fas } \leftarrow \text{find-fas } \{ g-V = \ g-V G, g-E = g-E G - F1, g-E G - FI, g-V0 = \text{nodes} \}; \\
\text{RETURN } (FI \cup fas) \end{cases}
\]

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

**Lemma** is-fas-join: is-fas \( G \ (F1 \cup F2) \leftrightarrow is-fas \ (g-V = g-V G, g-E = g-E G - F2, g-V0 = (g-E G)^* g-V0 G ) \)

**Proof**

unfolding is-fas-def

apply (auto simp: set-diff-diff-left Un-commute)

by (metis ImageI rtrancl-trans subsetCE rtrancl-mono[of g-E G - F2 g-E G, OF Diff-subset])

**Lemma** graphI-init:

assumes graph \( G \)

shows graph \( \{ g-V = g-V G, g-E = g-E G - F1, g-V0 = (g-E G)^* g-V0 G \} \)

proof –

interpret graph \( G \) by fact

show ?thesis

apply unfold-locales

using reachable-V apply simp

using E-ss apply force

done

qed

**Lemma** find-fas-init-correct:

assumes [simp, intro!]: graph \( G \)

assumes [simp, intro!]: finite \((g-E G)^* g-V0 G\)
shows \( \text{find-fas-init} G \text{FI} \leq \text{SPEC} \) (\( \lambda \text{fas. is-fas G fas} \land \text{FI} \subseteq \text{fas} \))

unfolding \( \text{find-fas-init-def} \)
apply \((\text{refine-vcg order-trans[OF find-fas-correct]})\)
apply \((\text{clarsimp-all})\)
apply \((\text{rule graphI-init})\)
apply \((\text{simp})\)
apply \((\text{rule finite-subset[rotated], rule assms})\)
apply \((\text{metis Diff-subset Image-closed-trancl reachable-mono rtrancl-image-unfold-right rtrancl-refcl trancl-trancl-refcl trancl-rtrancl-absorb})\)
apply \((\text{simp add: is-fas-join[where ?F2.0=FI] Un-commute})\)
done

lemma \( \text{gen-cast-set[autoref-rules-raw]}:\)
\begin{align*}
\text{assumes} & \quad \text{PRIO-TAG-GEN-ALGO} \\
\text{assumes} & \quad \text{INS: GEN-OP \text{ins Set.insert} (Rk→(Rk)Rs2→(Rk)Rs2)} \\
\text{assumes} & \quad \text{EM: GEN-OP \text{emp} \{\} ((Rk)Rs2)} \\
\text{assumes} & \quad \text{IT: SIDE-GEN-ALGO (is-set-to-list Rk Rs1 tsl)} \\
\text{shows} & \quad (\lambda s. \text{gen-union} (\lambda x. \text{foldli} (\text{tsl x}) \text{ ins s emp,CAST}) \\
& \quad \in ((Rk)Rs1) → ((Rk)Rs2))
\end{align*}
proof
−
have \( A : \forall s. (\lambda x. x\in s,CAST s) \in \text{br Collect } (\lambda -. True) \)
by \((\text{auto simp: br-def})\)
show \(?\text{thesis}\)
unfolding \( A \)
by \(\text{autoref}\)
qed

lemma \( \text{gen-cast-fun-set-rel[autoref-rules-raw]}:\)
\begin{align*}
\text{assumes} & \quad \text{INS: GEN-OP \text{mem} (\in) (Rk→(Rk)Rs→\text{bool-rel})} \\
\text{shows} & \quad (\lambda s x. \text{mem x s,CAST}) \in ((Rk)Rs) → ((Rk)\text{fun-set-rel})
\end{align*}
proof
−
have \( A : \forall s. (\lambda x. x\in s,CAST s) \in \text{br Collect } (\lambda -. True) \)
by \(\text{(auto simp: br-def)}\)
show \(?\text{thesis}\)
unfolding \(\text{fun-set-rel-def}\)
apply \((\text{rule})\)
apply \((\text{rule})\)
def er
apply \((\text{rule A})\)
using \(\text{INS[ simplified]}\)
apply \(\text{parametricity}\)
done
qed
lemma find-fas-init-impl-aux-unfolds:
Let \((E^*''V0) = \text{Let } (\text{CAST } (E^*''V0))\)
\((\lambda S. \text{RETURN } (FI \cup S)) = (\lambda S. \text{RETURN } (FI \cup \text{CAST } S))\)
by simp-all

schematic-goal find-fas-init-impl:
fixes \(V:: (\text{'v} \times \text{'v}) \text{ set and } bhc\)
assumes \([\text{autoref-ga-rules}]: \text{is-bounded-hashcode } V \text{ eq } bhc\)
assumes \([\text{autoref-rules}]: (eq,(=)) \in V \rightarrow V \rightarrow \text{bool-rel}\)
assumes \([\text{autoref-ga-rules}]: \text{is-valid-def-hm-size } \text{TYPE } (\text{'v}) \text{ sz}\)
assumes \([\text{autoref-rules}]: (Gi, G) \in (\text{Rm}, V) g-impl-rel-ext\)
\((FI,FI) \in (V \times rV) \text{fun-set-rel}\)
shows \(\text{RETURN } (?c::'?c) \leq ?R (\text{find-fas-init } G FI)\)
unfolding find-fas-init-def
unfolding find-fas-init-impl-aux-unfolds
by \((\text{autoref-monadic (plain,trace)})\)

concrete-definition find-fas-init-code for eq bhc sz Gi FIi
uses find-fas-init-impl
export-code find-fas-init-code checking SML

context begin interpretation autoref-syn .
The following theorem declares our implementation to Autoref:

theorem find-fas-init-code-autoref[autoref-rules]:
fixes \(V:: (\text{'v} \times \text{'v}) \text{ set and } bhc\)
defines \(RR \equiv (V \times V) \text{fun-set-rel}\)
assumes \(\text{SIDE-GEN-ALGO } (\text{is-bounded-hashcode } V \text{ eq } bhc)\)
assumes \(\text{GEN-OP eq } (=) \in (V \rightarrow V \rightarrow \text{bool-rel})\)
assumes \(\text{SIDE-GEN-ALGO } (\text{is-valid-def-hm-size } \text{TYPE } (\text{'v}) \text{ sz})\)
shows \((\lambda Gi FI. \text{RETURN } (\text{find-fas-init-code } eq \text{ bhc } sz \text{ Gi } FII), \text{find-fas-init})\)
\(\in (\text{Rm}, V) g-impl-rel-ext \rightarrow RR \rightarrow (RR) nres-rel\)
unfolding RR-def
apply \((\text{intro fun-relI nres-relI})\)
using assms
by \((\text{simp add: find-fas-init-code.refine})\)
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
find-fas-init-correct.

thm find-fas-code-autoref find-fas-init-code-autoref
thm find-fas-correct thm find-fas-init-correct

end

2.5 Nested DFS

theory Nested-DFS
imports DFS-Find-Path
begin

Nested DFS is a standard method for Buchi-Automaton emptiness check.

2.5.1 Auxiliary Lemmas

lemma closed-restrict-aux:
  assumes CL: $E^*F \subseteq F \cup S$
  assumes NR: $E^*U \cap S = \{\}$
  assumes SS: $U \subseteq F$
  shows $E^*U \subseteq F$
  — Auxiliary lemma to show that nodes reachable from a finished node must be finished if, additionally, no stack node is reachable
proof clarify
  fix $u,v$
  assume $A$: $(u,v) \in E^* u \in U$
  hence $M$: $E^*\{u\} \cap S = \{\} u \in F$ using NR SS by blast+
  from $A(1)$ $M$ show $v \in F$
  apply (induct rule: converse-rtrancl-induct)
  using CL apply (auto dest: rtrancl-Image-advance-ss)
done

qed

2.5.2 Instantiation of the Framework

record \'v blue-dfs-state = \'v state +
  lasso :: (\'v list \times \'v list) option
  red :: \'v set

type-synonym \'v blue-dfs-param = (\'v, (\'v,unit) blue-dfs-state-czt) parameterization

lemma lasso-more-cong[cong]: state.more s = state.more s' \implies lasso s = lasso s'
  by (cases s, cases s') simp
lemma red-more-cong[cong]: state.more s = state.more s' \implies red s = red s'
by (cases s, cases s') simp 

lemma [simp]: \( s |\) state.more := (\( lasso = \text{foo}, \) red := \text{bar} \) \( ) = \( s |\) lasso := \text{foo}, \) red := \text{bar} \( ) 
by (cases s) simp 

abbreviation dropWhileNot v ≡ dropWhile ((\( \neq \)) v) 
abbreviation takeWhileNot v ≡ takeWhile ((\( \neq \)) v) 

locale BlueDFS-defs = graph-defs G 
for G :: (\( 'v, 'more \)) graph-rec-scheme + 
fixes accept :: \( 'v \Rightarrow \) 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 \( :: 'v blue-dfs-state \Rightarrow 'v \) fpr-result \( \Rightarrow \) ('v,unit) blue-dfs-state-ext 
where 
mk-blue-witness s redS \( \equiv \) case redS of 
  Inl R' \( \Rightarrow \) (\( lasso = \text{None}, \) red = (\( R' \) |\( \text{\( \neq \) } \)) \( ) \) 
  | Inr (vs, v) \( \Rightarrow \) let rs = rev (stack s) in 
  (\( lasso = \text{Some} \) (rs, vs@dropWhileNot v rs), \( \text{red} = \text{red} s \)) 

definition run-red-dfs \( :: 'v \Rightarrow 'v blue-dfs-state \Rightarrow \) ('v,unit) blue-dfs-state-ext nres 
where 
run-red-dfs u s \( \equiv \) case lasso s of None \( \Rightarrow \) do 
  redS ← red-dfs (red s) (\( \lambda x. x = u \lor x \in cyan s \)) u; 
  RETURN (mk-blue-witness s redS) 
} 
| - \( \Rightarrow \) NOOP s 

Schwoon-Esparza extension 

definition se-back-edge u v s \( \equiv \) case lasso s of 
  None \( \Rightarrow \) 
  — it's a back edge, so u and v are both on stack 
  — we differentiate whether u or v is the 'culprit' 
  — to generate a better counter example 
  if accept u then 
  let rs = rev (tl (stack s)); 
  ur = rs; 
  ul = u#dropWhileNot v rs 
  in RETURN (l|lasso = \text{Some} \( (\text{ur},ul) \), \text{red} = \text{red} s) \) 

164
else if accept v then
  let rs = rev (stack s);
  vr = takeWhileNot v rs;
  vl = dropWhileNot v rs
  in RETURN (lasso = Some (vr,vl), red = red s)
else NOOP s

definition blue-dfs-params :: 'v blue-dfs-param
  where blue-dfs-params = []
    on-init = RETURN (lasso = None, red = [])
    on-new-root = lambda v s. NOOP s
    on-discover = lambda u v s. NOOP s
    on-finish = lambda u s. if accept u then run-red-dfs u s else NOOP s
    on-back-edge = se-back-edge
    on-cross-edge = lambda u v s. NOOP s
    is-break = lambda s. lasso s \neq None []

schematic-goal blue-dfs-params-simps[simp]:
  on-init blue-dfs-params = ?OI
  on-new-root blue-dfs-params = ?ONR
  on-discover blue-dfs-params = ?OD
  on-finish blue-dfs-params = ?OF
  on-back-edge blue-dfs-params = ?OBE
  on-cross-edge blue-dfs-params = ?OCE
  is-break blue-dfs-params = ?IB

unfolding blue-dfs-params-def gen-parameterization.simps
  by (rule refl)+

sublocale param-DFS-defs G blue-dfs-params
  by unfold-locales

end

locale BlueDFS = BlueDFS-defs G accept + param-DFS G blue-dfs-params
  for G :: ('v, 'more) graph-rec-scheme and accept :: 'v \Rightarrow 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-starts begin

lemma BlueDFS-invar-equiv[simp]:
  shows DFS-invar G blue-dfs-params s ↔ BlueDFS-invar G accept s
proof
  assume DFS-invar G blue-dfs-params s
  interpret DFS-invar G blue-dfs-params s by fact
  show BlueDFS-invar G accept s by unfold-locales
next
  assume BlueDFS-invar G accept s
  then interpret BlueDFS-invar G accept s .
  show DFS-invar G blue-dfs-params s by unfold-locales
qed

end

2.5.3 Correctness Proof

context BlueDFS begin

definition blue-basic-invar s ≡
  case lasso s of
  None ⇒ restr-invar E (red s) (∀ 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 = Empty
  shows restr-invar E (red s) (∀ x∈set (stack s))
and restr-invar E (red s) (∀ x∈set (stack s))
and restr-invar E (red s) (∀ x∈set (stack s))
using stack-reachable (stack s ≠ [])
apply (rule-tac restr-invar-subset, auto) []
apply (rule-tac restr-invar-subset, auto) []
using BI apply (simp add: blue-basic-invar-def)
done

lemma (in BlueDFS-invar) red-dfs-pres-bbi:
  assumes BI: blue-basic-invar s
  assumes [simp]: lasso s = Empty 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. DFS-invar G blue-dfs-params (finish (hd (stack s))) s|state.more := e)

166
proof –
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 (\text{stack} s \neq [])
unfolding finish-def cyan-def by (auto simp: neq-Nil-cone)

show ?thesis
unfolding run-red-dfs-def
apply simp
apply (refine-vcg)
apply simp

proof –
fix fp1
define s′ where s′ = \text{finish} (\text{hd} (\text{stack} s)) s
assume FP-spec: \text{find-path1-restr-pred} (G \ | \ g-V0 := \{\text{hd} (\text{stack} s)\} []) (\lambda x. x \in \text{set} (\text{stack} s))
(fp1)
assume BlueDFS-invar G accpt (s′|\text{state}.more := \text{mk-blue-witness} s′ fp1[])

then interpret i: BlueDFS-invar G accpt (s′|\text{state}.more := \text{mk-blue-witness} s′ fp1[])
by simp

have [simp]:
 red s′ = red s
discovered s′ = discovered s
dom (finished s′) = insert (\text{hd} (\text{stack} s)) (dom (finished s))
unfolding s′-def finish-def by auto

