Slicing Guarantees Information Flow
Noninterference

Daniel Wasserrab
August 16, 2018

Abstract

In this contribution, we show how correctness proofs for intra- [8] and interprocedural slicing [9] can be used to prove that slicing is able to guarantee information flow noninterference. Moreover, we also illustrate how to lift the control flow graphs of the respective frameworks such that they fulfil the additional assumptions needed in the noninterference proofs. A detailed description of the intraprocedural proof and its interplay with the slicing framework can be found in [10].

1 Introduction

Information Flow Control (IFC) encompasses algorithms which determines if a given program leaks secret information to public entities. The major group are so called IFC type systems, where well-typed means that the respective program is secure. Several IFC type systems have been verified in proof assistants, e.g. see [1, 2, 5, 3, 7].

However, type systems have some drawbacks which can lead to false alarms. To overcome this problem, an IFC approach basing on slicing has been developed [4], which can significantly reduce the amount of false alarms. This contribution presents the first machine-checked proof that slicing is able to guarantee IFC noninterference. It bases on previously published machine-checked correctness proofs for slicing [8, 9]. Details for the intraprocedural case can be found in [10].

2 HRB Slicing guarantees IFC Noninterference

theory NonInterferenceInter
  imports HRB-Slicing,FundamentalProperty
begin
2.1 Assumptions of this Approach

Classical IFC noninterference, a special case of a noninterference definition using partial equivalence relations (per) [6], partitions the variables (i.e. locations) into security levels. Usually, only levels for secret or high, written \( H \), and public or low, written \( L \), variables are used. Basically, a program that is noninterferent has to fulfill one basic property: executing the program in two different initial states that may differ in the values of their \( H \)-variables yields two final states that again only differ in the values of their \( H \)-variables; thus the values of the \( H \)-variables did not influence those of the \( L \)-variables.

Every per-based approach makes certain assumptions: (i) all \( H \)-variables are defined at the beginning of the program, (ii) all \( L \)-variables are observed (or used in our terms) at the end and (iii) every variable is either \( H \) or \( L \). This security label is fixed for a variable and can not be altered during a program run. Thus, we have to extend the prerequisites of the slicing framework in [9] accordingly in a new locale:

locale NonInterferenceInterGraph =
  SDG sourcenode targetnode kind valid-edge Entry
get-proc get-return-edges procs Main Exit Def Use ParamDefs ParamUses
for sourcenode :: 'edge \( \Rightarrow \) 'node and targetnode :: 'edge \( \Rightarrow \) 'node
and kind :: 'edge \( \Rightarrow \) ('var,'val,'ret,'pname) edge-kind
and valid-edge :: 'edge \( \Rightarrow \) bool
and Entry :: 'node ('('.'Entry'-.') ) and get-proc :: 'node \( \Rightarrow \) 'pname
and get-return-edges :: 'edge \( \Rightarrow \) 'edge set
and procs :: ('pname \( \times \) 'var list \( \times \) 'var list) list and Main :: 'pname
and Exit::'node ('('.'Exit'-.') )
and Def :: 'node \( \Rightarrow \) 'var set and Use :: 'node \( \Rightarrow \) 'var set
and ParamDefs :: 'node \( \Rightarrow \) 'var set and ParamUses :: 'node \( \Rightarrow \) 'var set list +
fixes H :: 'var set
fixes L :: 'var set
fixes High :: 'node ('('.'High'-.') )
fixes Low :: 'node ('('.'Low'-.') )
assumes Entry-edge-Exit-or-High:
\[
[\text{valid-edge a}; \text{sourcenode a} = (-Entry-) ] \Rightarrow \text{targetnode a} = (-Exit-) \lor \text{targetnode a} = (-High-)
\]
and High-target-Entry-edge:
\[
\exists a. \text{valid-edge a} \land \text{sourcenode a} = (-Entry-) \land \text{targetnode a} = (-High-) \land \\
\text{kind a} = (\lambda s. \text{True})_0
\]
and Entry-predecessor-of-High:
\[
[\text{valid-edge a}; \text{targetnode a} = (-High-)] \Rightarrow \text{sourcenode a} = (-Entry-)
\]
and Exit-edge-Entry-or-Low: [valid-edge a; targetnode a = (-Exit-)]
\[
\Rightarrow \text{sourcenode a} = (-Entry-) \lor \text{sourcenode a} = (-Low-)
\]
and Low-source-Exit-edge:
\[
\exists a. \text{valid-edge a} \land \text{sourcenode a} = (-Low-) \land \text{targetnode a} = (-Exit-) \land \\
\text{kind a} = (\lambda s. \text{True})_0
\]
and Exit-successor-of-Low:
\[
[\text{valid-edge a}; \text{sourcenode a} = (-Low-)] \Rightarrow \text{targetnode a} = (-Exit-)
\]
and DefHigh: Def (-High-) = H
and UseHigh: Use (-High-) = H
and UseLow: Use (-Low-) = L
and HighLowDistinct: H \cap L = \{\}
and HighLowUNIV: H \cup L = UNIV

begin

lemma Low-neq-Exit: assumes L \neq \{\} shows (-Low-) \neq (-Exit-)
proof
  assume (-Low-) = (-Exit-)
  have Use (-Exit-) = \{\} by fastforce
  with UseLow \langle L \neq \{\} \rangle \langle (-Low-) = (-Exit-) \rangle show False by simp
qed

lemma valid-node-High [simp]:valid-node (-High-)
  using High-target-Entry-edge by fastforce

lemma valid-node-Low [simp]:valid-node (-Low-)
  using Low-source-Exit-edge by fastforce

lemma get-proc-Low:
  get-proc (-Low-) = Main
proof
  from Low-source-Exit-edge obtain a where valid-edge a
    and sourcenode a = (-Low-) and targetnode a = (-Exit-)
    and intra-kind (kind a) by (fastforce simp:intra-kind-def)
  from \langle valid-edge a \rangle \langle intra-kind (kind a) \rangle
  have get-proc (sourcenode a) = get-proc (targetnode a) by (rule get-proc-intra)
  with \langle sourcenode a = (-Low-) \rangle \langle targetnode a = (-Exit-) \rangle get-proc-Exit
  show \?thesis by simp
qed

lemma get-proc-High:
  get-proc (-High-) = Main
proof
  from High-target-Entry-edge obtain a where valid-edge a
    and sourcenode a = (-Entry-) and targetnode a = (-High-)
    and intra-kind (kind a) by (fastforce simp:intra-kind-def)
  from \langle valid-edge a \rangle \langle intra-kind (kind a) \rangle
  have get-proc (sourcenode a) = get-proc (targetnode a) by (rule get-proc-intra)
  with \langle sourcenode a = (-Entry-) \rangle \langle targetnode a = (-High-) \rangle get-proc-Entry
  show \?thesis by simp
qed
lemma Entry-path-High-path:
assumes ⟨-Entry-⟩ − as→∗ n and inner-node n
obtains a' as' where as = a'♯ as' and ⟨-High-⟩ − as'→* n
and kind a' = (λs. True)

proof (atomize-elim)
from ⟨-Entry-⟩ − as→∗ n ⟨inner-node n⟩
show ∃ a' as'. as = a'♯ as' ∧ ⟨-High-⟩ − as'→* n ∧ kind a' = (λs. True)

