

Depth-First Search

Toshiaki Nishihara Yasuhiko Minamide

September 13, 2023

Abstract

Depth-first search of a graph is formalized with `function`. It is shown that it visits all of the reachable nodes from a given list of nodes. Executable ML code of depth-first search is obtained with code generation feature of Isabelle/HOL. The formalization contains two implementations of depth-first search: one by stack and one by nested recursion. They are shown to be equivalent. The termination condition of the version with nested-recursion is shown by the method of inductive invariants.

Contents

1	Depth-First Search	1
1.1	Definition of Graphs	1
1.2	Depth-First Search with Stack	2
1.3	Depth-First Search with Nested-Recursion	2
1.4	Basic Properties	3
1.5	Correctness	5
1.6	Executable Code	5

1 Depth-First Search

```
theory DFS
imports Main
begin
```

1.1 Definition of Graphs

```
typedcl node
type-synonym graph = (node * node) list

primrec nexts :: [graph, node] ⇒ node list
where
  nexts [] n = []
| nexts (e#es) n = (if fst e = n then snd e # nexts es n else nexts es n)
```

definition *nextss* :: [graph, node list] \Rightarrow node set
 where *nextss* *g xs* = set *g* “ set *xs*

lemma *nexts-set*: $y \in \text{set } (\text{nexts } g \ x) = ((x,y) \in \text{set } g)$
 by (induct *g*) auto

lemma *nextss-Cons*: $\text{nextss } g \ (x\#xs) = \text{set } (\text{nexts } g \ x) \cup \text{nextss } g \ xs$
 unfolding *nextss-def* by (auto simp add:Image-def *nexts-set*)

definition *reachable* :: [graph, node list] \Rightarrow node set
 where *reachable* *g xs* = (set *g*)^{*} “ set *xs*

1.2 Depth-First Search with Stack

definition *nodes-of* :: graph \Rightarrow node set
 where *nodes-of* *g* = set (map fst *g* @ map snd *g*)

lemma [*simp*]: $x \notin \text{nodes-of } g \Longrightarrow \text{nexts } g \ x = []$
 by (induct *g*) (auto simp add: *nodes-of-def*)

lemma [*simp*]: finite (*nodes-of* *g* - set *ys*)

proof(rule *finite-subset*)
 show finite (*nodes-of* *g*)
 by (auto simp add: *nodes-of-def*)
qed (auto)

function

dfs :: graph \Rightarrow node list \Rightarrow node list \Rightarrow node list
 where
 dfs-base: *dfs* *g* [] *ys* = *ys*
 | *dfs-inductive*: *dfs* *g* (*x*\#*xs*) *ys* = (if List.member *ys* *x* then *dfs* *g* *xs* *ys*
 else *dfs* *g* (*nexts* *g* *x*@*xs*) (*x*\#*ys*))
 by *pat-completeness* auto

termination

apply (relation *inv-image* (*finite-psubset* <*lex*> *less-than*)
 ($\lambda(g,xs,ys). (\text{nodes-of } g - \text{set } ys, \text{size } xs)$))

apply *auto*[1]

apply (*simp-all* add: *finite-psubset-def*)

by (*case-tac* $x \in \text{nodes-of } g$) (auto simp add: *List.member-def*)

- The second argument of *dfs* is a stack of nodes that will be visited.
- The third argument of *dfs* is a list of nodes that have been visited already.

1.3 Depth-First Search with Nested-Recursion

function

$dfs2 :: graph \Rightarrow node\ list \Rightarrow node\ list \Rightarrow node\ list$

where

$dfs2\ g\ []\ ys = ys$

| $dfs2$ -inductive:

$dfs2\ g\ (x\#\!xs)\ ys = (if\ List.member\ ys\ x\ then\ dfs2\ g\ xs\ ys$
 $else\ dfs2\ g\ xs\ (dfs2\ g\ (nexts\ g\ x)\ (x\#\!ys)))$

by pat -completeness $auto$

lemma $dfs2$ -invariant: $dfs2$ -dom $(g, xs, ys) \implies set\ ys \subseteq set\ (dfs2\ g\ xs\ ys)$

by (induct $g\ xs\ ys$ rule: $dfs2$.pinduct) (force simp add: $dfs2$.psimps)+

termination $dfs2$

apply (relation inv -image (finite-psubset $<*\!lex*\!>$ less-than)

$(\lambda(g,xs,ys). (nodes\ of\ g - set\ ys, size\ xs)))$)

apply $auto[1]$

apply (simp-all add: finite-psubset-def)

apply (case-tac $x \in nodes\ of\ g$)

apply (auto simp add: List.member-def)[2]

by (insert $dfs2$ -invariant) force

lemma dfs -app: $dfs\ g\ (xs\@\!ys)\ zs = dfs\ g\ ys\ (dfs\ g\ xs\ zs)$

by (induct $g\ xs\ zs$ rule: dfs .induct) $auto$

lemma $dfs2\ g\ xs\ ys = dfs\ g\ xs\ ys$

by (induct $g\ xs\ ys$ rule: $dfs2$.induct) (auto simp add: dfs -app)