{
 fix R′
 assume [simp]: fp1 = Inl R′
 from FP-spec\[unfolded find-path1-restr-pred-def, simplified]\n have
 R′FMT: R′ = red s \cup E^+ \cdot \{\text{hd} (\text{stack} s)\}
 and RI: restr-invar E R′ (\lambda x. x \in \text{set} (\text{stack} s))
 by auto

 from BI have red s \subseteq dom (finished s)
 unfolding blue-basic-invar-def by auto
 also have E^+ \cdot \{\text{hd} (\text{stack} s)\} \subseteq dom (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 \land (u, v) \in E^*
 by (auto simp: trancl-unfold-left)

167
from R1 have NR: E+ "\{hd (stack s)\} \cap set (stack s) = {}" 
unfolding restr-invar-def by (auto simp: R'FMT)

with \(\langle hd (stack s), u\rangle \in E\) have u\# set (stack s) by auto 
with i.fmed-closed[simplified] \(\langle hd (stack s), u\rangle \in E\)

have UID: u\#dom (finished s) by (auto simp: stack-set-def)

from NR \(\langle hd (stack s), u\rangle \in E\) have NR': E* "\{u\} \cap set (stack s) = {}" 
by (auto simp: trancl-unfold-left)

have CL: E "\ dom (finished s) \subseteq dom (finished s) \cup set (stack s)" 
using finished-closed discovered-eq-finished-un-stack 
by simp

from closed-restrict-aux[OF CL NR'] UID have E* "\{u\} \subseteq dom (finished s)" by simp 
with \(\langle u, v\rangle \in E^*\) show v \in dom (finished s) by auto 
qed 
finally (sup-least) 
have R' \subseteq dom (finished s) \land red s \subseteq dom (finished s) 
by (simp add: R'FMT) 
} note aux1 = this

show blue-basic-invar (s'\{state.more := mk-blue-witness s' fp1\}) 
unfolding blue-basic-invar-def mk-blue-witness-def 
apply (simp split: option.splits sum.splits) 
apply (intro allI conjI impI)

using FP-spec SNE 
apply (auto simp: s'-def blue-basic-invar-def find-path1-restr-pred-def 
    simp: restr-invar-def 
    simp: neq-Nil-conv) []

apply (auto dest!: aux1) []
done 
qed 

lemma blue-basic-invar: is-invar blue-basic-invar
proof (induct rule: establish-invar1)
case (finish s) then interpret BlueDFS-invar where s=s by simp

have [simp]: \((\lambda x. x = hd (stack s) \lor x \in cyan (finish (hd (stack s)) s)) = 
(\lambda x. x\in set (stack s))\)
using (stack s \# \[])
unfolding finish-def cyan-def by (auto simp: neq-Nil-conv)

from finish show \?case
apply (simp)
apply (intro conjI impl)
apply (rule leaf-trans[OF red-dfs-pres-bbi], assumption+, simp)

apply (auto simp: restr-invar-def blue-basic-invar-def neg-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:
  \( E \subseteq \) red s \subseteq red s
using s-blue-basic-invar
unfolding blue-basic-invar-def restr-invar-def
by simp

lemma red-stack-disjoint:

169
\begin{verbatim}
set (stack s) \cap red s = \{\}
using s-blue-basic-invar
unfolding blue-basic-invar-def restr-invar-def by auto

lemma red-finished: red s \subseteq dom (finished s)
using s-blue-basic-invar
unfolding blue-basic-invar-def by auto

lemma all-nodes-colored: white s \cup blue s \cup cyan s \cup red s = UNIV
unfolding white-def blue-def cyan-def by (auto simp: stack-set-def)

lemma colors-disjoint:
  white s \cap (blue s \cup cyan s \cup red s) = \{}
  blue s \cap (white s \cup cyan s \cup red s) = \{}
  cyan s \cap (white s \cup blue s \cup red s) = \{}
  red s \cap (white s \cup blue s \cup cyan s) = \{}
unfolding white-def blue-def cyan-def
using finished-discovered red-finished
unfolding stack-set-def by blast

end

lemma (in BlueDFS) i-no-accpt-cyle-in-finish:
  is-invar (\lambda s. lasso s = None \longrightarrow (\forall x. accept x \land x \in dom (finished s) \longrightarrow (x,x) \notin E^+))
proof (induct rule: establish-invarI)
case (finish s s' u) then interpret BlueDFS-invar where s=s by simp

let ?onstack = \lambda x. x \in set (stack s)
let ?rE = rel-restrict E (red s)

fix R'::'v set
let ?R' = R' \{hd (stack s)\}
let ?s = s \{lasso := None, red := ?R'\}

assume \forall v. (hd (stack s), v) \in ?rE \Longrightarrow \neg ?onstack v
and accept: accept u
and NL[simp]: lasso s = None

end
\end{verbatim}
hence no-hd-cycle: \( (\text{hd } (\text{stack } s), \text{hd } (\text{stack } s)) \notin \mathcal{E}^+ \)
by auto

from finish have stack \( s \neq [\] \) by simp
from hd-in-set[of this] have \( \text{hd } (\text{stack } s) \notin \text{red } s \)
  using red-stack-disjoint
by auto
hence \( (\text{hd } (\text{stack } s), \text{hd } (\text{stack } s)) \notin \mathcal{E}^+ \)
using no-hd-cycle rel-restrict-tranclI red-closed[of NL]
by metis
with accept finish have
\( \forall x. \text{accept } x \land x \in \text{dom } (\text{finished } s) \rightarrow (x, x) \notin \mathcal{E}^+ \)
by auto
}

} with finish have
red-dfs (red \( s \)) ?onstack (\text{hd } (\text{stack } s))
\leq \text{SPEC } (\lambda x. \forall R. x = \text{Inl } R \rightarrow 
\text{DFS-invar } G \text{ blue-dfs-params (lasso-update Map.empty } s' (\text{red } = R) \rightarrow
\text{run-red-dfs-def mk-blue-witness-def cyan-def})
apply –
apply (rule find-path1-restr-spec-rule, intro conjI)
unfolding 1
apply (rule red-DFS-precond, simp-all) []
apply (rule red-DFS-precond, simp-all) []
apply (auto simp: find-path1-restr-pred-def restr-invar-def)
done
note aux = leof-trans[of this(simplified, THEN leaf-lift)]

note [refine-vcg del] = find-path1-restr-spec-rule

from finish show ?case
apply simp
apply (intro conjI impI)
unfolding run-red-dfs-def mk-blue-witness-def cyan-def
apply clarsimp
apply (refine-vcg aux)
apply (auto split: sum.splits)
done
next
case back-edge thus ?case
by (simp add: se-back-edge-def split: option.split)
qed simp-all

lemma no-accept-cycle-in-finish:
[lasso \( s \) = None; accept \( v \); \( v \in \text{dom } (\text{finished } s) \)] \( \rightarrow (v, v) \notin \mathcal{E}^+ \)
using i-no-accept-cycle-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 uw-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

moreover have path E (hd ?ul) ?ul (hd ?ul)

172
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 (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
{
  assume \neg \text{accept} (\text{hd} (\text{stack} \ s)) \Rightarrow \text{accept} \ v
  let \ ?s = s' (\text{state} . more := \text{state} . more \ s')

  assume \text{DFS-invar} \ G \ \text{blue-dfs-params} \ ?s

  from \text{assms have lasso-inv} \ ?s
  \quad \text{by (auto simp add: lasso-inv-def)}
}

ultimately show \ ?thesis
  using \ R \ s'
  unfolding \text{se-back-edge-def}
  by (auto split: option.splits)
qed

lemma lasso-inv:
  \text{is-invar lasso-inv}
proof (induct rule: establish-invarI)
  case (finish \ s \ s' \ u)
  then interpret \text{BlueDFS-invar} where \ s = s
  by simp

  let \ ?onstack = \lambda x . x \in \text{set} (\text{stack} \ s)
  let \ ?rE = \text{rel-restrict} \ E (\text{red} \ s)
  let \ ?revs = \text{rev} (\text{tl} (\text{stack} \ s))

  note \ ne = \langle \text{stack} \ s \neq [] \rangle
  note [simp] = \langle \text{u=hd} \ (\text{stack} \ s) \rangle

  from \text{finish have} [simp]:
  \ \\quad \forall x. x = \text{hd} (\text{stack} \ s) \lor x \in \text{set} (\text{stack} \ s') \iff x \in \text{set} (\text{stack} \ s)
  \ \\quad \text{red} \ s' = \text{red} \ s
  \ \\quad \text{lasso} \ s' = \text{lasso} \ s
  \ \\quad \text{by (auto simp: neq-Nil-conv)}
}

{\fix \ v \ vs
  let \ ?cyc = vs \@ \text{dropWhileNot} \ v \ ?revs
  let \ ?s = s' (\text{lasso} := \text{Some} (?\text{revs}, \ ?\text{cyc}), \text{red} := \text{red} \ s')

  assume \text{DFS-invar} \ G \ \text{blue-dfs-params} \ ?s
    \quad \text{and vs: vs \neq []} \ \text{path} \ ?rE (\text{hd} (\text{stack} \ s)) \ \text{vs} \ v
    \quad \text{and v: \?onstack} \ v
    \quad \text{and accept: accept} (\text{hd} (\text{stack} \ s))
  from \text{vs have} \ P: \text{path} \ E (\text{hd} (\text{stack} \ s)) \ \text{vs} \ v
by (metis path-mono rel-restrict-sub)

have hd$[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 (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 ∈ 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 P0 False P show ?thesis
        by (force intro: path-conc[OF P] dropWhileNot-path[where p=?revs])
  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), red:=red s))

175
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 ∀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 leaf-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 ∀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 ≠ []
    and accpt-loop: accpt (hd pl)
    and loop-is-path: path E (hd pl) pl (hd pl)
    and loop-reachable: ∃ v0∈V0. path E v0 pr (hd pl)
  using assms s-lasso-inv
  by (simp-all add: lasso-inv-def)

lemma blue-dfs-correct:
  assumes NC: ¬ cond s
shows case lasso s of
  None ⇒ ¬(∃v0∈V0. ∃v. (v0,v) ∈ E* ∧ accpt v ∧ (v,v) ∈ E+)}
| Some (pr,pl) ⇒ (∃v0∈V0. ∃v. path E v0 pr v ∧ accpt v ∧ pl≠[] ∧ path E v pl v)

proof (cases lasso s)
  case None
  moreover
  { fix v v0
    assume v0 ∈ V0 (v0,v) ∈ E* accpt v (v,v) ∈ E+
    moreover
    hence v ∈ reachable by (auto)
    with nc-finished-eq-reachable NC None have v ∈ dom (finished s)
    by simp
    moreover note no-accept-cycle-in-finish None
    ultimately have False by blast
  }
  ultimately show ?thesis by auto
next
  case (Some prpl) with s-lasso-inv show ?thesis
  by (cases prpl)
  (auto intro: path-is-rtrancl path-is-trancl simp: lasso-inv-def)
qed

end

2.5.4 Interface
interpretation BlueDFS-defs for G accpt.

definition nested-dfs-spec G accpt ≡ λr. case r of
  None ⇒ ¬(∃v0∈g-V0 G. ∃v. (v0,v) ∈ (g-E G)* ∧ accpt v ∧ (v,v) ∈ (g-E G)+)
| Some (pr,pl) ⇒ (∃v0∈g-V0 G. ∃v. path (g-E G) v0 pr v ∧ accpt v ∧ pl≠[] ∧ path (g-E G) v pl v)

definition nested-dfs G accpt ≡ do {
  ASSERT (fb-graph G);
  s ← it-dfs TYPE(a) G accpt;
  RETURN (lasso s)
}

theorem nested-dfs-correct:
  assumes fb-graph G
  shows nested-dfs G accpt ≤ SPEC (nested-dfs-spec G accpt)
proof –
  interpret fb-graph G by fact
  interpret BlueDFS G accpt by unfold-locales
  show ?thesis
unfolding nested-dfs-def
apply (refine-reg refine-reg)
apply fact
apply (rule weaken-SPEC[of it-dfs-correct])
apply clarsimp
proof -
  fix s
  assume BlueDFS-invar G accept s
  then interpret BlueDFS-invar G accept s.
  assume ¬cond TYPE(′b) G accept s
  from blue-dfs-correct[of this] show nested-dfs-spec G accept (lasso s)
  unfolding nested-dfs-spec-def by simp
qed
qed

2.5.5 Implementation

record ′v bdfs-state-impl = ′v simple-state +
  lasso-impl :: (′v list × ′v list) option
  red-impl :: ′v set

definition bdfs-erel ≡ {((lasso-impl=li,red-impl=ri),(lasso=l, red=r)) |
  li r l r ∧ 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′ ∪ lasso-impl=ri,red-impl=r) |
        Inr (vs, v) ⇒
          let
            rs = rev (map fst (CAST (ss-stack s)))
            in
              lasso-impl = Some (rs, vs @ dropWhileNot v vs),
              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 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)
} 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 vi ≤⇓ bdfs-erel (run-red-dfs TYPE('u) G u s)
unfolding run-red-dfs-impl-def run-red-dfs-def
apply refine-rcg
apply refine-dref-type
apply (cases si, cases s, auto simp: bdfs-erel-def simple-state-rel-def) []
apply (cases si, cases s, auto simp: bdfs-erel-def simple-state-rel-def cyan-impl-def cyan-def) []
apply (auto simp: bdfs-erel-def simple-state-rel-def) [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]: [(accept,accept)∈Id; (ui,u)∈Id; (vi,v)∈Id; (si,s)∈bdfs-rel]
⇒ se-back-edge-impl accept ui vi si ≤⇓ bdfs-erel (se-back-edge accept u v s)
unfolding se-back-edge-impl-def se-back-edge-def
apply refine-rev
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) ∈ bdfs-rel
⇒ RETURN (simple-state.more si) ≤⇓ bdfs-erel (NOOP s)
apply (simp add: pw-le-iff refine-pw-simps)
apply (auto simp: simple-state-rel-def)
done

definition bdfs-params-impl :: (′v, ′more) graph-rec-scheme ⇒ (′v ⇒ bool) ⇒ (′v,′v bdfs-state-impl,(′v,unit)bdfs-state-impl-ext)
gen-parameterization
where bdfs-params-impl G accpt ≡ (|
on-init = RETURN (lasso-impl = None, red-impl = {}), on-new-root = λv0 s. RETURN (simple-state.more s), on-discover = λu v s. RETURN (simple-state.more s), on-finish = λu s.
if accpt u then run-red-dfs-impl G u s else RETURN (simple-state.more s), on-back-edge = se-back-edge-impl accpt,
on-cross-edge = λu v s. RETURN (simple-state.more s), is-break = λs. lasso-impl s ≠ None |
lemmas bdfs-params-impl-simps[simp, DFS-code-unfold] =
gen-parameterization.simps|mk-record-simp, OF bdfs-params-impl-def]

interpretation impl: simple-impl-defs G bdfs-params-impl G accpt blue-dfs-params TYPE(′a) G accpt
for G accpt .