proof (induct n ′≡⟨-Entry-⟩ as n rule:path.induct)
  case (Cons-path n ′′ as n' a)
    from ⟨n'' − as→∗ n'⟩ ⟨inner-node n'⟩ have n'' ≠ ⟨-Exit-⟩
      by (fastforce simp:inner-node-def)
    with valid-edge a (sourcenode a = ⟨-Entry-⟩) ⟨targetnode a = n''⟩
    have n'' = ⟨-High-⟩ by -(drule Entry-edge-Exit-or-High,auto)
    from High-target-Entry-edge
    obtain a' where valid-edge a' and sourcenode a' = ⟨-Entry-⟩
      and targetnode a' = ⟨-High-⟩ and kind a' = (λs. True)
      by blast
    with valid-edge a (sourcenode a = ⟨-Entry-⟩) ⟨targetnode a = n''⟩
      ⟨n'' = ⟨-High-⟩⟩ have a = a' by (auto dest:edge-det)
    with ⟨n'' − as→∗ n'⟩ ⟨n'' = ⟨-High-⟩⟩ ⟨kind a' = (λs. True)⟩ show ?case by blast
qed fastforce

lemma Exit-path-Low-path:
assumes n − as→* ⟨-Exit-⟩ and inner-node n
obtains a' as' where as = as'@ [a'] and n − as'→* ⟨-Low-⟩
and kind a' = (λs. True)

proof (atomize-elim)
from ⟨n − as→* ⟨-Exit-⟩⟩
show ∃ a' as'. as = as'@ [a'] ∧ n − as'→* ⟨-Low-⟩ ∧ kind a' = (λs. True)

proof (induct as rule:rev-induct)
  case Nil
    with ⟨inner-node n⟩ show ?case by fastforce
next
case (snc a' as')
  from ⟨n − as'@[a']→* ⟨-Exit-⟩⟩
  have n − as'→* source-node a' and valid-edge a' and target-node a' = ⟨-Exit-⟩
    by (auto elim:path-split-snc)
    { assume source-node a' = ⟨-Entry-⟩
      with ⟨n − as'→* source-node a'⟩ have n = ⟨-Entry-⟩
        by (blast intro!:path-Entry-target)
      with ⟨inner-node n⟩ have False by (simp add:inner-node-def) }
    with valid-edge a' (target-node a' = ⟨-Exit-⟩) have source-node a' = ⟨-Low-⟩
      by (blast dest!:Entry-edge-Exit-or-Low)
  from Low-source-Exit-edge
  obtain ax where valid-edge ax and source-node ax = ⟨-Low-⟩
and targetnode ax = (-Exit-) and kind ax = (λs. True) ∨
by blast
with (valid-edge a') (targetnode a' = (-Exit-)) (sourcenode a' = (-Low-))
have a' = ax by (fastforce intro:edge-det)
with (n-as'→* sourcenode a') (sourcenode a' = (-Low-)) (kind ax = (λs. True)) ∨
show ?case by blast
qed

lemma not-Low-High: V /∈ L ⇒ V ∈ H
using HighLowUNIV
by fastforce

lemma not-High-Low: V /∈ H ⇒ V ∈ L
using HighLowUNIV
by fastforce

2.2 Low Equivalence

In classical noninterference, an external observer can only see public values, in our case the L-variables. If two states agree in the values of all L-variables, these states are indistinguishable for him. Low equivalence groups those states in an equivalence class using the relation ≈_L:

definition lowEquivalence :: ('var → 'val) list ⇒ ('var → 'val) list ⇒ bool
(infixl ≈_L 50)
where s ≈_L s' ≡ ∀ V ∈ L. hd s V = hd s' V

The following lemmas connect low equivalent states with relevant variables as necessary in the correctness proof for slicing.

lemma relevant-vars-Entry:
assumes V ∈ rv S (CFG-node (-Entry-)) and (-High-) /∈ [HRB-slice S]CFG
shows V ∈ L
proof −
from (V ∈ rv S (CFG-node (-Entry-))) obtain as n'
where (-Entry-) as→_s parent-node n'
and n' ∈ HRB-slice S and V ∈ UseSDG n'
and ∀ n'', valid-SDG-node n'' ∧ parent-node n'' ∈ set (sourcenodes as)
→ V /∈ DefSDG n'' by (fastforce elim:rvE)
from (-Entry-) as→_s parent-node n' have valid-node (parent-node n')
by (fastforce intro: path-valid-node simp: intra-path-def)
thus ?thesis
proof (cases parent-node n' rule: valid-node-cases)
case Entry
with (V ∈ UseSDG n') have False
by ¬(drule SDG-Use-parent-Use, simp add: Entry-empty)
thus ?thesis by simp

5
next
  case Exit
  with \( V \in \text{Use}_{SDG} n' \) have False
  by \( \neg(\text{drule SDG-Use-parent-Use}, \text{simp add:Exit-empty}) \)
  thus ?thesis by simp
next
  case inner
  with \( (-\text{Entry-}) - \text{as} \rightarrow_* \text{parent-node } n' \) obtain \( a' \) as' where \( \text{as} = a' \# \text{as}' \)
  and \( (-\text{High-}) - \text{as}' \rightarrow_* \text{parent-node } n' \)
  by \( (\text{fastforce elim:Entry-path-High-path}, \text{simp:intra-path-def}) \)
  from \( (-\text{Entry-}) - \text{as} \rightarrow_* \text{parent-node } n' \) \( \langle \text{as} = a' \# \text{as}' \rangle \)
  have sourcenode \( a' = (-\text{Entry-}) \) by \( (\text{fastforce elim: path}. \text{cases simp:intra-path-def}) \)
  show ?thesis proof (cases \( \text{as}' = [] \))
    case True
    with \( (-\text{High-}) - \text{as}' \rightarrow_* \text{parent-node } n' \) have parent-node \( n' = (-\text{High-}) \)
    by \( (\text{fastforce simp:intra-path-def}) \)
    with \( n' \in \text{HRB-slice } S \) \( (-\text{High-}) \notin [\text{HRB-slice } S]_{CFG'} \)
    have False
    thus ?thesis by simp
next
  case False
  with \( (-\text{High-}) - \text{as}' \rightarrow_* \text{parent-node } n' \) have hd \( (\text{sourcenodes as'}) = (-\text{High-}) \)
  by \( (\text{fastforce intro:path-source-node simp:intra-path-def}) \)
  from False have hd \( (\text{sourcenodes as'}) = \text{set (sourcenodes as')} \)
  by \( (\text{fastforce intro:hd-in-set simp:sourcenodes-def}) \)
  with \( \text{as} = a' \# \text{as}' \) have hd \( (\text{sourcenodes as'}) = \text{set (sourcenodes as')} \)
  by \( (\text{simp add:sourcenodes-def}) \)
  from \( \langle \text{hd (sourcenodes as')} = (-\text{High-}) \rangle \)
  have valid-node \( (\text{hd (sourcenodes as'))} \) by simp
  have \( \text{valid-SDG-node (CFG-node (-\text{High-}))} \) by simp
  with \( \langle \text{hd (sourcenodes as')} = (-\text{High-}) \rangle \)
  \( \langle \text{hd (sourcenodes as')} = \text{set (sourcenodes as')} \rangle \)
  \( \forall n''. \text{valid-SDG-node } n'' \land \text{parent-node } n'' \in \text{set (sourcenodes as)} \)
  \( \rightarrow V \notin \text{Def}_{SDG} n'' \)
  have \( V \notin \text{Def} (-\text{High-}) \)
  by \( (\text{fastforce dest:CFG-Def-SDG-Def OF (valid-node (hd (sourcenodes as')))} \)
  hence \( V \notin H \) by \( (\text{simp add:DefHigh}) \)
  thus ?thesis by \( (\text{rule not-High-Low}) \)
qed
qed