1.4 Basic Properties

lemma $visit$ -subset- dfs : $set\ ys \subseteq set\ (dfs\ g\ xs\ ys)$

by (induct $g\ xs\ ys$ rule: dfs .induct) $auto$

lemma $next$ -subset- dfs : $set\ xs \subseteq set\ (dfs\ g\ xs\ ys)$

proof(induct $g\ xs\ ys$ rule: dfs .induct)

case(2 $g\ x\ xs\ ys$)

show ?case

proof(cases $x \in set\ ys$)

case True

have $set\ ys \subseteq set\ (dfs\ g\ xs\ ys)$

by (rule $visit$ -subset- dfs)

with 2 **and** True **show** ?thesis

by (auto simp add: List.member-def)

next

case False

have $set\ (x\#\!ys) \subseteq set\ (dfs\ g\ (nexts\ g\ x\ @\ xs)\ (x\#\!ys))$

by(rule $visit$ -subset- dfs)

with 2 **and** *False* **show** *?thesis*
by (*auto simp add: List.member-def*)
qed
qed(*simp*)

lemma *nextss-closed-dfs'*[*rule-format*]:
 $nextss\ g\ ys \subseteq set\ xs \cup set\ ys \longrightarrow nextss\ g\ (dfs\ g\ xs\ ys) \subseteq set\ (dfs\ g\ xs\ ys)$
by (*induct g xs ys rule:dfs.induct, auto simp add:nextss-Cons List.member-def*)

lemma *nextss-closed-dfs*: $nextss\ g\ (dfs\ g\ xs\ []) \subseteq set\ (dfs\ g\ xs\ [])$
by (*rule nextss-closed-dfs', simp add: nextss-def*)

lemma *Image-closed-trancl*: **assumes** $r\ ''\ X \subseteq X$ **shows** $r^*\ ''\ X = X$
proof
show $r^*\ ''\ X \subseteq X$
proof –
{
 fix $x\ y$
 assume $y: y \in X$
 assume $(y,x) \in r^*$
 then have $x \in X$
 by (*induct*) (*insert assms y, auto simp add: Image-def*)
}
then show *?thesis* **unfolding** *Image-def* **by** *auto*
qed
qed *auto*

lemma *reachable-closed-dfs*: $reachable\ g\ xs \subseteq set(dfs\ g\ xs\ [])$
proof –
 have $reachable\ g\ xs \subseteq reachable\ g\ (dfs\ g\ xs\ [])$
 unfolding *reachable-def* **by** (*rule Image-mono*) (*auto simp add: next-subset-dfs*)
 also have $\dots = set(dfs\ g\ xs\ [])$
 unfolding *reachable-def*
 proof (*rule Image-closed-trancl*)
 from *nextss-closed-dfs*
 show $set\ g\ ''\ set\ (dfs\ g\ xs\ []) \subseteq set\ (dfs\ g\ xs\ [])$
 by (*simp add: nextss-def*)
qed
 finally show *?thesis* .
qed

lemma *reachable-nexts*: $reachable\ g\ (nexts\ g\ x) \subseteq reachable\ g\ [x]$
unfolding *reachable-def*
by (*auto intro: converse-rtrancl-into-rtrancl simp: nexts-set*)

lemma *reachable-append*: $reachable\ g\ (xs\ @\ ys) = reachable\ g\ xs \cup reachable\ g\ ys$
unfolding *reachable-def* **by** *auto*

```

lemma dfs-subset-reachable-visit-nodes:  $set (dfs\ g\ xs\ ys) \subseteq reachable\ g\ xs \cup set\ ys$ 
proof(induct g xs ys rule: dfs.induct)
  case 1
  then show ?case by simp
next
  case (2 g x xs ys)
  show ?case
  proof (cases x ∈ set ys)
    case True
    with 2 show  $set (dfs\ g\ (x\#\!xs)\ ys) \subseteq reachable\ g\ (x\#\!xs) \cup set\ ys$ 
    by (auto simp add: reachable-def List.member-def)
  next
  case False
  have  $reachable\ g\ (nexts\ g\ x) \subseteq reachable\ g\ [x]$ 
  by (rule reachable-nexts)
  hence a:  $reachable\ g\ (nexts\ g\ x\ @\ xs) \subseteq reachable\ g\ (x\#\!xs)$ 
  by(simp add: reachable-append, auto simp add: reachable-def)
  with False 2
  show  $set (dfs\ g\ (x\#\!xs)\ ys) \subseteq reachable\ g\ (x\#\!xs) \cup set\ ys$ 
  by (auto simp add: reachable-def List.member-def)
  qed
qed

```

1.5 Correctness

theorem *dfs-eq-reachable*: $set (dfs\ g\ xs\ []) = reachable\ g\ xs$

proof

```

  have  $set (dfs\ g\ xs\ []) \subseteq reachable\ g\ xs \cup set\ []$ 
  by (rule dfs-subset-reachable-visit-nodes[of g xs []])
  thus  $set (dfs\ g\ xs\ []) \subseteq reachable\ g\ xs$ 
  by simp
qed(rule reachable-closed-dfs)

```

theorem $y \in set (dfs\ g\ [x]\ []) = ((x,y) \in (set\ g)^*)$

by(*simp only:dfs-eq-reachable reachable-def, auto*)

1.6 Executable Code

consts *Node* :: *int* \Rightarrow *node*

code-datatype *Node*

instantiation *node* :: *equal*

begin

definition *equal-node* :: *node* \Rightarrow *node* \Rightarrow *bool*

where

[*code del*]: *equal-node* = *HOL.eq*

```
instance proof  
qed (simp add: equal-node-def)  
  
end  
  
declare [[code abort: HOL.equal :: node ⇒ node ⇒ bool]]  
  
export-code dfs dfs2 in SML file ⟨dfs.ML⟩  
  
end
```