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)

180
apply (auto simp add: bdfs-erel-def simple-state-rel-def) []
done

lemmas impl = impl.simple-tailrec-refine
end
definition nested-dfs-impl G accpt ≡ do {
  ASSERT (fb-graph G);
  s ← impl.tailrec-impl TYPE('a) G accpt;
  RETURN (lasso-impl s)
}

lemma nested-dfs-impl[refine]:
  assumes (Gi,G) ∈ Id
  assumes (accepti,accept) ∈ Id
  shows nested-dfs-impl Gi accpti ≤⇓((Id)list-rel ×r (Id)list-rel)option-rel
    (nested-dfs G accpt)
  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 accpt ≤⇓bdfs-rel (it-dfs TYPE('b) G accpt) .
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' = l, red-impl' = r)) (lasso-impl = l, red-impl = r)) | l l r.
  (li,ri) ∈ ((Vi)list-rel ×r (Vi)list-rel)option-rel ∧ (ri,r) ∈ (Vi)dflt-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)
∈ ⟨⟨⟨Vi⟩list-rel ×r ⟨Vi⟩list-rel⟩option-rel⟩nres-rel
unfolding nested-dfs-impl-def[abs-def] Vid
se-back-edge-impl-def run-red-dfs-impl-def mk-blue-witness-impl-def
cyan-impl-def
DFS-code-unfold
using [[autoref-trace-failed-id]]
apply (autoref-monadic (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) →
(⟨⟨Vi⟩list-rel ×r ⟨Vi⟩list-rel⟩option-rel⟩nres-rel

182
proof –

from assms have Vid: $V = \text{Id}$ by simp
note nested-dfs-code.refine[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
lowlink-path $s\ v\ p\ w$ ≡ $\text{path} E\ v\ p\ w \land p \neq []$
\land (last p, w) \in \text{cross-edges} \text{ s} \cup \text{back-edges} \text{ s}
\land (\text{length} p > 1 \implies
p!1 \in \text{dom} (\text{finished} \text{ s})$
\land (\forall k < \text{length} p - 1. (p!k, p!Suc k) \in \text{tree-edges} \text{ s})

definition
lowlink-set $s\ v$ ≡ \{w ∈ dom (discovered \text{ s}). $v = w$
\lor (v,w) ∈ E^+ \land (w,v) ∈ E^+$
\land (∃ p. lowlink-path $s\ v\ p\ w$)\}

context begin interpretation timing-syntax .
abbreviation LowLink where
LowLink $s\ v$ ≡ Min ($δ$ $s$ · lowlink-set $s\ v$)
end end

context DFS-invar begin

lemma lowlink-setI:
assumes lowlink-path $s\ v\ p\ w$
and $w \in \text{dom} (\text{discovered} \text{ s})$
and $(v,w) \in E^+ (w,v) \in E^+$
shows $w \in \text{lowlink-set} \text{ s} \text{ v}$
proof (cases $v = w$)
  case True thus ?thesis by (simp add: lowlink-set-def assms)

end

183
next

\textbf{case} \texttt{False with} \texttt{assms have} \((v,w) \in E^+ (w,v) \in E^+ \texttt{ by} (\texttt{metis rtrancl-eq-or-trancl})^+

\texttt{with} \texttt{assms show} \ ?\texttt{thesis by} (\texttt{auto simp add: lowlink-set-def})

\texttt{qed}

\textbf{lemma} \texttt{lowlink-set-discovered:}
\begin{itemize}
\item \texttt{lowlink-set s v \subseteq dom (discovered s)}
\item \texttt{by blast}
\end{itemize}

\textbf{lemma} \texttt{lowlink-set-finite}: [\texttt{simp, intro}!]:
\begin{itemize}
\item \texttt{finite (lowlink-set s v)}
\item \texttt{using lowlink-set-discovered discovered-finite}
\item \texttt{by (metis finite-subset)}
\end{itemize}

\textbf{lemma} \texttt{lowlink-set-not-empty:}
\begin{itemize}
\item \texttt{assumes v \in dom (discovered s)}
\item \texttt{shows lowlink-set s v \neq \{\}}
\item \texttt{unfolding lowlink-set-def}
\item \texttt{using assms by auto}
\end{itemize}

\textbf{lemma} \texttt{lowlink-path-single:}
\begin{itemize}
\item \texttt{assumes (v,w) \in cross-edges s \cup back-edges s}
\item \texttt{shows lowlink-path s v [v] w}
\item \texttt{unfolding lowlink-path-def}
\item \texttt{using assms cross-edges-ssE back-edges-ssE}
\item \texttt{by (auto simp add: path-simps)}
\end{itemize}

\textbf{lemma} \texttt{lowlink-path-Cons:}
\begin{itemize}
\item \texttt{assumes lowlink-path s v (x\#xs) w}
\item \texttt{and xs \neq []}
\item \texttt{shows \exists u. lowlink-path s u xs w}
\item \texttt{proof –}
\item \texttt{from assms have path: path E v (x\#xs) w}
\item \texttt{and cb: (last xs, w) \in cross-edges s \cup back-edges s}
\item \texttt{and f: (x\#xs)!1 \in dom (finished s)}
\item \texttt{and t: (\forall k < length xs. ((x\#xs)!k, (x\#xs)!Suc k) \in tree-edges s)}
\item \texttt{unfolding lowlink-path-def}
\item \texttt{by auto}
\end{itemize}

\texttt{from path obtain u where path E u xs w by (auto simp add: path-simps)}

\textbf{moreover note} \texttt{cb (xs \neq [])}

\textbf{moreover} { \texttt{fix k define k' where k' = Suc k}
\begin{itemize}
\item \texttt{assume k < length xs - 1}
\item \texttt{with k'-def have k' < length xs by simp}
\item \texttt{with t have ((x\#xs)!k', (x\#xs)!Suc k') \in tree-edges s by simp}
\item \texttt{hence (xs!k, xs!Suc k) \in tree-edges s by (simp add: k'-def nth-Cons')}
\end{itemize}

\textbf{note t' = this}

\textbf{moreover} {
assume *: length xs > 1
from \( f \) have \( \text{xs!0} \in \text{dom} \ (\text{finished} \ s) \) by auto
moreover from \( \{\text{of} \{0\} * \) have \( \text{xs!0, xs!1} \) \in \text{tree-edges} \ s \) by simp
ultimately have \( \text{xs!1} \in \text{dom} \ (\text{finished} \ s) \) using tree-edge-impl-parenthesis by metis
ultimately have lowlink-path \( s \ u \) \( \text{xs w} \) by (auto simp add: lowlink-path-def)
thus \( \text{thesis} \) ..
qed

lemma lowlink-path-in-tree:
assumes \( p: \text{lowlink-path} \ s \ v \ p \ w \) and \( j: j < \text{length} \ p \) and \( k: k < j \) shows \( \langle p!k, p!j \rangle \in (\text{tree-edges} \ s)^+ \)
proof –
from \( p \) have \( p \neq [] \) by (auto simp add: lowlink-path-def)
thus \( \text{thesis} \) using \( p j k \)
proof (induction arbitrary: \( v \ j \ k \) rule: list-nonempty-induct)
case single
thus \( \text{case} \) by auto
next
case (cons \( x \) \( xs \))
define \( j' \) where \( j' = j - 1 \)
with cons have \( j'-le: j' < \text{length} \ xs \) and \( k \leq j' \) and \( j: j = \text{Suc} j' \) by auto
from cons lowlink-path-Cons obtain \( u \) where \( p: \text{lowlink-path} \ s \ u \) \( \text{xs w} \) by blast
show \( \text{case} \)
proof (cases \( k = 0 \))
  case True
  from cons have \( \land \ k, k < \text{length} \ (x#xs) - 1 \Longrightarrow ((x#xs)!k,(x#xs)!\text{Suc} k) \in \text{tree-edges} \ s \)
  unfolding lowlink-path-def
  by auto
moreover from True cons have \( k < \text{length} \ (x#xs) - 1 \) by auto
ultimately have *: \( ((x#xs)!k,(x#xs)!\text{Suc} k) \in \text{tree-edges} \ s \) by metis
show \( \text{thesis} \)
proof (cases \( j' = 0 \))
  case True with * \( j (k=0) \) show \( \text{thesis} \) by simp
next
case False with True have \( j' > k \) by simp
  with cons.IH' [OF p j'-le] have \( \text{xs!k, xs!j'} \) \in \( (\text{tree-edges} \ s)^+ \).
  with \( j \) have \( ((x#xs)!\text{Suc} k, (x#xs)!j) \in (\text{tree-edges} \ s)^+ \) by simp
  with * show \( \text{thesis} \) by (metis trancl-into-trancl2)
qed
next
case False
define k’ where k’ = k - 1
with False: |k ≤ j’| have k’ < j’ and k = Suc k’ by simp-all
with cons.H(OF p j’-le) have (xs!k’, xs!j’) ∈ (tree-edges s)+ by metis
hence ((x#xs)!Suc k’, (x#xs)!Suc j’) ∈ (tree-edges s)+ by simp
with k j show ?thesis by simp
qed
qed

lemma lowlink-path-finished:
assumes p: lowlink-path s v p w
and j: j < length p j > 0
shows p!j ∈ dom (finished s)
proof –
from j have length p > 1 by simp
with p have f: p!1 ∈ dom (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) ∈ (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: lowlink-path s v p w
and tree-edges: (u,v) ∈ (tree-edges s)+
and fin: u ∈ dom (finished s) ∨ (stack s ≠ [] ∧ u = hd (stack s))
shows ∃ p. lowlink-path s u p w
proof –
note lowlink-path-def[simp]

from tree-edges trancl-is-path obtain tp where
  tp: path (tree-edges s) u tp v tp ≠ []
  by metis

from tree-path-impl-parenthesis assms hd-stack-tree-path-finished have
  v-fin: v ∈ dom (finished s) by blast
from p have p!0 = hd p by (simp add: hd-conv-nth)
with p have p-0: p!0 = v by (auto simp add: path-hd)
let ?p = tp @ p

{ from tp path-mono[OF tree-edges-ssE] have path E u tp v by simp
  also from p have path E v p w by simp

186
finally have path E u \ ?p w.
}

moreover from p have \ ?p \neq \ [] by simp

moreover from p have (last \ ?p, w) \in cross-edges s \cup back-edges s by simp

moreover {
assume length \ ?p > 1

have \ ?p ! 1 \in dom (finished s)
proof (cases length tp > 1)
  case True hence tp1: \ ?p ! I = tp ! I by (simp add: nth-append)
  from tp True have (tp ! 0, tp ! I) \in (tree-edges s)'
    by (auto simp add: path-nth-conv nth-append elim: allE[where x=\0])
  also from True have tp ! 0 = hd tp by (simp add: hd-conv-nth)
  also from tp have hd tp = u by (simp add: path-hd)
  finally have tp ! 1 \in dom (finished s)
    using tree-path-impl-parenthesis fin hd-stack-tree-path-finished by blast
  thus \ ?thesis by (subst tp1)
next
  case False with tp have length tp = 1 by (cases tp) auto
  with p-0 have \ ?p ! 1 = v by (simp add: nth-append)
  thus \ ?thesis by (simp add: v-fin)
qed

also have \ \forall k < length \ ?p - 1. \ (?p!k, \ ?p!Suc k) \in tree-edges s
proof (safe)
  fix k
  assume A: k < length \ ?p - 1
  show (\ ?p!k, \ ?p!Suc k) \in tree-edges s
proof (cases k < length tp)
  case True hence k: \ ?p ! k = tp ! k by (simp add: nth-append)
  show \ ?thesis
  proof (cases Suc k < length tp)
    case True hence \ ?p ! Suc k = tp ! Suc k by (simp add: nth-append)
    moreover from True tp have (tp ! k, tp ! Suc k) \in tree-edges s
      by (auto simp add: path-nth-conv nth-append elim: allE[where x=k])
    ultimately show \ ?thesis
  next
  case False with True have *: Suc k = length tp by simp
  with tp True have (tp ! k, v) \in tree-edges s
    by (auto simp add: path-nth-conv nth-append elim: allE[where x=k])
  also from * p-0 have v = \ ?p ! Suc k by (simp add: nth-append)
  finally show \ ?thesis by (simp add: k)
qed