lemma lowEquivalence-relevant-nodes-Entry:
assumes \( s \approx_L s' \) and \( (-\text{High-}) \notin [\text{HRB-slice } S]_{CFG} \)
shows $\forall V \in \text{rv} S \ (\text{CFG-node} \ (-\text{Entry}-)). \ \text{hd} \ s \ V = \text{hd} \ s' \ V$

proof
fix $V$ assume $V \in \text{rv} S \ (\text{CFG-node} \ (-\text{Entry}-))$
with $(-\text{High}-) \notin [\text{HRB-slice} \ S]_{\text{CFG}}$ have $V \in L$ by -(rule relevant-vars-Entry)
with $(s \approx_L \ s')$ show $\text{hd} \ s \ V = \text{hd} \ s' \ V$ by(simp add:lowEquivalence-def)

qed

2.3 The Correctness Proofs

In the following, we present two correctness proofs that slicing guarantees IFC noninterference. In both theorems, $\text{CFG-node} \ (-\text{High}-) \notin \text{HRB-slice} \ S$, where $\text{CFG-node} \ (-\text{Low}-) \in S$, makes sure that no high variable (which are all defined in $(-\text{High}-)$) can influence a low variable (which are all used in $(-\text{Low}-)$).

First, a theorem regarding $(-\text{Entry}-) \ -\text{as}\rightarrow (-\text{Exit}-)$ paths in the control flow graph (CFG), which agree to a complete program execution:

lemma slpa-rv-Low-Use-Low:
assumes $\text{CFG-node} \ (-\text{Low}-) \in S$
shows $\{\text{same-level-path-aux cs as; \ upd-cs cs as = []; \ same-level-path-aux cs as'};
\ \forall c \in \text{set} \ cs. \ \text{valid-edge} \ c; \ m -\text{as}\rightarrow (-\text{Low}-); \ m -\text{as'}\rightarrow (-\text{Low}-);
\ \forall i < \text{length} \ cs. \ \forall V \in \text{rv} S \ (\text{CFG-node} \ (\text{sourcenode} \ (cs!i))).
\ \text{fst} \ (s!\text{Suc} \ i) \ V = \text{fst} \ (s^{\text{Suc} \ i}) \ V; \ \forall i < \text{Suc} \ (\text{length} \ cs). \ \text{snd} \ (s!i) = \text{snd} \ (s^{\text{i}});$
\ \forall V \in \text{rv} S \ (\text{CFG-node} \ m). \ \text{state-val} \ s \ V = \text{state-val} \ s' \ V;
\ \text{preds} \ (\text{slice-kinds} \ S \ as) \ s; \ \text{preds} \ (\text{slice-kinds} \ S \ as') \ s';
\ \text{length} \ s = \text{Suc} \ (\text{length} \ cs); \ \text{length} \ s' = \text{Suc} \ (\text{length} \ cs)]
\implies \forall V \in \text{Use} \ (-\text{Low}-). \ \text{state-val} \ (\text{transfers}(\text{slice-kinds} \ S \ as) \ s) \ V = \text{state-val} \ (\text{transfers}(\text{slice-kinds} \ S \ as') \ s') \ V$

proof (induct arbitrary;$m$ as'$ s' rule:slpa-induct)
case (slpa-empty cs)
from $(m -[]\rightarrow (-\text{Low}-))$ have $m = (-\text{Low}-)$ by fastforce
from $(m -[]\rightarrow (-\text{Low}-))$ have valid-node m
by(rule path-valid-node)+
{ fix $V$ assume $V \in \text{Use} \ (-\text{Low}-)$
moreover
from (valid-node m) $(m = (-\text{Low}-))$ have $(\text{Low}-) -[]\rightarrow \gamma \ (\text{Low}-)$
by(fastforce intro:empty-path simp:intra-path-def)
moreover
from (valid-node m) $(m = (-\text{Low}-))$ \ $(\text{CFG-node} \ (-\text{Low}-) \in S)$
have $\text{CFG-node} \ (-\text{Low}-) \in \text{HRB-slice} \ S$
by(fastforce intro:HRB-slice-refl)
ultimately have $V \in \text{rv} S \ (\text{CFG-node} \ m)$
using $(m = (-\text{Low}-))$
by(auto intro!:rel $\text{CFG-Use-SDG-Use simp:sourcenodes-def}$)
} hence $\forall V \in \text{Use} \ (-\text{Low}-). \ V \in \text{rv} S \ (\text{CFG-node} \ m)$ by simp

show ?case
proof (cases $L = \{\}$)
case True with $\text{UseLow}$ show ?thesis by simp
next

7
case False
from ⟨m − as'→* (Low−)⟩ m = (Low−) have as' = []
proof (induct m as' m'≡(Low−) rule:path.induct)
case (Cons-path m'' as' a m)
  from ⟨valid-edge a (sourcenode a = m) ⟩ m = (Low−)
  have targetnode a = (Exit−) by -(rule Exit-successor-of-Low,simp+)
  with ⟨targetnode a = m'' m''−as→* (Low−)⟩
  have (Low−) = (Exit−) by -(drule path-Exit-source,auto)
  with False have False by -(drule Low-neq-Exit,simp)
  thus ?case by simp
qed simp
with ⟨∀ V ∈ Use (Low−). V ∈ rv S (CFG-node m)⟩
∀ V ∈ rv S (CFG-node m). state-val s V = state-val s' V) Nil
show ?thesis by(auto simp:slice-kinds-def)
qed

next
case (slpa-intra cs a as)

note IH = ⟨∀ m as' s s'. [upd-cs cs as = [] ; same-level-path-aux cs as as'] ;
∀ a ∈ set cs. valid-edge a ; m − as→* (Low−) ; m − as'→* (Low−) ;
∀ i<length cs. ∀ V∈rv S (CFG-node (sourcenode (cs ! i)))
fst (s ! Suc i) V = fst (s' ! Suc i) V ;
∀ i<Suc (length cs) , snd (s ! i) = snd (s' ! i) ;
∀ V∈rv S (CFG-node m) . state-val s V = state-val s' V ;
preds (slice-kinds S as) s ; preds (slice-kinds S as') s' ;
length s = Suc (length cs) ; length s' = Suc (length cs) ⟩
implies ⟨∀ V∈Use (Low−). state-val (transfers(slice-kinds S as) s) V =
state-val (transfers(slice-kinds S as') s') V ⟩