187
next
  case False hence *: Suc k − length tp = Suc (k − length tp) by simp
  define k' where k' = k − length tp
  with False have k': ?p ! k = p ! k' ?p ! Suc k = p ! Suc k'
  by (simp-all add: nth-append)
  from k'-def False A have k' < length p − 1 by simp
  with p have (p!k', p!Suc k') ∈ tree-edges s by simp
  with k' show *thesis by simp
qed
qed

also (conjI) note calculation

ultimately have lowlink-path s u ?p w by simp
thus *thesis ..
qed

lemma lowlink-path-complex:
  assumes (u,v) ∈ (tree-edges s)+
  and u ∈ dom (finished s) ∨ (stack s ≠ [] ∧ u = hd (stack s))
  and (v,w) ∈ cross-edges s ∪ back-edges s
  shows ∃ p. lowlink-path s u p w
proof −
  from assms lowlink-path-single have lowlink-path s v [v] w by simp
  with assms lowlink-path-tree-prepend show *thesis by simp
qed

lemma no-path-imp-no-lowlink-path:
  assumes edges s "{v} = {}
  shows ¬lowlink-path s v p w
proof
  assume p: lowlink-path s v p w
  hence [simp]: p≠[] by (simp add: lowlink-path-def)
  from p have hd p = v by (auto simp add: lowlink-path-def path-hd)
  with hd-conv-nth[OF (p≠[])] have v: p!0 = v by simp
  show False
    proof (cases length p > 1)
      case True with p have (p!0,p!1) ∈ tree-edges s by (simp add: lowlink-path-def)
      with v assms show False by auto
    next
      case False with (p≠[]): have length p = 1 by (cases p) auto
      hence last p = v by (simp add: last-conv-nth v)
      with p have (v,w) ∈ edges s by (simp add: lowlink-path-def)
      with assms show False by auto
    qed
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 contr)
interpret s’: DFS-invar where s=s’ by fact
assume A: LowLink s w ≠ LowLink s’ w
from lowlink-set-discovered sub sub-m w-disc have
  sub': δ s' lowlink-set s w ⊆ δ s' lowlink-set s' w
and w-disc': w ∈ dom (discovered s')
and eq: ∀ll. ll ∈ lowlink-set s w =⇒ δ s' ll = δ s ll
by (force simp: map-le-def)+

from lowlink-set-not-empty[OF w-disc] A Min-antimono[OF sub'] s'.lowlink-set-finite
have
  LowLink s' w < LowLink s w by fastforce
then obtain ll where ll ∈ lowlink-set s' w and ll-le: δ s ll < LowLink s w
by (metis s' LowLink-lessE w-disc')

hence
  LowLink s w ≤ δ s ll by (force simp add: lowlink-set-def)

proof
  assume ll ∈ lowlink-set s w with
  lowlink-set-finite eq show ?thesis by force
  next
  assume ll ∈ X with
  ll show ?thesis by (metis X)
  qed
with ll-le show False by simp
qed

lemma LowLink-eq-disc-iff-scc-root:
  assumes v ∈ dom (finished s) ∨ (stack s ≠ [] ∧ v = hd (stack s) ∧ pending s "{v} = {v}"
  shows LowLink s v = δ s v =⇒ scc-root s v (scc-of E v)
proof
  let ?scc = scc-of E v
  assume scc: scc-root s v ?scc
  show LowLink s v = δ s v
proof (rule ccontr)
  assume A: LowLink s v ≠ δ s v

from assms finished-discovered stack-discovered hd-in-set have disc: v ∈ dom
  (discovered s) by blast
with assms LowLink-le-disc A have LowLink s v < δ s v by force
with disc obtain w where w: δ s w < δ s v w ∈ lowlink-set s v
by (metis LowLink-lessE)
with lowlink-set-discovered have wdisc: w ∈ dom (discovered s) by auto

from w have (v, w) ∈ E* (w, v) ∈ E* by (auto simp add: lowlink-set-def)
moreover have is-scc E ?scc v ∈ ?scc by simp-all
ultimately have w ∈ ?scc by (metis is-scc-closed)
with wdisc scc-root-disc-le[OF ?scc] have δ s v ≤ δ s w by simp
with w show False by auto
qed
next
  assume LL: LowLink s v = δ s v

190
from assm finished-discovered stack-discovered hd-in-set have
v-disc: v ∈ dom (discovered s) by blast

from assms finished-no-pending have
v-no-p: pending s "\{v\} = {}" by blast

let ?scc = scc-of E v
have is-scc: is-scc E ?scc by simp

{ fix r
  assume r ≠ v
  and r ∈ ?scc r ∈ dom (discovered s)

  have v ∈ ?scc by simp
  with (r ∈ ?scc) is-scc have \( (v,r) ∈ (Restr E ?scc)^* \)
    by (simp add: is-scc-connected')

  hence \( (v,r) ∈ (tree-edges s)^+ \) using \( (r ≠ v) \)
  proof (induction rule: rtrancl-induct)
    case (step y z) hence \( (v,z) ∈ (Restr E ?scc)^* \)
      by (metis rtrancl-into-rtrancl)
    hence \( (v,z) ∈ E^* \) by (metis Restr-rtrancl-mono)

  from step have \( (z,v) ∈ E^* \) by (simp add: is-scc-connected[OF is-scc])

  { assume z-disc: z ∈ dom (discovered s)
    and ∃ p. lowlink-path s v p z
    with \( (z,v) ∈ E^* \), \( (v,z) ∈ E^* \) have ll: z ∈ lowlink-set s v
      by (metis lowlink-setI)
    have δ s v ≤ δ s z
      by (rule ccontr)
    hence δ s v ≥ δ s z with \( (z ≠ v) \) v-disc disc-unequal have δ s z
      by fastforce
      with ll have LowLink s v < δ s v by (metis LowLink-lessI)
      with LL show False by simp
    qed simp
  } note δ z = this

  show ?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 ∈ dom (discovered s) by blast

    from step edges-covered v-no-p v-disc have \( (v,z) ∈ edges s \) by auto
    thus ?thesis

  qed simp
proof (rule edgesE-CB)
assume (v, z) ∈ tree-edges s thus ?thesis ..
next
assume CB: (v, z) ∈ cross-edges s ∪ back-edges s
hence lowlink-path s v [v, z]
by (simp add: lowlink-path-single)
with δz (OF z-disc) no-pending-succ-impl-path-in-tree v-disc v-no-p step
show ?thesis
by auto
qed
next
case False with step.IH have T: (v, y) ∈ (tree-edges s)⁺ .
with tree-path-impl-parenthesis assms hd-stack-tree-path-finished tree-path-disc
have y-fin: y ∈ dom (finished s)
and y-δ : δ s v < δ s y by blast+
with step have z-disc: z ∈ dom (discovered s)
using finished-imp-succ-discovered
by auto
from step edges-covered finished-no-pending[of y] y-fin finished-discovered
have (y, z) ∈ edges s
by fast
thus ?thesis
proof (rule edgesE-CB)
assume (y, z) ∈ tree-edges s with T show ?thesis ..
next
assume CB: (y, z) ∈ cross-edges s ∪ back-edges s
with lowlink-path-complex (OF T) assms have
∃ p. lowlink-path s v p z by blast
with δz z-disc have δz: δ s v < δ s z by simp
show ?thesis
proof (cases v ∈ dom (finished s))
case True with tree-path-impl-parenthesis T have y-f: φ s y < φ s v
by blast
from CB show ?thesis
proof
assume C: (y, z) ∈ cross-edges s
with cross-edges-finished-decr y-fin y-f have φ s z < φ s v
by force
moreover note δz
moreover from C cross-edges-target-finished have
z ∈ dom (finished s) by simp
ultimately show ?thesis
using parenthesis-impl-tree-path (OF True) by metis
next

assume $B: (y,z) \in \text{back-edges } s$
with $\text{back-edge-disc-lt-fin } y$-$\text{fin } y$-$f$
have $\delta s z < \varphi s v$
by $\text{force}$
moreover note $\delta z z$-$\text{disc}$
ultimately have $z \in \text{dom } (\text{finished } s)$ $\varphi s z < \varphi s v$
using $\text{parenthesis-contained}[\text{OF True}]$ by $\text{simp-all}$
with $\delta z$ show $\text{thesis}$
using $\text{parenthesis-impl-tree-path}[\text{OF True}]$ by $\text{metis}$
qed

next
  case $\text{False} : v \notin \text{dom } (\text{finished } s)$
  with $\text{assms}$ have $\text{st}$: $\text{stack } s \neq []$ $v = \text{hd } (\text{stack } s)$ $\text{pending } s " \{v\} = \{\}$ by $\text{blast+}$

  have $z \in \text{dom } (\text{finished } s)$
  proof (rule $\text{ccnot}$)
    assume $z \notin \text{dom } (\text{finished } s)$
    with $\text{z-disc}$ have $z \in \text{set } (\text{stack } s)$ by (simp add: $\text{stack-set-def}$)
    with $(z \neq v)$ $\text{st}$ have $z \in \text{set } (\text{tl } (\text{stack } s))$ by (cases $\text{stack } s$) $\text{auto}$
    with $\text{st tl-lt-stack-hd-discover } \delta z$ show $\text{False}$ by $\text{force}$
  qed

  with $\delta z$ $\text{parenthesis-impl-tree-path-not-finished } v$-$\text{disc}$ $\text{False}$ show $\text{thesis}$
  by $\text{simp}$
  qed
  qed
  qed
  simp

  hence $v \in (\text{tree-edges } s)^* " \{v\} \text{ by } \text{auto}$
  }

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

  thus $\text{scc-root } s v ?\text{scc}$ by (auto intro: $\text{scc-rootI } v$-$\text{disc}$)
qed
end
end

2.7 Tarjan’s Algorithm

theory Tarjan
imports
  Tarjan-LowLink
begin

We use the DFS Framework to implement Tarjan’s algorithm. Note that,
currently, we only provide an abstract version, and no refinement to efficient
code.
2.7.1 Preliminaries

lemma tjs-union:
  fixes tjs u
  defines dw ≡ dropWhile (≠ u) tjs
  defines tw ≡ takeWhile (≠ u) tjs
  assumes u ∈ set tjs
  shows set tjs = set (tl dw) ⊔ insert u (set tw)
proof
  from takeWhile-dropWhile-id have set tjs = set (tw @ dw)
    by (auto simp: dw-def tw-def)
  hence set tjs = set tw ⊔ set dw
    by (metis set-append)
  moreover from ⟨u ∈ set tjs⟩ dropWhile-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
typsig 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 \( \text{tj-stack-pop-set} \):
\[
\text{tj-stack-pop} \ tjs \ u \leq \text{SPEC} \ (\lambda(tjs', scc). \ u \in \text{set} \ tjs \rightarrow \text{set} \ tjs = \text{set} \ tjs' \cup \text{set} \ scc \land u \in \text{set} \ scc)
\]

\( \text{proof} \)
- from \( \text{tjs-union[of u tjs]} \) show \( \text{thesis} \)
  unfolding \( \text{tj-stack-pop-def} \)
by (refine-vcg) auto
qed

lemmas \( \text{tj-stack-pop-set-leaf-rule} = \text{weaken-SPEC[of tj-stack-pop-set, THEN leaf-lift]} \)

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}
\begin{align*}
&\text{let ll = (if stack s = [] then lowlink s} \\
&\quad\text{else let u = hd (stack s) in}
&\quad\text{(lowlink s)(u \mapsto \text{min} (\zeta \ s \ u) (\zeta \ s \ v))};
&\text{let s' = s[\text{lowlink} := ll ];}
&\text{ASSERT (v \in \text{set} (\text{tj-stack s})};
&\text{ASSERT' (distinct (\text{tj-stack s})});
&\text{if \( \zeta \ s \ v = \delta \ s \ v \) then do \{
&\quad\text{ASSERT (\text{scc-root' E s v (\text{scc-of E v});
&\quad((tjs, scc) \leftarrow \text{tj-stack-pop (tj-stack s) v;
&\quad\text{RETURN (state.more (s[\text{tj-stack} := tjs, \text{sccs} := \text{insert scc (sccs s)])})}
&\text{} \}) \text{ else do } \{
&\quad\text{ASSERT (\neg \text{scc-root' E s v (\text{scc-of E v});
&\quad\text{RETURN (state.more s')}
&\text{})}
\end{align*}
\]

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}
\begin{align*}
&\text{let ul' = \text{min} (\zeta \ s \ u) (\delta \ s \ v)}
&\text{in \text{RETURN (state.more (s[\text{lowlink} := (\text{lowlink s)(\text{u\rightarrow ul'})])})}
&\text{else NOOP s)
&\end{align*}
\]

definition \( \text{tarjan-params} :: 'v \text{tarjan-param} \) where
\[
\text{tarjan-params} = ()
\begin{align*}
\text{on-init} = \text{RETURN () \text{sccs} = \{\}}, \text{lowlink} = \text{Map.empty}, \text{tj-stack} = []
&\text{on-new-root = tarjan-disc,}
&\text{on-discover = \( \lambda u. \text{tarjan-disc},
&\text{on-finish = tarjan-fin,}
&\text{on-back-edge = tarjan-back,}
&\text{on-cross-edge = tarjan-back,}
\end{align*}
\]

195
is-break = λs. False 

schematic-goal tarjan-params-simps[simp]:
on-init tarjan-params = ?OI
on-new-root tarjan-params = ?ONR
on-discover tarjan-params = ?OD
on-finish tarjan-params = ?OF
on-back-edge tarjan-params = ?OBE
on-cross-edge tarjan-params = ?OCE
is-break tarjan-params = ?IB
unfolding tarjan-params-def gen-parameterization.simps
by (rule refl)+

sublocale param-DFS-defs G tarjan-params .
end

locale Tarjan = Tarjan-def G +
param-DFS G tarjan-params
for G :: (′v, ′more) graph-rec-scheme
begin

lemma [simp]:
sccs (empty-state (sccs = s, lowlink = l, tj-stack = t)) = s
lowlink (empty-state (sccs = s, lowlink = l, tj-stack = t)) = l
tj-stack (empty-state (sccs = s, lowlink = l, tj-stack = t)) = t
by (simp-all add: empty-state-def)

lemma sccs-more-cong[cong]: state.more s = state.more s' → sccs s = sccs s'
by (cases s, cases s') simp

lemma lowlink-more-cong[cong]: state.more s = state.more s' → lowlink s =
lowlink s'
by (cases s, cases s') simp

lemma tj-stack-more-cong[cong]: state.more s = state.more s' → tj-stack s =
tj-stack s'
by (cases s, cases s') simp

lemma [simp]:
s(state.more := (sccs := sc, lowlink := l, tj-stack := t))
= s(sccs := sc, lowlink := l, tj-stack := t)
by (cases s) simp

end

locale Tarjan-invar = Tarjan +
DFS-invar where param = tarjan-params

context Tarjan-def begin

lemma Tarjan-invar-eq[simp]:
DFS-invar G tarjan-params s ↔ Tarjan-invar G s (is ?D ↔ ?T)
proof
assume \( ?D \) then interpret \( \text{DFS-invar} \) where \( \text{param} = \text{tarjan-params} \).
show \( ?T \).
next
assume \( ?T \) then interpret \( \text{Tarjan-invar} \).
show \( ?D \).
qed
end

2.7.3 Correctness Proof

class \( \text{Tarjan} \)
begin

lemma \( \text{i-tj-stack-discovered} \):
\( \text{is-invar} \ (\lambda s. \text{set} (\text{tj-stack} s) \subseteq \text{dom} (\text{discovered} s)) \)
proof (induct rule: establish-invarI)
  case (finish \( s \))
  from finish show \( ?\)case
    apply simp
    unfolding \( \text{tarjan-fin-def} \)
    apply (refine-vcg \( \text{tj-stack-pop-set-leaf-rule} \))
    apply auto
    done
qed (auto simp add: \( \text{tarjan-disc-def} \) \( \text{tarjan-back-def} \))

lemmas (in \( \text{Tarjan-invar} \)) \( \text{tj-stack-discovered} = \)
\( \text{i-tj-stack-discovered}[\text{THEN make-invar-thm}] \)

lemma \( \text{i-tj-stack-distinct} \):
\( \text{is-invar} \ (\lambda s. \text{distinct} (\text{tj-stack} s)) \)
proof (induct rule: establish-invarI-ND)
  case (new-discover \( s \) \( s' \) \( v \))
  then interpret \( \text{Tarjan-invar} \) where \( s = s \) by simp
  from new-discover \( \text{tj-stack-discovered} \) have \( v \notin \text{set} (\text{tj-stack} s) \) by auto
  with new-discover show \( ?\)case by (simp add: \( \text{tarjan-disc-def} \))
next
  case (finish \( s \))
  thus \( ?\)case
    apply simp
    unfolding \( \text{tarjan-fin-def} \) \( \text{tj-stack-pop-def} \)
    apply (refine-vcg)
    apply (auto intro: distinct-tl)
    done
qed (simp-all add: \( \text{tarjan-back-def} \))

lemmas (in \( \text{Tarjan-invar} \)) \( \text{tj-stack-distinct} = \)
\( \text{i-tj-stack-distinct}[\text{THEN make-invar-thm}] \)

context begin interpretation \( \text{timing-syntax} \).
lemma \( \text{i-tj-stack-incr-disc} \):
\( \text{is-invar} \ (\lambda s. \forall k < \text{length} (\text{tj-stack} s). \forall j < k. \delta s (\text{tj-stack} s ! j) > \delta s (\text{tj-stack} s ! k)) \)
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 ∧ k>0 by (cases j=0) simp-all
  moreover with j < k have j' < k' by (simp add: j'-def k'-def)
  moreover from A have k' < 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 j < k 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
    by auto
ultimately have \( k' < \text{length} \ (\text{tj-stack } s) \) by (simp add: \( k' \)-def)

with \( \text{finish} \ (j' < k') \) have \( \delta \ s \ (\text{tj-stack } s \ ! \ k') < \delta \ s \ (\text{tj-stack } s \ ! \ j') \) by simp
also from \( \text{dropWhile-nth}(\text{OF} \ [\ ] \ !) \) have \( \text{tj-stack } s \ ! \ k' = \text{?dw} ! \ Suc k \)
  by (simp add: \( k' \)-def)
also from \( \text{dropWhile-nth}(\text{OF} \ [\ ] \ !) \) have \( \text{tj-stack } s \ ! \ j' = \text{?dw} ! \ Suc j \)
  by (simp add: \( j' \)-def)
also from \( \text{nth-tl}(\text{OF} \ [\ ] \ !) \) have \( \text{?dw} ! \ Suc k = a ! k \) by (simp add: \( A \)
finally have \( \delta \ s \ (a ! k) < \delta \ s \ (a ! j) \).

_aux = this
from \( \text{finish} \) show ?case
  apply simp
  unfolding \( \text{tarjan-fin-def} \ \text{tj-stack-pop-def} \)
  apply refine-vcg
  apply (auto intro!: aux)
  done
qed (simp-all add: \( \text{tarjan-back-def} \))
end end

context \( \text{Tarjan-invar} \) begin context begin interpretation

lemma \( \text{tj-stack-incr-disc} \):
  assumes \( k < \text{length} \ (\text{tj-stack } s) \)
  and \( j < k \)
  shows \( \delta \ s \ (\text{tj-stack } s \ ! \ j) > \delta \ s \ (\text{tj-stack } s \ ! \ k) \)
  using assms \( \text{i-tj-stack-incr-disc}[\text{THEN} \ \text{make-invar-thm}] \)
  by blast

lemma \( \text{tjs-disc-dw-tw} \):
  fixes \( u \)
  defines \( \text{dw} \equiv \text{dropWhile} \ ((\neq) \ u) \ (\text{tj-stack } s) \)
  defines \( \text{tw} \equiv \text{takeWhile} \ ((\neq) \ u) \ (\text{tj-stack } s) \)
  assumes \( x \in \text{set} \ \text{dw} \ y \in \text{set} \ \text{tw} \)
  shows \( \delta \ s \ x < \delta \ s \ y \)
  proof –
    from assms obtain \( k \) where \( k: \text{dw} ! k = x \ k < \text{length} \ \text{dw} \) by (metis \( \text{in-set-conv-nth} \))
    from assms obtain \( j \) where \( j: \text{tw} ! j = y \ j < \text{length} \ \text{tw} \) by (metis \( \text{in-set-conv-nth} \))
    have \( \text{length} \ (\text{tj-stack } s) = \text{length} \ \text{tw} + \text{length} \ \text{dw} \)
      by (simp add: \( \text{length-append}[(\text{symmetric})] \ \text{tw-def} \ \text{dw-def} \))
    with \( k \ j \) have \( \delta \ s \ (\text{tj-stack } s \ ! \ (k + \text{length} \ \text{tw})) < \delta \ s \ (\text{tj-stack } s \ ! \ j) \)
      by (simp add: \( \text{tj-stack-incr-disc} \))
    also from \( j \) takeWhile-nth have \( \text{tj-stack } s \ ! \ j = y \) by (metis \( \text{tw-def} \))
    also from \( \text{dropWhile-nth} \ k \) have \( \text{tj-stack } s \ ! \ (k + \text{length} \ \text{tw}) = x \) by (metis \( \text{tw-def} \ \text{dw-def} \))
    finally show ?thesis .
  qed

end
lemma i-sccs-finished-stack-ss-tj-stack:
  is-invar (λs. ∪(sccs s) ⊆ dom (finished s) ∧ set (stack s) ⊆ set (tj-stack s))

proof (induct rule: establish-invar1)
  case (finish s s' u) then interpret Tarjan-invar where
    s = s by simp

    let ?tw = takeWhile ((≠) u) (tj-stack s)
    let ?dw = dropWhile ((≠) u) (tj-stack s)

    \{
      fix x
      \assume A: x ≠ u \in set ?tw u \in set (tj-stack s)
      \hence x-tj: x \in set (tj-stack s) by (auto dest: set-takeWhileD)
    \}

    have x ∈ dom (finished s)
    proof (rule ccontr)
      \assume x \notin dom (finished s)
      with x-tj tj-stack-discovered discovered-eq-finished-un-stack have x ∈ set (stack s) by blast
      with tl-lt-stack-hd-discover finish have \*: δ s x < δ s u by simp
    \}

    from A have ?dw ≠ [] by simp
    with hd-dropWhile[OF this] hd-in-set have u ∈ set ?dw by metis
    with tjs-disc-dw-tw \{ x \in set ?tw \} have δ s u < δ s x by simp

    with \* show False by force
    qed

  \}

  note aux-scc = this

  \{
  fix x
  \assume A: x ∈ set (tl (stack s)) u ∈ set (tj-stack s)
  with finish stack-distinct have x ≠ u by (cases stack s) auto

  moreover
  from A have x ∈ set (stack s) by (metis in-set-tlD)
  with stack-not-finished have x \notin dom (finished s) by simp
  with A aux-scc[OF x ≠ u] have x \notin set ?tw by blast

  moreover
  from finish \{ x ∈ set (stack s) \} have x ∈ set (tj-stack s) by auto

  moreover note tjs-union[OF \{ u ∈ set (tj-stack s) \}]

  ultimately have x ∈ set (tl ?dw) by blast

200
lemma i-tj-stack-ss-stack-finished:
is-invar (\lambda s. set (tj-stack s) \subseteq set (stack s) \cup 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. dom (finished s) \subseteq \bigcup (sccs s) \cup 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 have v \notin dom (finished s) by auto
  with new-discover 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)

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

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

lemma sccs-finished:
  \bigcup (sccs s) \subseteq dom (finished s)
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 /∈ set (tj-stack s)))
proof (induct rule: establish-invarI-ND-CB)
case (new-discover s s' v) then interpret Tarjan-invar where s=s by simp
{
  fix x
  let ?s = s'

  assume TRANS: (∀Ψ. tarjan-disc v s' ≤n SPEC Ψ ⇒ Ψ 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 /∈ 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 iffD2])
    apply clarsimp-all
    done
  with new-discover r have (∀ x ∈ scc. x /∈ set (tj-stack s') by simp

  ultimately have (∀ x∈scc. x /∈ set (tj-stack ?s) by (auto simp: new-discover)
}
with new-discover show ?case by (simp add: pw-leof-iff)
next
case (cross-back-edge s s' u v) then interpret Tarjan-invar where s=s by simp

{ fix x
  let ?s = s'⟨state.more := x⟩
  assume TRANS: ∀Ψ. tarjan-back u v s' ≤ₚ SPEC Ψ ⇒ Ψ x
  and r: scc-root ?s r scc r ∈ dom (finished s')
  with cross-back-edge have scc-root s r scc
    by (simp add: scc-root-transfer[where s'=?s])

  moreover
  have tj-stack ?s = tj-stack s by (rule TRANS) (simp add: cross-back-edge tarjan-back-def)

  ultimately have ∀x∈scc. x ∉ set (tj-stack ?s)
    using cross-back-edge r by simp
  }
with cross-back-edge show ?case by (simp add: pw-leof-iff)

next

  case (finish s s' u) then interpret Tarjan-invar where s=s by simp

  { fix x
    let ?s = s'⟨state.more := x⟩
    assume TRANS: ∀Ψ. tarjan-fin u s' ≤ₚ SPEC Ψ ⇒ Ψ 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 ∀x∈scc. x ∉ set (tj-stack ?s)
      proof (cases r = u)
        case False with finish r have ∀x∈scc. x ∉ set (tj-stack s)
          using scc-root-transfer[where s'=?s]
          by simp
        moreover have set (tj-stack ?s) ⊆ set (tj-stack s)
          apply (rule TRANS)
          unfolding tarjan-fin-def
          apply (refine-vcg tj-stack-pop-set-leof-rule)
          apply (simp-all add: finish)
          done
        ultimately show ?thesis by blast
      next
        case True with r' s'.scc-root-unique-is-scc have scc-root ?s u (scc-of E u)
        by simp
        with s'.scc-root-transfer[where s'=s'] finish have scc-root s' u (scc-of E u) by simp
        moreover
      }

next
hence [simp]: tj-stack ?s = tl (dropWhile ((/=) u) (tj-stack s))
apply (rule-tac TRANS)
unfolding tarjan-fin-def tj-stack-pop-def
apply (refine-vcg)
apply (simp-all add: finish)
done

{ let ?dw = dropWhile ((/=) u) (tj-stack s)
let ?tw = takeWhile ((/=) u) (tj-stack s)
fix x
define j::nat where j = 0
assume x: x ∈ set (tj-stack ?s)
then obtain i where i: i < length (tj-stack ?s) tj-stack ?s ! i = x
  by (metis in-set-conv-nth)

  have length (tj-stack s) = length ?tw + length ?dw
    by (simp add: length-append[symmetric])
  with i have δ s (tj-stack s ! (Suc i + length ?tw)) < δ s (tj-stack s ! length ?tw)
    by (simp add: tj-stack-incr-disc)

  also from hd-stack-in-tj-stack finish have: ?dw ≠ [] and length ?dw > 0 by simp-all
    from hd-dropWhile[OF ne] hd-cone-nth[OF ne] have: ?dw ! 0 = u by simp
    with dropWhile-nth[OF this(2)] have: tj-stack s ! length ?tw = u by simp

  also from i have: ?dw ! Suc i = x Suc i < length ?dw by (simp-all add: nth-tl[OF ne])
    with dropWhile-nth[OF this(2)] have: tj-stack s ! (Suc i + length ?tw) = x by simp

  finally have δ ?s x < δ ?s u by (simp add: finish)

  moreover from x s’.tj-stack-discovered have: x ∈ dom (discovered ?s) by auto
    ultimately have: x ∉ scc using s’.scc-root-disc-le r True by force
  } thus ?thesis by metis
qed

with finish show ?case by (simp add: pw-leaf-iff)
qed simp-all
end end

context Tarjan-invar begin
lemma no-finished-root:
  assumes scc-root s r scc

204
and \( r \in \text{dom} (\text{finished} s) \)
and \( x \in \text{scc} \)
shows \( x \notin \text{set} (\text{tj-stack} s) \)
using \text{assms}
using \text{i-no-finished-root}[\text{THEN make-invar-thm}] by \text{blast}

context begin interpretation \text{timing-syntax}.