note rvs = ⟨∀ i<length cs. ∀ V∈rv S (CFG-node (sourcenode (cs ! i)))
fst (s ! Suc i) V = fst (s' ! Suc i) V ⟩
from ⟨m − a # as→* (Low−)⟩ have sourcenode a = m and valid-edge a
and targetnode a = as→* (Low−) by(auto elim:path-split-Cons)
show ?case
proof(cases L = { })
case True with UseLow show ?thesis by simp
next
case False
show ?thesis
proof(cases as as')
case Nil
with ⟨m − as'→* (Low−)⟩ have m = (Low−) by fastforce
with ⟨valid-edge a (sourcenode a = m) ⟩ have targetnode a = (Exit−)
  by -(rule Exit-successor-of-Low,simp+)
from Low-source-Exit-edge obtain a' where valid-edge a'
  and sourcenode a' = (Low−) and targetnode a' = (Exit−)
  and kind a' = (λs. True) by blast
from ⟨valid-edge a (sourcenode a = m) ⟩ m = (Low−)
⟨targetnode a = (Exit−) ⟩ ⟨valid-edge a' (sourcenode a' = (Low−) ⟩
⟨targetnode a' = (Exit−) ⟩
have a = a' by(fastforce dest:edge-det)
with \(\langle \text{kind } a' = (\lambda s. \text{True}), \rangle\) have kind a = (\(\lambda s. \text{True}\)) \(\langle\) by simp
with \(\langle \text{targetnode } a = (\text{-Exit}) \rangle\) \(\langle \text{targetnode } a -\text{as} -\text{as} -\text{as} \to (\text{-Low} -)\rangle\)
have \(\langle \text{-Low} -\rangle = (\text{-Exit} -)\) by \(-(\text{drule path-Exit-source,auto})\)
with \(\text{False}\) have \(\text{False}\) \(\langle\) by \(\langle\) \ -(\text{drule Low-neq-Exit,simp})\)
thus \(?thesis\) by simp