lemma tj-stack-reach-stack:
assumes \( u \in \text{set} (\text{tj-stack} s) \)
shows \( \exists v \in \text{set} (\text{stack} s). (u,v) \in E^* \land \delta s v \leq \delta s u \)
proof –
  have \( u\text{-scc} \): \( u \in \text{scc-of} E u \) by \text{simp}
  from \text{assms} \text{tj-stack-discovered} have \( u\text{-disc} \): \( u \in \text{dom} (\text{discovered} s) \) by \text{auto}
  with \text{r-scc-root-of-node-exists}
  obtain \( r \) where \( r \): \( \text{scc-root} s r \) (\text{scc-of} E u) by \text{blast}
  have \( r \in \text{set} (\text{stack} s) \) proof (rule \text{ccontr})
    assume \( r \notin \text{set} (\text{stack} s) \)
    with \( \text{r[unfolded} \text{scc-root-def]} \text{stack-set-def} \) have \( r \in \text{dom} (\text{finished} s) \) by \text{simp}
    with \( u\text{-scc} \) have \( u \notin \text{set} (\text{tj-stack} s) \) using \text{i-no-finished-root} \text{by} \text{blast}
    with \text{assms}
    show \( \text{False} \) by \text{contradiction}
  qed
  moreover from \( r \text{ scc-reach-scc-root} u\text{-scc} u\text{-disc} \) \( (u,r) \in E^* \) by \text{blast}
  moreover from \( r \text{ scc-root-disc-le} u\text{-scc} u\text{-disc} \) \( \delta s r \leq \delta s u \) by \text{blast}
  ultimately show \( ?\text{thesis} \) by \text{metis}
qed

lemma tj-stack-reach-hd-stack:
assumes \( v \in \text{set} (\text{tj-stack} s) \)
shows \( (v, \text{hd} (\text{stack} s)) \in E^* \)
proof –
  from \text{tj-stack-reach-stack} \text{assms}
  obtain \( r \) where \( r \): \( \text{r scc-root-disc-le} u\text{-scc} u\text{-disc} \) \( \delta s r \leq \delta s u \) by \text{blast}
  with \text{r-trancl-eq-or-trancl}
  have \( ?\text{thesis} \) by \text{metis}
next
  from \( r \)
  have \( \text{ne:stack} s \neq [] \) by \text{auto}

  assume \( r \in \text{set} (\text{tl} \text{ (stack} s)) \)
  with \( \text{tl-stack-hd-tree-path ne} \) have \( (r, \text{hd} (\text{stack} s)) \in (\text{tree-edges} s)^+ \) by \text{simp}
  with \text{r-trancl-eq-or-trancl}
  have \( ?\text{thesis} \) by \text{metis}
qed
lemma empty-stack-imp-empty-tj-stack:
  assumes stack s = []
  shows tj-stack s = []
proof (rule ccontr)
  assume ne: tj-stack s ≠ []
  then obtain x where: x ∈ set (tj-stack s) by auto
  with tj-stack-reach-stack obtain r where: r ∈ set (stack s) by auto
  with assms show False by simp
qed

lemma stacks-eq-iff: stack s = [] ←→ tj-stack s = []
using empty-stack-imp-empty-tj-stack stack-ss-tj-stack
by auto
end

context Tarjan begin
context begin interpretation timing-syntax .

lemma i-sccs-are-sccs:
  is-invar (λs. ∀scc ∈ sccs s. is-scc E scc)
proof (induction rule: establish-invarI)
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 s′ ⪯n SPEC Ψ → 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)
      unfolding tarjan-fin-def tj-stack-pop-def
      by (refine-vcg) (simp-all add: False)
  }
thus \( \text{thesis by (simp add: finish)} \)

next

\[ \text{case } \text{True} \]

let \( \text{?dw} = \text{dropWhile } \((\neq) \text{ u}) (\text{tj-stack } s) \)

let \( \text{?tw} = \text{takeWhile } \((\neq) \text{ u}) (\text{tj-stack } s) \)

let \( \text{?tw}' = \text{insert } \text{u} \text{ (set } \text{?tw}) \)

have \( \text{[simp]: sccs } \text{?s} = \text{insert } \text{?tw}' \text{ (sccs s)} \)

apply (rule TRANS)

unfolding \text{tarjan-fin-def tj-stack-pop-def}

by (refine-vcg) (simp-all add: True)

have \( \text{[simp]: tj-stack } \text{?s} = \text{tl } \text{?dw} \)

apply (rule TRANS)

unfolding \text{tarjan-fin-def tj-stack-pop-def}

by (refine-vcg) (simp-all add: True)

from \text{True scc-root-transfer[where } \text{s'}=s'] have scc-root } \text{s } \text{u } \text{(scc-of E u)}

by simp

with \text{inv'} scc-root-transfer[where } \text{s'}=?s] \text{u-disc have u-root: scc-root } \text{?s } \text{u } \text{(scc-of E u)} \text{ by simp}

have \( \text{?tw}' \subseteq \text{scc-of E } \text{u} \)

proof

fix \text{v} assume \text{v: v } \in \text{?tw'}

show \text{v } \in \text{scc-of E } \text{u}

proof cases

assume \text{v } \neq \text{u with v have v: v } \in \text{set } \text{?tw} \text{ by auto}

hence \text{v-tj: v } \in \text{set } \text{(tj-stack s) by (auto dest: set-takeWhileD)}

with \text{tj-stack-discovered have v-disc: v } \in \text{dom } \text{(discovered s) by auto}

from \text{hd-stack-in-tj-stack finish have } \text{?dw } \neq [\text{ by simp}

with \text{hd-dropWhile[OF this] hd-in-set have u } \in \text{set } \text{?dw} \text{ by metis}

with \text{v have } \delta s v > \delta s u \text{ using tjs-disc-dw-tw} \text{ by blast}

moreover have \text{v } \in \text{dom } \text{(finished s)}

proof (rule ccontr)

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

with \text{v-disc stack-set-def have v: v } \in \text{set } \text{(stack s) by auto}

with \text{v:w finish have v: v } \in \text{set } \text{(tl } \text{(stack s)) by (cases stack s) auto}

with \text{tl-tt-stack-hd-discover finish have } \delta s v < \delta s u \text{ by simp}

with \text{\delta s v > \delta s w show False by force}

qed

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

using parenthesis-implemented-tree-path-not-finished[OF u-disc] u-n-fin

by force

with trancl-mono-mp tree-edges-ssE have \text{(u,v) } \in \text{E' by (metis rtrancl-eq-or-trancl)}
moreover have \( \text{is-scc} (\text{scc-of } E \ u) \ u \in \text{scc-of } E \ u \) by simp
ultimately show \?thesis using \text{is-scc-closed} by metis
qed simp

moreover have \( \text{scc-of } E \ u \subseteq ?tw' \)
proof
fix \( v \) assume \( v \in \text{scc-of } E \ u \)
moreover note \( u\)-root
ultimately have \( v \in \text{dom} (\text{finished } ?s) \) by simp
using \( s'.\text{scs-finished-impl-scc-finished} s'.\text{no-finished-root} \)
by auto
with \( s'.\text{finished-ss-sccs-tj-stack} \) have \( v \in \bigcup (\text{sccs } s) \) by blast
hence \( v \in \bigcup (\text{sccs } s) \lor v \in ?tw' \) by auto
thus \( v \in ?tw' \)
proof
assume \( v \in \bigcup (\text{sccs } s) \)
then obtain \( \text{scc} \) where \( \text{scc} \in \text{sccs } s \)
by auto
moreover with \( \text{finish} \) have \( \text{is-scc} \ E \ \text{scc} \) by simp
moreover have \( \text{is-scc} E (\text{scc-of } E \ u) \) by simp
moreover note \( v \)
ultimately have \( \text{scc} = \text{scc-of } E \ u \) using \( \text{is-scc-unique} \) by metis
hence \( u \in \text{scs} \) by simp
with \( \text{scc sccs-finished} \) have \( u \in \text{dom} (\text{finished } s) \) by auto
with \( \text{u-n-fin} \) show \?thesis by contradiction
qed simp
qed
ultimately have \( ?tw' = \text{scc-of } E \ u \) by auto
hence \( \text{is-scc } E ?tw' \) by simp
with \( \text{finish} \) show \?thesis by auto
qed
}

thus \( ?\text{case by (auto simp: pu-leof-iff \text{finish})} \)
qed (simp-all add: tarjan-back-def tarjan-disc-def)
end

lemmas (in Tarjan-invar) \( \text{sccs-are-sccs} = \)
\( \text{i-sccs-are-sccs}[\text{THEN make-invar-thm}] \)

context begin interpretation timing-syntax .

lemma \( \text{i-lowlink-eq-LowLink} : \)
\( \text{is-invar} (\lambda s. \forall x \in \text{dom} (\text{discovered } s). \zeta s x = \text{LowLink } s x) \)
proof –
{
\textbf{fix } s \ s' :: 'v \text{tarjan-state}\\
\textbf{fix } v \ w\\
\textbf{fix } x\\

\textbf{let } \ ?s = s'(\text{state}.\text{more} := x)\\

\textbf{assume } \text{pre-ll-sub-rev}: \forall w. \ (\text{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\}\\
\textbf{assume } \text{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)\})\\

\textbf{assume } \text{Tarjan-invar } G \ s\\
\textbf{assume } \text{simp}: \ \text{discovered } s' = (\text{discovered } s)(v \mapsto \text{counter } s)\\
\ \\ \ \text{lowlink } s' = \text{lowlink } s\\
\ \\ \ \text{cross-edges } s' = \text{cross-edges } s \land \text{back-edges } s' = \text{back-edges } s\\
\textbf{assume } \text{v-n-disc}: \ v \notin \text{dom}(\text{discovered } s)\\
\textbf{assume } \text{IH}: \ \forall w. \ w \in \text{dom}(\text{discovered } s) \implies \ (s' \ w = \text{LowLink } s \ w)\\

\textbf{assume } \text{TRANS}: \ \forall \Psi. \ \text{tarjan-disc } v \ s' \leq \Psi \ \text{SPEC } \Psi \implies \Psi \ x\\
\textbf{and } \text{INV}: \text{DFS-invar } G \text{ tarjan-params } ?s\\
\textbf{and } \text{w-disc}: \ w \in \text{dom}(\text{discovered } ?s)\\

\textbf{interpret } \text{Tarjan-invar where } s=s \text{ by fact}\\
\textbf{from } \text{INV } \text{interpret } s':\text{Tarjan-invar where } s=?s \text{ by simp}\\

\textbf{have } \text{simp}: \ \text{lowlink } ?s = (\text{lowlink } s)(v \mapsto \text{counter } s)\\
\ \ \ \ \text{by (rule TRANS) (auto simp: tarjan-disc-def)}\\

\textbf{from } \text{v-n-disc edge-imp-discovered have } \text{edges } s' \setminus \{v\} = \{\} \text{ by auto}\\
\textbf{with } \text{tree-sub tree-edge-imp-discovered have } \text{edges } ?s \setminus \{v\} = \{\} \text{ by auto}\\
\textbf{with } \text{s'.no-path-imp-no-lowlink-path have } \forall w. \neg\exists p. \ \text{lowlink-path } ?s \ v \ p \ w)\\
\text{by metis}\\
\textbf{hence } \text{ll-v: } \text{lowlink-set } ?s \ v = \{v\}\\
\textbf{unfolding } \text{lowlink-set-def by auto}\\

\textbf{have } ?s \ w = \text{LowLink } ?s \ w\\
\textbf{proof (cases } w=v)\\
\ \ \ \ \text{case True with } \text{ll-v show } ?\text{thesis by simp}\\
\textbf{next}\\
\ \ \ \ \text{case False hence } ?s \ w = ?s \ w \text{ by simp}\\
\textbf{also from } \text{IH have } ?s \ w = \text{LowLink } s \ w \text{ using } \text{w-disc False by simp}\\
\textbf{also have } \text{LowLink } s \ w = \text{LowLink } ?s \ w\\
\textbf{proof (rule LowLink-eq[OF INV])}\\
\textbf{from } \text{v-n-disc show } \text{discovered } s \subseteq \text{discovered } ?s \text{ by (simp add: map-le-def)}\\

\textbf{from } \text{tree-sub show } \text{lowlink-set } s \ w \subseteq \text{lowlink-set } ?s \ w\\
\textbf{unfolding } \text{lowlink-set-def lowlink-path-def}\\
\textbf{by auto}
show \text{lowlink-set} \ ?s w \subseteq \text{lowlink-set} s w \cup \{v\}
proof \ (\text{cases} \ w = v)
  case \ \text{True} \ with \ ll-v \ show \ \text{?thesis} \ by \ \text{auto}
next
  case \ \text{False} \ thus \ ?thesis
    using \ \text{pre-ll-sub-rev} \ w-disc \ \text{INV}
    by \ \text{simp}
qued

show \ w \in \text{dom} \ (\text{discovered} \ s) \ using \ \text{w-disc} \ \text{False} \ by \ \text{simp}

fix \ ll \ assume \ ll \in \{v\} \ with \ \text{timing-less-counter} \ \text{lowlink-set-discovered}
have \ \forall x. x \in \delta s \text{lowlink-set} s w \implies x < \delta ?s ll \ by \ \text{simp} \ \text{force}
moreover \ from \ \text{Min-in} \ \text{lowlink-set-finite} \ \text{lowlink-set-not-empty} \ w-disc \ \text{False}
have \ \text{LowLink} s w \in \delta s \text{lowlink-set} s w \ by \ \text{auto}
ultimately \ show \ \text{LowLink} s w \leq \delta ?s ll \ by \ \text{force}
qued
finally \ show \ ?thesis .
qued
\}