next
case \(\langle \text{Cons } ax \text{ asx} \rangle\)
with \(\langle m -\text{as} -\text{as} -\text{as} -\to (\text{-Low} -)\rangle\) have \(\text{sourcecenode } ax = m\) and \(\text{valid-edge } ax\)
and \(\text{targetnode } ax -\text{as} -\text{as} -\text{as} -\text{as} \to (\text{-Low} -)\) \(\langle\) by \(\langle\) \ -(\text{auto elim:split-Cons})\)
from \(\langle \text{preds } \text{slice-kinds } S (a # as) s)\rangle\)
obtain \(\text{cfs}\) where \(\langle\) \(\text{simp}\): \(s = \text{cfs}\) \(\langle\) by \(\langle\) \(\langle\) \(\text{cases } as\rangle\) \(\langle\) \(\text{auto simp: slice-kinds-def}\rangle\)
from \(\langle \text{preds } \text{slice-kinds } S as' s\rangle\) \(\langle\) \(\text{as}' = ax # asx\rangle\)
obtain \(\text{cfs}'\) where \(\langle\) \(\text{simp}\): \(s' = \text{cfs}'\) \(\langle\) by \(\langle\) \(\langle\) \(\text{cases } s\rangle\) \(\langle\) \(\text{auto simp: slice-kinds-def}\rangle\)
have \(\text{intra-kind } \langle\) \(\text{kind } ax)\rangle\)
proof \(\langle\) \(\text{cases } \text{kind } ax \text{ rule: edge-kind-cases}\rangle\)
case \(\langle\) \(\text{Call } Q r p fs)\rangle\)
have \(\text{False}\)
proof \(\langle\) \(\text{cases } \text{sourcecenode } a \in [\text{HRB-slice } S]\langle\text{CFG}\rangle\rangle\)
case \(\langle\) \(\text{True}\rangle\)
with \(\langle\) \(\text{intra-kind } \langle\) \(\text{kind } a)\rangle\rangle\)
have \(\text{slice-kind } S a = \text{kind } a\)
by \(\langle\) \(\text{rule slice-intra-kind-in-slice}\rangle\)
from \(\langle\) \(\text{valid-edge } ax\rangle\)
\(\langle\) \(\text{kind } ax = Q : r \to p fs)\rangle\)
have \(\text{unique:} \exists a'. \text{valid-edge } a' \wedge \text{sourcecenode } a' = \text{sourcecenode } ax \wedge\)
\(\langle\) \(\text{intra-kind } \langle\) \(\text{kind } a')\rangle\rangle\) \(\text{by }\langle\) \(\text{rule call-only-one-intra-edge}\rangle\)
from \(\langle\) \(\text{valid-edge } ax\rangle\)
\(\langle\) \(\text{kind } ax = Q : r \to p fs)\rangle\)
\obtain \(\langle\) \(\text{x} \in \text{get-return-edges } ax)\rangle\)
\(\text{by}\) \(\langle\) \(\text{fastforce dest:get-return-edge-call}\rangle\)
with \(\langle\) \(\text{valid-edge } ax\rangle\)
\(\text{obtain } \langle\) \(\text{a' where } \text{valid-edge } a')\rangle\)
and \(\text{sourcecenode } a' = \text{sourcecenode } ax \wedge \text{kind } a' = (\lambda cf. \text{False})\rangle\)
\(\text{by}\) \(\langle\) \(\text{fastforce dest:call-return-node-edge}\rangle\)
with \(\langle\) \(\text{valid-edge } ax\rangle\)
\(\langle\) \(\text{sourcecenode } a = m\) \(\langle\) \(\text{sourcecenode } ax = m)\rangle\)
\(\langle\) \(\text{intra-kind } \langle\) \(\text{kind } a)\rangle)\]\(\text{unique}\)
\(\text{have } a' = a\) \(\langle\) \(\text{by}\) \(\langle\) \(\text{fastforce simp: intra-kind-def})\rangle\)
with \(\langle\) \(\text{kind } a' = (\lambda cf. \text{False})\rangle\)
\(\langle\) \(\text{slice-kind } S a = \text{kind } a)\rangle\)
\(\langle\) \(\text{preds } \text{slice-kinds } S (a # as) s)\rangle\)
\(\text{have } \text{False}\) \(\langle\) \(\text{by}\) \(\langle\) \(\text{cases } s)\) \(\langle\) \(\text{auto simp: slice-kinds-def}\rangle\)
thus \(?thesis\) by simp
next
case \(\langle\) \(\text{False})\rangle\)
with \(\langle\) \(\text{kind } ax = Q : r \to p fs)\rangle\)
\(\langle\) \(\text{sourcecenode } a = m\) \(\langle\) \(\text{sourcecenode } ax = m)\rangle\)
\(\text{have } \text{slice-kind } S ax = (\lambda cf. \text{False}): r \to p fs)\)
\(\text{by}\) \(\langle\) \(\text{fastforce intra: slice-kind-Call}\rangle\)
with \(\langle\) \(\text{as}' = ax # asx)\rangle\)
\(\langle\) \(\text{preds } \text{slice-kinds } S as')\rangle\)
\(\text{have } \text{False}\) \(\langle\) \(\text{by}\) \(\langle\) \(\text{cases } s)\) \(\langle\) \(\text{auto simp: slice-kinds-def}\rangle\)
thus \(?thesis\) by simp
qed
thus \(?thesis\) by simp
next
case \(\langle\) \(\text{Return } Q p f)\rangle\)
proof show thesis have same-level-path-aux cs asx with simp qed simp
with (same-level-path-aux cs asx' (as' = ax#axs)) have same-level-path-aux cs axs by(simp:same-level-path-aux cs asx)
show thesis proof(cases targetnode a = targetnode ax)
  case True with (valid-edge a) (valid-edge ax) (source node a = m) (source node ax = m)
  have a = ax by (fastforce intro:edge-def)
  with (valid-edge a) (intra-kind (kind a)) (source node a = m)
  ∀ V ∈ rv S (CFG-node m). state-val s V = state-val s' V
  with preds (slice-kinds S (a ≠ as)) s
  with preds (slice-kinds S as) s' (as' = ax # axs)
  have rv∀ V ∈ rv S (CFG-node (targetnode a)),
  state-val (transfer (slice-kind S a) s) V =
  state-val (transfer (slice-kind S a) s') V
  by -(rule rv-edge-slice-kinds,auto)
  from upd-cs cs (a ≠ as) = [] by (fastforce simp: intra-kind-def)
  from (targetnode ax = axs→* (-Low-)) (a = ax)
  have targetnode a = axs→* (-Low-) by simp
  from (valid-edge a) (intra-kind (kind a))
  obtain cfx where cfx:transfer (slice-kind S a) s = cfx#cfs ∧ snd cfx = snd cf
    apply(cases cf)
    apply(cases sourc enode a ∈ [HRB-slice S]CFG) apply auto
    apply(fastforce dest:slice-intra-kind-in-slice simp: intra-kind-def)
    apply(auto simp: intra-kind-def)
    apply(drule slice-kind-_upd) apply auto
    by (erule kind-Predicate-notin-slice-slice-kind-Predicate) auto
  from (valid-edge a) (intra-kind (kind a))
  obtain cfx'
    where cfx':transfer (slice-kind S a) s' = cfx'#cfs' ∧ snd cfx' = snd cf'
    apply(cases cf')
    apply(cases sourc enode a ∈ [HRB-slice S]CFG) apply auto
    apply(fastforce dest:slice-intra-kind-in-slice simp: intra-kind-def)
    apply(auto simp: intra-kind-def)
    apply(drule slice-kind-_upd) apply auto
    by (erule kind-Predicate-notin-slice-slice-kind-Predicate) auto
  with cfx ∀ i < Suc (length cs). snd (s!i) = snd (s^n*i)
  have snds: ∀ i < Suc (length cs).
    snd (transfer (slice-kind S a) s ! i) =
    snd (transfer (slice-kind S a) s'! i)
  by auto(case-tac i,auto)
  from rvs cfx cfx' have rvs∀ i <length cs.
    ∀ V ∈ rv S (CFG-node (source node (cs ! i))).
\begin{verbatim}

```isar-raw
fst (transfer (slice-kind S a) s ! Suc i) V =
fst (transfer (slice-kind S a) s' ! Suc i) V
  by fastforce

from (preds (slice-kinds S (a # as)) s)
have preds (slice-kinds S as)
  (transfer (slice-kind S a) s) by(simp add:slice-kinds-def)

moreover
from (preds (slice-kinds S as') s' \langle as' = ax # ax : a = ax \rangle)
have preds (slice-kinds S ax) (transfer (slice-kind S a) s')
  by(simp add:slice-kinds-def)

moreover
from (valid-edge a) \langle intra-kind (kind a) \rangle
have length (transfer (slice-kind S a) s) = length s
  by(cases sourcenode a \in [HRB-slice S]_{CFG})
(auto dest:slice-intra-kind-in-slice slice-kind-Upd
 elim:kind-Predicate-notin-slice-slice-kind-Predicate simp: intra-kind-def)