\note \ \text{tarjan-disc-aux} = \text{this}

show \ ?thesis
proof \ (\text{induct rule: establish-invarI-CB})
  case \ (\text{new-root} s s' v0)
  \{
    fix \ w x
    let \ ?s = \text{new-root} v0 s[\text{state}.more := x]
    have \ \text{lowlink-set} ?s w \subseteq \text{lowlink-set} s w \cup \{v0\}
      unfolding \ \text{lowlink-set-def} \ \text{lowlink-path-def}
      by \ \text{auto}
  \}
  \note \ \ast = \text{this}

from \ \text{new-root} \ show \ ?case
  using \ \text{tarjan-disc-aux} \[OF \ \ast]
  by \ (\text{auto simp add: pw-leof-iff})
next
  case \ (\text{discover} s s' u v) \ \text{then interpret} \ \text{Tarjan-invar} \ \text{where} \ s=s \ by \ \text{simp}
    let \ ?s' = \text{discover} (hd (\text{stack} s)) \ v \ (s[\text{pending} := \text{pending} s - \{(hd (\text{stack} s),v)\}])
  \{
    fix \ w x
    let \ ?s = ?s'[\text{state}.more := x]
    assume \ \text{INV: Tarjan-invar} \ G ?s
      and \ d: \ w \in \text{dom} \ (\text{discovered} ?s')
      and \ w \neq v
  \}

210
interpret s': Tarjan-invar where s=?s by fact

have lowlink-set ?s w ⊆ lowlink-set s w ∪ {v}
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)
thus ll ∈ lowlink-set s w ∪ {v} (is ll ∈ ?L)
proof
assume ll = w with d show ?thesis by (auto simp add: lowlink-set-def)
next
assume ∃ p. lowlink-path ?s w p ll
then guess p .. note p = this

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 ?thesis
proof (rule tri-caseE)
  assume v≠ll v ∉ set p hence lowlink-path s w p ll
  using p by (auto simp add: lowlink-path-def)
  with ll show ?thesis by (auto simp add: lowlink-set-def)
next
assume v = ll thus ?thesis by simp
next
assume v ∈ set p v ≠ ll
then obtain i where i: i < length p p!i = v
  by (metis in-set-conv-nth)
have False
proof (cases i)
  case 0 with i have hd p = v by (simp add: hd-conv-nth)
  with ⟨hd p = w : w ≠ v⟩ show False by simp
next
  case (Suc n) with i s'.lowlink-path-finished[OF p, where j=i] have
v ∈ dom (finished ?s) by simp
  with finished-discovered discover show False by auto
qed
thus ?thesis..
qed
qed
qed

} note * = this

from discover hd-in-set stack-set-def have v ≠ u by auto
with discover have **: tree-edges ?s' = tree-edges s ∨ (∃ u. u ≠ v ∧ tree-edges ?s' = tree-edges s ∪ {(u,v)}) by auto

211
from discover show \texttt{?case}
using tarjan-disc-aux[\texttt{OF \ast \ast}]
by (\texttt{auto simp: pw-leaf-if})

next
case (cross-back-edge s s' u v) then interpret Tarjan-invar where \(s = s\) by simp

from cross-back-edge have [simp]:
  discovered s' = discovered s
  finished s' = finished s
  tree-edges s' = tree-edges s
  lowlink s' = lowlink s
  by simp-all
{
  fix \(w :: \texttt{'v}\)
  fix \(x\)

  let \(\?s = s'\langle\texttt{state.more := x}\rangle\)
  let \(\?L = \delta \ s' \cdot \text{lowlink-set} \ s \ w\)
  let \(\?L' = \delta \ ?s \cdot \text{lowlink-set} \ ?s \ w\)

  assume \texttt{TRANS}: \(\forall \Psi. \text{tarjan-back } u \ v \ s' \leq_n \texttt{SPEC } \Psi = \Rightarrow \Psi \ x\)
  and \texttt{inv'}: DFS-invar \(G\) tarjan-params \(?s\)
  and \texttt{w-disc'}: \(w \in \texttt{dom (discovered } \texttt{?s)}\)

  from \texttt{inv'} interpret \(s'\cdot\text{Tarjan-invar where}\ s = ?s\ by simp

  have ll-sub: lowlink-set s w \subseteq lowlink-set ?s w
    unfolding lowlink-set-def lowlink-path-def
    by (auto simp: cross-back-edge)

  have ll-sub-rev: lowlink-set ?s w \subseteq lowlink-set s w \cup \{v\}
    unfolding lowlink-set-def lowlink-path-def
    by (auto simp: cross-back-edge)

  from \texttt{w-disc'} have w-disc: \(w \in \texttt{dom (discovered s)}\ \texttt{by simp}\)
  with \texttt{LowLink-le-disc} have \(LLw: \text{LowLink } s w \leq \delta \ s w\ \texttt{by simp}\)

  from cross-back-edge hd-in-set have u-n-fin: \(u \notin \texttt{dom (finished s)}\)
  using stack-not-finished by auto

  { assume \*: \(v \in \texttt{lowlink-set } ?s \ w = \Rightarrow \text{LowLink } s w \leq \delta \ ?s \ v\)
    have LowLink s w = LowLink ?s w
      proof (rule LowLink-eqI[\texttt{OF inv' - ll-sub ll-sub-rev w-disc}])
        show discovered s \subseteq_m discovered ?s \ by simp
          fix ll assume ll \in \{v\} ll \in lowlink-set ?s w
          with * show LowLink s w \leq \delta \ ?s \ ll \ by simp
      qed

  212
} note LL-eql = this

have $\zeta ?s w = \text{LowLink } ?s w$
proof (cases w=u)
case True show ?thesis
proof (cases ($\delta s v < \delta s w \land v \in \text{set } (tj-stack s) \land \delta s v < \zeta s w)$)
case False note all-False = this
with ($w = w$ have $\zeta ?s w = \zeta s w$
by (rule-tac TRANS) (auto simp add: tarjan-back-def cross-back-edge)
also from cross-back-edge w-disc have $\zeta w: ... = \text{LowLink } s w$ by simp
also have LowLink s w = LowLink ?s w
proof (rule LL-eql)
assume $v: v \in \text{lowlink-set } ?s w$
show LowLink s w $\leq \delta ?s v$
proof (cases $\delta s v < \delta s w \land \delta s v < \zeta s w$)
case False with $\langle \text{LowLink s w } \leq \delta s w \zeta w \rangle$
show ?thesis by auto
next
case True with all-False have v-n-tj: $v \notin \text{set } (tj-stack s)$ by simp
from $v$ have e: $(v,u) \in E^*(u,v) \in E^*$
unfolding lowlink-set-def by (auto simp add: $\langle w=u \rangle$)
from v-n-tj have $v \notin \text{set } (stack s)$ using stack-ss-tj-stack by auto
with cross-back-edge have $v \in \text{dom } (\text{finished s})$ by (auto simp add: stack-set-def)
with finished-ss-sccs-tj-stack v-n-tj sccs-are-sccs obtain scc
where scc: $v \in \text{scc } \text{scc } \in \text{sccs s is-ss } E \text{ scc by blast}$
with is-ss-closed e have u $\in \text{scc }$ by metis
with scc sccs-finished u-n-fin have False by blast
thus ?thesis ..
qed
qed
finally show ?thesis .

next
case True note all-True = this
with ($w=w$) 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)
with s'.lowlink-path-single have lowlink-path ?s u [u] v by auto
moreover from cb s'.cross-edges-ssE s'.back-edges-ssE have $(u,v) \in E$ by blast
hence $(u,v) \in E^*$ ..

213
moreover from all-True \( t \)-stack-reach-hd-stack have \((v,u) \in E^*\) by
\(\text{simp add: cross-back-edge}\)
moreover note \(v \in \text{dom (discovered } s)\)
ultimately show \(?\text{thesis}\) by \((\text{auto intro: } s'.\text{lowlink-set}\text{I simp: } \langle w = w \rangle)\)
\(\text{qed}\)
with \(ll\text{-sub} ll\text{-sub-rev} \) have \(\text{lowlink-set } s w = \text{lowlink-set } s w \cup \{v\}\) by \(\text{auto}\)
hence \(\text{Min (} ?L \cup \{\delta s v\}\rangle = \text{LowLink } s w\) by \(\text{simp}\)
finally show \(?\text{thesis}\) .
\(\text{qed}\)
next
case False — \(w \neq u\)
\(\text{hence } \zeta s w = \zeta s w\) by \((\text{rule-tac TRANS}) (\text{simp add: tarjan-back-def cross-back-edge})\)
also have \(\zeta s w = \text{LowLink } s w\) using \(w\text{-disc False}\) by \((\text{simp add: cross-back-edge})\)
also have \(\text{LowLink } s w = \text{LowLink } ?s w\) proof \((\text{rule LL-eqI})\)
assume \(v \in \text{lowlink-set } ?s w\)
thus \(\text{LowLink } s w \leq \delta s v\) using \(\text{LLw}\) proof cases
\(\text{assume } v \neq w\)
with \(v\) obtain \(p\) where \(p: \text{lowlink-path } ?s w \; p \; p\neq []\)
by \((\text{auto simp add: lowlink-set-def lowlink-path-def})\)
\(\text{hence } \text{hd } p = w\) by \((\text{auto simp add: lowlink-path-def path-hd})\)
show \(?\text{thesis}\)
proof \((\text{cases } u \in \text{set } p)\)
\(\text{case False with last-in-set } p\) \((\text{cross-back-edge})\) have \(\text{last } p \neq \text{hd (stack } s)\) by \(\text{force}\)
with \(p\) have \(\text{lowlink-path } s w \; p \; v\)
by \((\text{auto simp: cross-back-edge lowlink-path-def})\)
with \(v\) have \(v \in \text{lowlink-set } s w\)
by \((\text{auto intro: lowlink-setI simp: lowlink-set-def cross-back-edge})\)
thus \(?\text{thesis}\) by \(\text{simp}\)
next
case \(\text{True then obtain } i\) where \(i: i < \text{length } p \; p!i = u\)
by \((\text{metis in-set-cone-nth})\)
\(\text{have False}\)
proof \((\text{cases } i)\)
\(\text{case 0 with } i\) have \(\text{hd } p = u\) by \((\text{simp add: hd-cone-nth})\)
with \(\text{hd } p = w\) \(\langle w \neq w \rangle\) show \(\text{False}\) by \(\text{simp}\)
next
case \((\text{Suc } n)\) with \(i\) \(s'.\text{lowlink-path-finished}[OF } p(1), \text{ where } j=i\) have
\(u \in \text{dom (finished } s)\) by \(\text{simp}\)
with \(u\text{-n-fin}\) show \(?\text{thesis}\) by \(\text{simp}\)
\(\text{qed}\)
thus \(?\text{thesis}\) ..
qed
qed simp
qed
finally show \textit{thesis}.
qed
}

note aux = this

with \textit{cross-back-edge} show \textit{case} by (auto simp: pw-leaf-iff)

next

\textbf{case} (finish \textit{s s'} \textit{u}) \textbf{then interpret} \textit{Tarjan-invar} \textbf{where} \textit{s}=	extit{s} \textbf{by simp}

from finish \textbf{have} \textit{[simp]}:

\begin{itemize}
  \item discovered \textit{s'} = discovered \textit{s}
  \item finished \textit{s'} = (finished \textit{s})(w→counter \textit{s})
  \item tree-edges \textit{s'} = tree-edges \textit{s}
  \item back-edges \textit{s'} = back-edges \textit{s}
  \item cross-edges \textit{s'} = cross-edges \textit{s}
  \item lowlink \textit{s'} = lowlink \textit{s}
  \item \textit{tj-stack} \textit{s'} = \textit{tj-stack} \textit{s}
\end{itemize}

by simp-all

\textbf{from} finish \textbf{hd-in-set stack-discovered have} \textbf{u-disc}: \textit{u} \in \textit{dom} \textbf{(discovered \textit{s})} \textbf{by blast}

{\textbf{fix} \textit{w :: v}\textbf{fix} \textit{x}\n
\begin{itemize}
  \item let \textit{?s} = \textit{s'}\langle \textit{state}.more := \textit{x}⟩
  \item let \textit{?L} = \textit{lowlink-set \textit{s} \textit{w}}
  \item let \textit{?Lu} = \textit{lowlink-set \textit{s} \textit{u}}
  \item let \textit{?L'} = \textit{lowlink-set \textit{s} \textit{u}}
\end{itemize}

\textbf{assume} \textbf{TRANS}: \textstyle \forall \textit{\Psi}. \textit{tarjan-fin} \textit{u \textit{s} ≤ \textit{n SPEC} \textit{\Psi} \Rightarrow \textit{\Psi x}}

\textbf{and} \textbf{inv'}: \textit{DFS-invar \textit{G tarjan-params} \textit{?s}}

\textbf{and} \textbf{w-disc}: \textit{w} \in \textbf{dom} \textbf{(discovered \textit{?s})}

\textbf{from} \textit{inv'} \textbf{interpret} \textit{s'}:\textit{Tarjan-invar} \textbf{where} \textit{s}=?\textit{s} \textbf{by simp}

\textbf{have} \textbf{ll-sub}: \textit{lowlink-set \textit{s} \textit{w} ⊆ \textit{lowlink-set \textit{?s} \textit{w}}}