with \langle length s = Suc (length cs) \rangle
have length (transfer (slice-kind S a) s) = Suc (length cs)
  by simp

moreover
from \langle a = ax \rangle (valid-edge a) \langle intra-kind (kind a) \rangle
have length (transfer (slice-kind S a) s') = length s'
  by(cases sourcenode ax \in [HRB-slice S]_{CFG})
(auto dest:slice-intra-kind-in-slice slice-kind-Upd
 elim:kind-Predicate-notin-slice-slice-kind-Predicate simp: intra-kind-def)
with \langle length s' = Suc (length cs) \rangle
have length (transfer (slice-kind S a) s') = Suc (length cs)
  by simp

moreover
from IH \langle OF \rangle \langle upd-cs cs as = [] \rangle (same-level-path-aux cs ax)
\langle \forall c \in set cs. valid-edge c \langle targetnode a \rightarrow \times \rangle (-Low-)
\rangle (targetnode a \rightarrow ax \times (-Low-) \langle res' \rangle \langle snds rv calculation \rangle
\langle as' = ax # ax \rangle \langle a = ax \rangle)
show ?thesis by(simp add:slice-kinds-def)

next

case False
from \langle \forall i < Suc (length cs). snd (s!i) = snd (s's!i) \rangle
have snd (hd s) = snd (hd s') by(erule-tac x=0 in allE) fastforce
with (valid-edge ax) (valid-edge ax) (sourcenode a = m)
(sourcenode ax = m) \langle as' = ax # ax \rangle False
\langle intra-kind (kind a) \rangle \langle intra-kind (kind ax) \rangle
\langle preds (slice-kinds S (a # as)) s \rangle
\langle preds (slice-kinds S as') s' \rangle
\langle \forall V \in rv S \langle CFG-node m \rangle. state-val s V = state-val s' V \rangle
\langle length s = Suc (length cs) \rangle \langle length s' = Suc (length cs) \rangle
have False by(fastforce intro!:re-branching-edges-slice-kinds-False[of a ax])
thus ?thesis by simp

qed

qed
```
```

\end{verbatim}

11
qed

next

case (slpa-Call cs a as Q r p fs)

note IH = i Lazy as' s' s'.

[upd-cs (a ≠ cs) as = []; same-level-path-aux (a ≠ cs) as';
∀ c∈set (a ≠ cs). valid-edge c; m → as→* (-Low-); m → as'→* (-Low-);
∀ i<length (a ≠ cs). ∀ V∈r S (CFG-node (source-node ((a ≠ cs) ! i))).
fst (s! Suc i) V = fst (s'! Suc i) V;
∀ i<Suc (length (a ≠ cs)). snd (s! i) = snd (s'! i);
∀ V∈r S (CFG-node m). state-val s V = state-val s' V;
preds (slice-kinds S as s); preds (slice-kinds S as' s');
length s = Suc (length (a ≠ cs)); length s' = Suc (length (a ≠ cs))]
⇒ ∀ V∈Use (-Low-). state-val (transfers(slice-kinds S as s) s V) =
state-val (transfers(slice-kinds S as' s') s' V);

note rus = ∀ i<length cs. ∀ V∈r S (CFG-node (source-node (cs ! i))).
fs (s! Suc i) V = fs (s'! Suc i) V;

from (m → a ≠ as→* (-Low-)) have source-node a = m and valid-edge a
and target-node a → as→* (-Low-)
by(auto elim:path-split-Cons)

from ∀ c∈set cs. valid-edge c; valid-edge a
have ∀ c∈set (a ≠ cs). valid-edge c by simp

show ?case

proof(cases l = { })

case True with UseLow show ?thesis by simp

next

case False

show ?thesis

proof(cases as')

case Nil

with (m → as'→* (-Low-)) have m = (-Low-) by fastforce

with (valid-edge a) ⟨source-node a = m⟩ have target-node a = (-Exit-) by
(rule Exit-successor-of-Low,simp+)

from Low-source-Exit-Edge obtain a' where valid-edge a'
and source-node a' = (-Low-) and target-node a' = (-Exit-)
and kind a' = (λs. True) by blast

from (valid-edge a) ⟨source-node a = m⟩ ⟨m = (-Low-)⟩
⟨target-node a = (-Exit-)⟩ ⟨valid-edge a'⟩ ⟨source-node a' = (-Low-)⟩
⟨target-node a' = (-Exit-)⟩

have a = a' by(fastforce dest:edge-det)

with (kind a' = (λs. True)) have kind a = (λs. True) by simp

with (target-node a = (-Exit-) ⟨target-node a → as→* (-Low-)⟩)
have (-Low-) = (-Exit-) by -(drule path-Exit-source,auto)

with False have False by -(drule Low-neq-Exit,simp)

thus ?thesis by simp

next

case (Cons ax asx)

with (m → as'→* (-Low-)) have source-node ax = m and valid-edge ax
and target-node ax → as→* (-Low-) by(auto elim:path-split-Cons)

from (preds (slice-kinds S (a ≠ as)) s)

obtain cf cf s where [simp];s = cf ≠ cf s by(cases s)(auto simp:slice-kinds-def)
from \(\langle \text{preds (slice-kinds } S \text{ as')} s' \rangle \text{ as'} = ax \# ax'\)

obtain \(c' cf'\) where \([\text{simp]} : \langle as' = cf'\# cf'\rangle\)

by \((\text{cases s') (auto simp:slice-kinds-def)}\)

have \(\exists Q r p fs. \text{ kind } ax = Q : r \rightarrow p fs\)

proof \((\text{cases kind } ax \text{ rule:edge-kind-cases)}\)

case Intra

have False

proof \((\text{cases source node } ax \in [\text{HRB-slice } S]_{\text{CFG)}}\)

case True

with \(\langle \text{intra-kind (kind } ax)\rangle\)

have slice-kind \(S ax = \text{ kind } ax\)

by \(-(\text{rule slice-intra-kind-in-slice)}\)

from \((\text{valid-edge } a) \langle \text{kind } a = Q : r \rightarrow p fs\rangle\)

have unique: \(\exists a'. \text{ valid-edge } a' \land \text{ source node } a' = \text{ source node } a \land\)

\(\text{ intra-kind (kind } a')\) by \((\text{rule call-only-one-intra-edge)}\)

from \((\text{valid-edge } a) \langle \text{kind } a = Q : r \rightarrow p fs\rangle\) obtain \(x\)

where \(x \in \text{ get-return-edges } a\) by \((\text{fastforce dest: get-return-edge-call)}\)

with \((\text{valid-edge } a) \langle \text{ obtain } a' \text{ where } \text{ valid-edge } a'\rangle\)

\(\text{ and source node } a' = \text{ source node } a\) and \(\text{ kind } a' = (\lambda cf. \text{ False})\chi\)

by \((\text{fastforce dest: call-return-node-edge)}\)

with \((\text{valid-edge AX}) \langle \text{ source node } ax = m\rangle \langle \text{ source node } a = m\rangle\)

\(\langle \text{ intra-kind (kind } ax)\rangle\) unique

have \(a' = ax\) by \((\text{fastforce simp: intra-kind-def)}\)

with \(\langle \text{kind } a' = (\lambda cf. \text{ False})\chi\rangle\)

\(\langle \text{ slice-kind } S ax = \text{ kind } ax\rangle \langle as' = ax \# ax'\rangle\)

\(\langle \text{ preds (slice-kinds } S \text{ as')} s' \rangle\)

have False by \((\text{simp add: slice-kinds-def)}\)

thus ?thesis by simp

next

case False

with \(\langle \text{kind } a = Q : r \rightarrow p fs\rangle \langle \text{ source node } ax = m\rangle \langle \text{ source node } a = m\rangle\)

have slice-kind \(S a = (\lambda cf. \text{ False}) : r \rightarrow p fs\)

by \((\text{fastforce intro: slice-kind-Call)}\)

with \(\langle \text{ preds (slice-kinds } S \text{ (a \# as)} s \rangle\)

have False by \((\text{simp add: slice-kinds-def)}\)

thus ?thesis by simp

qed

thus ?thesis by simp

next

case \((\text{Return } Q' p' f')\)

from \((\text{valid-edge } ax) \langle \text{ kind } ax = Q' : p' f'\rangle \langle \text{ valid-edge } a \rangle \langle \text{ kind } a = Q : r \rightarrow p fs\rangle\)

\(\langle \text{ source node } a = m\rangle \langle \text{ source node } ax = m\rangle\)

have False by \(-(\text{drule return-edges-only, auto)}\)

thus ?thesis by simp

qed simp

have source node \(a \in [\text{HRB-slice } S]_{\text{CFG)}\)