\textbf{unfolding} \textbf{lowlink-set-def lowlink-path-def}

\textbf{by} \textbf{auto}

\textbf{have} \textbf{ll-sub-rev}: \textit{lowlink-set \textit{?s} \textit{w} ⊆ \textit{lowlink-set \textit{s} \textit{w} ∪ lowlink-set \textit{s} \textit{u}}}

\textbf{proof}

\textbf{fix} \textit{ll}

\textbf{assume} \textit{ll}: \textit{ll} \in \textit{lowlink-set \textit{?s} \textit{w}}

\textbf{hence} \textit{ll} = \textit{w} ∨ (\exists \textit{p}. \textit{lowlink-path ?s \textit{p \textit{ll}}} \textbf{by} \textbf{(auto simp add: lowlink-set-def)}

\textbf{thus} \textit{ll} \in \textit{lowlink-set \textit{s} \textit{w} ∪ lowlink-set \textit{s} \textit{u}}

\textbf{proof (rule disjEl)}
assume \( ll = w \) with \( w\)-disc show \( ?thesis \) by (auto simp add: lowlink-set-def)

next
assume \( ll \neq w \)
assume \( \exists p. \) lowlink-path \( ?s w p ll \)
then guess \( p \).

note \( p = \) this

hence \([simp]: p\neq[]\) by (simp add: lowlink-path-def)

from \( p \) have \( \text{hd} p = w \) by (auto simp add: lowlink-path-def path-hd)

show \( ?thesis \)
proof (cases \( u \in \text{set} p \))
case False
hence \( \text{lowlink-path} s w p ll \)
using \( p \) by (auto simp add: lowlink-path-def)
with \( ll \) show \( ?thesis \) by (auto simp add: lowlink-set-def)
next
case True
then obtain \( i \) where \( i: i < \text{length} p \)
by (metis in-set-conv-nth)
moreover
let \( ?dp = \) drop \( i \) \( p \) from \( i \)

have \( ?dp \neq [\] \) by simp

from \( i \)

have \( \text{hd} ?dp = u \) by (simp add: hd-drop-conv-nth)
moreover from \( i \)

have \( \text{last} ?dp = \text{last} p \) by simp
moreover {
fix \( k \)
assume \( 1 < \text{length} ?dp \)
and \( k < \text{length} ?dp - 1 \)
hence \( l: l < \text{length} p \)
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 \( i+k \leq \text{length} p \)
ultimately have \( i+\text{Suc} k \leq \text{length} p \) by simp-all

ultimately have \( (?dp! k, ?dp! \text{Suc} k) \in \text{tree-edges} s \) by (simp add: add.commute)
}

note \( aux = \) this

moreover {
assume \( *: l < \text{length} ?dp \)
hence \( l + i < \text{length} p \) by simp
with \( s'.\text{lowlink-path-finished}(OF \ p) \)

have \( p! (1+i) \in \text{dom} \) (finished ?s) by auto

moreover from \( l \)

ultimately have \( ?dp! l \in \text{dom} \) (finished ?s) by simp
moreover from \( aux[of 0] \) * have \( (?dp! 0, ?dp! \text{Suc} 0) \in \text{tree-edges} s \)
by simp

by simp 216
with \( \text{hd} \ ?dp = w \ \text{hd-cone-nth}(\text{of} \ ?dp) \) * have \((u, ?dp! \text{Suc} \ 0) \in\) tree-edges s by simp
with no-self-loop-in-tree have \( ?dp!1 \neq u \) by auto
ultimately have \( ?dp!1 \in \text{dom} \ (\text{finished} \ s) \) by simp
\}
moreover
from \( p \) have \( P: \text{path} \ E \ w \ p \ ll \) by (simp add: lowlink-path-def)

have \( p = (\text{take} \ i \ p)@dp \) by simp
with \( P \ \text{path-conc-conv} \) obtain \( x \) where \( p' = \text{path} \ E \ x \ ?dp \ ll \) path \( E \ w \) (take \( i \) \( p \) \( x \)) by metis
with \( ?dp \neq [] \) path-hd have \( \text{hd} \ ?dp = x \) by metis
with \( \text{hd} \ ?dp = w \) \( p' \) have \( u\)-path: \( \text{path} \ E \ u \ ?dp \ ll \) and \( \text{path-}u\): \( \text{path} \ E \ w \) (take \( i \) \( p \) \( u \)) by metis+
ultimately have \( \text{lowlink-path} \ s \ u \ ?dp \ ll \) using \( p \) by (simp add: lowlink-path-def)
moreover from \( \text{u-path} \ \text{path-is-trancl} \ ?dp \neq [] \) have \((u, ll) \in E^+\) by force
moreover \{ from \( ll \ \neq w \) have \((ll, w) \in E^+\) by (auto simp add: lowlink-set-def)
also from \( \text{path-}u \ \text{path-is-rtrancl} \) have \((w, u) \in E^*\) by metis
finally have \((ll, u) \in E^+\).
\}
moreover note \( ll \ u\)-disc
ultimately have \( ll \in \text{lowlink-set} \ s \ u \) unfolding lowlink-set-def by auto
thus \( \text{thesis} \) by auto
qed
qed
hence \( ll\)-sub-rev': \( ?L' \subseteq ?L \cup ?Lu \) by auto

have ref-ne: \( \text{stack} \ ?s \neq [] \implies \text{lowlink} \ ?s = \text{lowlink} \ ?s \ (\text{hd} \ (\text{stack} \ ?s)) \ (\zeta \ s \ (\text{hd} \ (\text{stack} \ ?s))) \ (\zeta \ s \ u)) \)
apply (rule TRANS)
unfolding tarjan-fin-def tj-stack-pop-def
by refine-vcg simp-all

have ref-e: \( \text{stack} \ ?s = [] \implies \text{lowlink} \ ?s = \text{lowlink} \ ?s \)
apply (rule TRANS)
unfolding tarjan-fin-def tj-stack-pop-def
by refine-vcg simp-all

have ref-tj: \( \zeta \ s \ u \neq \delta \) \( s \ u \implies \text{tj-stack} \ ?s = \text{tj-stack} \ s \)
apply (rule TRANS)
unfolding tarjan-fin-def tj-stack-pop-def
by refine-vcg simp-all

217
have \( \zeta \ ?s w = \text{LowLink} \ ?s w \)

proof (cases \( w = \text{hd} (\text{stack} \ ?s) \land \text{stack} \ ?s \neq [] \))

case True note all-True = this

with ref-ne have \( \ast: \zeta \ ?s w = \text{min} (\zeta \ s w) (\zeta \ s u) \) by simp

show \( \ast \)

proof (cases \( \zeta \ s u < \zeta \ s w \))

case False with \( \ast \)

fix \( ll \) assume \( ll \in \text{lowlink-set} \ s u \)

hence \( \text{LowLink} \ s u \leq \delta \ s ll \) by simp

moreover from \( \ast \) finish \( w\)-disc \( u\)-disc have \( \text{LowLink} \ s w \leq \text{LowLink} \ s u \)

ultimately have \( \ast \)

qed simp

finally show \( \ast \)

next

case True note \( \zeta \) rel = this

have \( \text{LowLink} \ s u \in \ ?L' \)

proof

−

from all-True finish \( w\)-tl have \( w\)-set \( (tl (\text{stack} \ s)) \) by auto

obtain \( ll \) where \( ll: ll \in \text{lowlink-set} \ s u \delta \ s ll = \text{LowLink} \ s u \)

using Min-in[of \( ?L' \)] lowlink-set-finite lowlink-set-not-empty w-disc

by fastforce

have \( ll \in \text{lowlink-set} \ ?s w \)

proof (cases \( \delta \ s u = \zeta \ s u \))

case True

moreover from \( w\)-tl finish \( ll\)-lt-stack-discover \( \delta \ s w < \delta \ s u \)

by simp

moreover from \( w\)-disc have \( \text{LowLink} \ s w \leq \delta \ s w \) by (simp add: LowLink-le-disc)

with \( w\)-disc finish have \( \zeta \ s w \leq \delta \ s w \) by simp

moreover note \( \zeta \) rel

ultimately have \( \ast \) by force

thus \( \ast \)

next

case False with \( u\)-disc finish \( ll \) have \( u \neq ll \) by auto

with \( ll \) have \( e: (ll,u) \in E^+ \ (u,ll) \in E^+ \) and

\( p: \exists p. \ \text{lowlink-path} \ s u p \ ll \) and

\( ll\)-disc: \( ll \in \text{dom} (\text{discovered} \ s) \)

by (auto simp: lowlink-set-def)

from \( p \) have \( p': \exists p. \ \text{lowlink-path} \ ?s u p \ ll \)

unfolding lowlink-path-def

by auto

218
from w-tl ll-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(l)
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 (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 (cases stack ?s)
  auto
with s'.il-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 ≤ 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
hence Min (?L ∪ ?Lu) = LowLink s u
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 w-disc finish * have LowLink s u = ζ ?s w by simp
finally show ?thesis ..
qed
next

219
case False note all-False = this
have ζ?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 ⟨v ≠ w⟩ obtain p where p: lowlink-path ?s w p v by (auto simp add: lowlink-path-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 by (auto simp add: lowlink-set-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 ≠ w 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 ∉ dom (finished s) by (metis
moreover from * no-self-loop-in-tree have p!n ≠ u by blast
ultimately have p!n ∈ set (stack ?s) using stack-set-def finish by
(cases stack s) auto
hence s-ne: stack ?s ≠ [] by auto
with all-False have w ≠ hd (stack ?s) by simp
from stack-is-tree-path finish obtain v0 where
path (tree-edges s) v0 (rev (stack ?s)) u
by auto
with s-ne have (hd (stack ?s), u) ∈ tree-edges s by (auto simp: neg-Nil-conv path-simps)
with * tree-eq-rule have **: hd (stack ?s) = p!n by simp
show ?thesis
proof (cases n)
case 0 with * have hd p = p!n by (simp add: hd-conv-nth)
with (hd p = w) ** have w = hd (stack ?s) by simp
with (w≠hd (stack ?s)): show False ..
next
case (Suc m) with ** s'.lowlink-path-finished[OF p, where j=n]
have hd (stack ?s) ∈ dom (finished ?s) by simp
with hd-in-set[OF s-ne] s'.stack-not-finished show ?thesis by blast
qed
qed
qed
qed
} 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-leof-iff)
qed simp-all
qed
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 ∈ dom (finished s) ∨ (stack s ≠ [] ∧ v = hd (stack s) ∧ pending s
  " {v} = {} )
  shows ζ s v = δ s v " " scc-root s v (scc-of E v)
  proof
    from assms have v ∈ dom (discovered s) using finished-discovered hd-in-set
    stack-discovered by blast
    qed
hence $\zeta s v = \text{LowLink} s v$ using lowlink-eq-LowLink by simp
with LowLink-eq-disc-iff-scc-root[OF assms] show ?thesis by simp
qed

lemma nc-sccs-eq-reachable:
  assumes NC: $\neg \text{cond } s$
  shows reachable $= \bigcup (\text{sccs } s)$
proof
  from nc-finished-eq-reachable NC have [simp]: reachable $= \text{dom} (\text{finished } s)$ by simp
  with sccs-finished show $\bigcup (\text{sccs } s) \subseteq \text{reachable}$ by simp

  from NC have stack s = [] by (simp add: cond-alt)
  with stacks-eq-iff have tj-stack s = [] by simp
  with finished-ss-sccs-tj-stack show reachable $\subseteq \bigcup (\text{sccs } s)$ by simp
qed

end

context Tarjan begin

lemma tarjan-fin-nofail:
  assumes pre-on-finish u s'
  shows nofail (tarjan-fin u s) by simp
proof
  from assms obtain s where s: DFS-invar G tarjan-params s stack s $\neq []$ u
    = hd (stack s) s' = 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 $\in$ set (tj-stack s') by simp
  moreover from s tj-stack-distinct have distinct (tj-stack s') by simp
  moreover have the (lowlink s' u) = the (discovered s' u) $\iff$ scc-root s' u (scc-of E u)
    proof
      from s have the (lowlink s' u) = the (discovered s' u) $\iff$ the (lowlink s u)
        = (discovered s u) by simp
      also from s lowlink-eq-disc-iff-scc-root have ... $\iff$ scc-root s u (scc-of E u)
      by blast
      also from s scc-root-transfer'[where s' = s'] have ... $\iff$ scc-root s' u (scc-of E u)
      by simp
    finally show ?thesis .
  qed
ultimately show ?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)

222
interpretation  
tarjan: Tarjan-def for $G$.

2.7.4 Interface

**definition** tarjan $G \equiv$ do 

```
ASSERT (fb-graph G);
s ← tarjan.it-dfs TYPE('a) G;
RETURN (seccs s)
```

**definition** tarjan-spec $G \equiv$ do 

```
ASSERT (fb-graph G);
SPEC (\lambda seccs. (\forall scc \in seccs. is-scc (g-E G) scc)
∧ \bigcup seccs = tarjan.reachable TYPE('a) G)
```

**lemma** tarjan-correct:

tarjan $G \leq$ tarjan-spec $G$

**unfolding** tarjan-def tarjan-spec-def

**proof** (refine-vcg lc-ASSERTI order-trans[OF DFS.it-dfs-correct])

```
assume fb-graph G
then interpret fb-graph G.
interpret Tarjan ..
show DFS G (tarjan.tarjan-params TYPE('b) G) ..
next
fix s
assume C: DFS-invar G (tarjan.tarjan-params TYPE('b) G) s ∧ ¬ tarjan.cond
TYPE('b) G s
then interpret Tarjan-invar G s by simp
```

```
from seccs-are-seccs show \forall secc \in seccs s. is-scc (g-E G) scc .
```

```
from nc-seccs-eq-reachable C show \bigcup (seccs s) = tarjan.reachable TYPE('b) G by simp
```

qed

end