proof \((\text{rule contr)}\)

assume source node \(a \notin [\text{HRB-slice } S]_{\text{CFG)}\)

from this \(\langle \text{ kind } a = Q : r \rightarrow p fs\rangle\)
have slice-kind S a = (\text{lcf}. \text{False}) : r \rightarrow p fs
by (rule slice-kind-Call)
with \text{preds} (slice-kinds S (a # as)) s
show \text{False} by (simp add: slice-kinds-def)
qed
with \text{preds} (slice-kinds S (a # as)) s : \langle \text{kind} a = Q : r \rightarrow p fs \rangle
have pred (kind a) s
by (fastforce dest: slice-kind-Call-in-slice simp: slice-kinds-def)
from \langle \text{source-node} a \in [HRB-slice S]_{\text{CFG}} \rangle
\langle \text{source-node} a = m \rangle \langle \text{source-node} ax = m \rangle
have source-node ax \in [HRB-slice S]_{\text{CFG}} by simp
with \langle as' = ax # asx \rangle \langle \text{preds} (slice-kinds S as') s' \rangle
\exists Q r p fs. kind ax = Q : r \rightarrow p fs
have pred (kind ax) s'
by (fastforce dest: slice-kind-Call-in-slice simp: slice-kinds-def)
\{ fix V assume V \in \text{Use} (source-node ax)
from \langle \text{valid-edge} a : \text{have source-node} a \in \text{valid-edge ax} \rangle
\langle \text{have source-node} a \in [HRB-slice S]_{\text{CFG}} \rangle
\langle \text{valid-edge} a : \langle V \in \text{Use} (source-node ax) \rangle \rangle
have V \in \text{rv S} (CFG-node (source-node ax)) by (auto intro!: rvf CFG-Use-SDG-Use simp: SDG-to-CFG-set-def source-nodes-def)
\}
with \langle \forall V \in \text{rv S} (CFG-node m) \rangle \langle \text{state-val} s V = \text{state-val} s' V \rangle
\langle \text{source-node} a = m \rangle
have \text{Use} V \in \text{Use} (source-node ax) \langle \text{state-val} s V = \text{state-val} s' V \rangle by simp
from \langle \forall i < \text{Suc} (\text{length} cs) \rangle \langle \text{snd} (s ! i) = \text{snd} (s' ! i) \rangle
have \text{snd} (hd s) = \text{snd} (hd s') by fastforce
with \langle \text{valid-edge} a : \langle \text{kind} a = Q : r \rightarrow p fs \rangle \rangle
\langle \text{valid-edge ax} \rangle
\langle \text{pred} (\text{kind} a) s \rangle \langle \text{pred} (\text{kind ax}) s' \rangle
\langle \text{Use} (\text{length} s = \text{Suc} (\text{length} cs)) \rangle
\langle \text{length} s' = \text{Suc} (\text{length} cs) \rangle
have \langle \text{simp} \rangle : \text{az} = a by (fastforce intro!: CFG-equal-Use-equal-call)
from \langle \text{same-level-path-aux cs as} \rangle \langle \text{as'} = ax # asx \rangle
\langle \text{kind} a = Q : r \rightarrow p fs \rangle
\langle \exists Q r p fs. \text{kind ax} = Q : r \rightarrow p fs \rangle
have \text{same-level-path-aux (a # cs) asx} by simp
from \langle \text{target-node ax asx} \rightarrow \ast \rangle \langle \text{Low} \rangle have \langle \text{target-node ax asx} \rightarrow \ast \rangle \langle \text{Low} \rangle
by simp
from \langle \text{kind} a = Q : r \rightarrow p fs \rangle \langle \text{upd-cs cs (a # as)} = [] \rangle
have \text{upd-cs (a # cs) as} = [] by simp
from \langle \text{source-node} a \in [HRB-slice S]_{\text{CFG}} \rangle \langle \text{kind} a = Q : r \rightarrow p fs \rangle
have slice-kind: slice-kind S a =
Q : r \rightarrow p fs (\text{cspp} (\text{target-node} a) (HRB-slice S) fs)
by (rule slice-kind-Call-in-slice)
from \langle \forall i < \text{Suc} (\text{length} cs) \rangle \langle \text{snd} (s ! i) = \text{snd} (s' ! i) \rangle slice-kind
have \text{snds} \langle \forall i < \text{Suc} (\text{length} (a # cs)) \rangle
\langle \text{snd} (\text{transfer} (\text{slice-kind} S a) s ! i) = \text{snd} (\text{transfer} (\text{slice-kind} S a) s' ! i) \rangle
by auto(case-tac i, auto)
from (valid-edge a) (kind a = Q;r→p(fs) obtain ins outs
\[
\text{where (p,ins,outs) } \in \text{ set proc by} (\text{fastforce dest!::callee-in-procs)}
\]
with (valid-edge a) (kind a = Q;r→p(fs)
\[
\text{have length (ParamUses (sourcenode a))} = \text{ length ins}
\]
by (fastforce intro:ParamUses-call-source-length)
with (valid-edge a)
\[
\text{have } \forall i < \text{ length ins. } \forall V \in (\text{ParamUses (sourcenode a)}!)i. \ V \in \text{ Use (sourcenode a)}
\]
by (fastforce intro:ParamUses-in-Use)
with \[
\forall V \in \text{ Use (sourcenode a). state-val s V} = \text{ state-val s' V}
\]
have \[
\forall i < \text{ length ins. } \forall V \in (\text{ParamUses (sourcenode a)}!)i.
\]
state-val s V = state-val s' V
by fastforce
with (valid-edge a) (kind a = Q;r→p(fs) (p,ins,outs) } \text{ set proc)
\[
\langle \text{ pred (kind a) s' } \rangle
\]
have \[
\forall i < \text{ length ins. } \text{ (params fs (fst (hd s)))!i} = (\text{params fs (fst (hd s'))})!i
\]
by (fastforce intro!:CFG-call-edge-params)
from (valid-edge a) (kind a = Q;r→p(fs) (p,ins,outs) } \text{ set proc)
\[
\text{have length fs} = \text{ length ins by (rule CFG-call-edge-length)}
\]
\{ fix i assume i < \text{ length fs}
\]
with (\text{length fs} = \text{ length ins}) \text{ have } i < \text{ length ins by simp}
\]
from (i < \text{ length fs}) \text{ have } (\text{params fs (fst cf)})!i = (fs!i) (fst cf)
by (rule params-nth)
\]
moreover \[
\text{from (i < \text{ length fs}) \text{ have } (\text{params fs (fst cf')})!i} = (fs!i) (fst cf')
\]
by (rule params-nth)
ultimately \[
\text{have } (fs!i) (fst (hd s)) = (fs!i) (fst (hd s'))
\]
using \[
\forall i < \text{ length ins. } (\text{params fs (fst (hd s)))!i} = (\text{params fs (fst (hd s'))})!i
\]
by simp \}
\]
\{ fix i assume i < \text{ length fs}
\]
with (\forall i < \text{ length fs. } (fs ! i) (fst cf) ∋ (fs ! i) (fst cf') by simp
\]
\{ fix i assume i < \text{ length fs}
\]
have (fs ! i) (fst cf) = (fs ! i) (fst cf') by simp
by (rule csppa-formal-in-notin-slice)
have (csppa (targetnode a) (HRB-slice S) 0 fs)!i (fst cf) = (csppa (targetnode a) (HRB-slice S) 0 fs)!i (fst cf')
\]
proof(cases Formal-in(targetnode a,i + 0) ∈ HRB-slice S)
case True
\[
\text{with (i < \text{ length fs)}
\]
have (csppa (targetnode a) (HRB-slice S) 0 fs)!i = fs!i
by (rule csppa-formal-in-in-slice)
\[
\text{with (fs ! i) (fst cf) = (fs ! i) (fst cf')} \text{ show } \text{thesis by simp}
\]
next
case False
\[
\text{with (i < \text{ length fs)}
\]
have (csppa (targetnode a) (HRB-slice S) 0 fs)!i = Map.empty
by (rule csppa-formal-in-notin-slice)
thus \text{thesis by simp}
qed \}
15
hence \( eq \forall i < \text{length } fs \).

\[
((\text{csppl} \ (\text{targetnode } a) \ (\text{HRB-slice } S) \ fs)!i)(\text{fst } cf) =
((\text{csppl} \ (\text{targetnode } a) \ (\text{HRB-slice } S) \ fs)!i)(\text{fst } cf')
\]  
by (simp add: cspp-def)

\{ fix \( i \) assume \( i < \text{length } fs \)

hence \( \text{params} \ (\text{csppl} \ (\text{targetnode } a) \ (\text{HRB-slice } S) \ fs)

(\text{fst } cf')!i =
((\text{csppl} \ (\text{targetnode } a) \ (\text{HRB-slice } S) \ fs)!i)(\text{fst } cf')
by (fastforce intro: params-nth)

moreover

from \( i < \text{length } fs \)

have \( \text{params} \ (\text{csppl} \ (\text{targetnode } a) \ (\text{HRB-slice } S) \ fs)

(\text{fst } cf')!i =
((\text{csppl} \ (\text{targetnode } a) \ (\text{HRB-slice } S) \ fs)!i)(\text{fst } cf')
by (fastforce intro: params-nth)

ultimately

have \( \text{params} \ (\text{csppl} \ (\text{targetnode } a) \ (\text{HRB-slice } S) \ fs)

(\text{fst } cf')!i =
(\text{params} \ (\text{csppl} \ (\text{targetnode } a) \ (\text{HRB-slice } S) \ fs)(\text{fst } cf')!i
using eq (\( i < \text{length } fs \) by simp }

hence \( \text{params} \ (\text{csppl} \ (\text{targetnode } a) \ (\text{HRB-slice } S) \ fs)(\text{fst } cf) =
\text{params} \ (\text{csppl} \ (\text{targetnode } a) \ (\text{HRB-slice } S) \ fs)(\text{fst } cf')
by (simp add: list-eq-iff-nth-eq)

with slice-kind \((p, ins, outs) \in \text{set procs}\)

obtain cfz where [simp]:

transfer (slice-kind S a) (cf#cfs) = cfz#cf#cfs
transfer (slice-kind S a) (cf'#cfs') = cfz#cf'#cfs'

by auto

hence rv: \( \forall V \in rv S \ (\text{CFG-node } (\text{targetnode } a)) \).

state-val (transfer (slice-kind S a) s) V =
state-val (transfer (slice-kind S a) s') V by simp

from \( rv: \forall V \in rv S \ (\text{CFG-node } m) \), state-val s V = state-val s' V \( \langle \text{source node } a = m \rangle \).

have \( res': \forall i < \text{length } (a \neq cs) \).

\( \forall V \in rv S \ (\text{CFG-node } (\text{source node } ((a \neq cs) \ i))) \).

fst ((transfer (slice-kind S a) s) ! Suc i) V =
fst ((transfer (slice-kind S a) s') ! Suc i) V
by auto(case-tac i, auto)

from \( \langle \text{preds } (\text{slice-kinds } S (a \neq as)) \rangle \) s

have \( \text{preds } (\text{slice-kinds } S \ ax) \)

(transfer (slice-kind S a) s) by (simp add: slice-kinds-def)

moreover

from \( \langle \text{preds } (\text{slice-kinds } S \ ax) \rangle \) s' \( \langle as' = ax \# ax \rangle \)

have \( \text{preds } (\text{slice-kinds } S \ ax) \)

(transfer (slice-kind S a) s') by (simp add: slice-kinds-def)

moreover

from \( \langle \text{length } s = \text{Suc } (\text{length } cs) \rangle \)

have \( \text{length } (\text{transfer } (slice-kind S a) s) =
\text{Suc } (\text{length } (a \neq cs)) \) by simp

16
moreover
from \( \text{length } s' = \text{Suc } (\text{length } cs) \)
have \( \text{length } (\text{transfer } (\text{slice-kind } S \ a) \ s') = \)
  \( \text{Suc } (\text{length } (a \ # cs)) \) by simp
moreover
from \( H \) \( \OF \ (\text{upd-cs } (a \ # cs) \ as = []) \) 
(\text{same-level-path-aux } (a \ # cs) \ asx)
\( \forall c \in \text{set } (a \ # cs). \text{valid-edge } c \) \( \langle \text{targetnode } a \ → \ as \rightarrow * \ (\text{-Low-}) \rangle \)
\( \langle \text{valid-edge } a \ → \ as \rightarrow * \ (\text{-Low-}) \rangle \) \( \text{rvs' } \text{snds } \text{rv calculation} | \text{as'} = ax#asx \)
show \( \text{?thesis } \) by (simp add: slice-kinds-def)
qed
qed
next
case (slpa-Return cs a as Q p f c' cs')
note \( H' = (\forall m \ as' \ s' s. \ [\text{upd-cs } cs' \ as = []]; \text{same-level-path-aux } cs' \ as';\)
\( \forall c \in \text{set } cs'. \text{valid-edge } c \); \( m \rightarrow as \rightarrow * \ (\text{-Low-}); \ m \rightarrow as' \rightarrow * \ (\text{-Low-}); \)
\( \forall i < \text{length } cs', \forall V \in \text{rv } S \) \( (\text{CFG-node } (\text{source-node } (cs' \ ! i)) \).
\( \text{fst } (s \ ! \ Suc \ i) \ V = \text{fst } (s' ! \ Suc \ i) \ V; \)
\( \forall i < \text{Suc } (\text{length } cs'). \text{snd } (s \ ! \ Suc \ i) \) \( = \text{snd } (s' ! \ Suc \ i); \)
\( \forall V \in \text{rv } S \) \( (\text{CFG-node } m) \). \text{state-val } s \ V = \text{state-val } s' V; \)
\( \text{preds } (\text{slice-kinds } S \ as) \ s); \text{preds } (\text{slice-kinds } S \ as') \ s'; \)
\( \text{length } s = \text{Suc } (\text{length } cs'); \text{length } s' = \text{Suc } (\text{length } cs') \)
\( \implies \forall V \in \text{Use } (\text{-Low-}). \text{state-val } (\text{transfers } (\text{slice-kinds } S \ as) \ s) \ V = \)
  \( \text{state-val } (\text{transfers } (\text{slice-kinds } S \ as') \ s') \ V' \).
note \( \text{rvs} = (\forall i < \text{length } cs'. \forall V \in \text{rv } S \) \( (\text{CFG-node } (\text{source-node } (cs' \ ! i)) \).
\( \text{fst } (s \ ! \ Suc \ i) \ V = \text{fst } (s' ! \ Suc \ i) \ V) \)
from \( (\exists m \rightarrow a \ # \ as \rightarrow * \ (\text{-Low-}) \) have \( \text{source-node } a = m \) and \( \text{valid-edge } a \)
  
and \( \text{targetnode } a \ → \ as \rightarrow * \ (\text{-Low-}) \) by (auto elim: path-split-Cns)
from \( \forall c \in \text{set } cs. \text{valid-edge } c \); \( \langle cs = c' \ # cs' \rangle \)
have \( \text{valid-edge } c' \) and \( \forall c \in \text{set } cs'. \text{valid-edge } c \) by simp-all
show \( ?case \)
proof(cases \( L = \{\} \))
  case True with UseLow show \( ?thesis \) by simp
next
case False
show \( ?thesis \)
proof(cases \( as' \))
case Nil
with \( (m \rightarrow as' \rightarrow * \ (\text{-Low-}) \) have \( m = (\text{-Low-}) \) by fastforce
with \( \langle \text{valid-edge } a \rangle \langle \text{source-node } a = m \rangle \) have \( \text{targetnode } a = (\text{-Exit-}) \)
  
and \( (\lambda s. \text{True}) \) by blast
from \( \text{Low-source-Exit-edge } \) obtain \( a' \) \( \text{where } \text{valid-edge } a' \)
  
and \( \text{source-node } a' = (\text{-Low-}) \) and \( \text{targetnode } a' = (\text{-Exit-}) \)
  
and \( \text{kind } a' = (\lambda s. \text{True}) \) by blast
from \( \langle \text{valid-edge } a \rangle \langle \text{source-node } a = m \rangle \langle m = (\text{-Low-}) \)
(\langle \text{targetnode } a = (\text{-Exit-}) \rangle \langle \text{valid-edge } a' \rangle \langle \text{source-node } a' = (\text{-Low-}) \rangle)
(\langle \text{targetnode } a' = (\text{-Exit-}) \rangle)
have \( a = a' \) by (fastforce dest: edge-det)
with \( \langle \text{kind } a' = (\lambda s. \text{True}) \rangle \) have \( \text{kind } a = (\lambda s. \text{True}) \) by simp
with \( \langle \text{targetnode } a = (\text{-Exit-}) \rangle \langle \text{targetnode } a \rightarrow as \rightarrow * \ (\text{-Low-}) \rangle \)
next
have (Low) = (Exit) by ¬(drule path-Exit-source, auto)
with False have False by ¬(drule Low-neq-Exit, simp)
thus ?thesis by simp
next
case (Cons ax asx)
with \( m \rightarrow \textit{as'} \rightarrow \ast \) (Low) have sourcenode ax = m and valid-edge ax
and targetnode ax \rightarrow \textit{as'} \rightarrow \ast \) (Low) by (auto elim:path-split-Cons)
from (valid-edge a) (valid-edge ax) (kind a = Q \rightarrow p)
(sourcenode a = m) (sourcenode ax = m)
have \( \exists \ Q \ f . \ 	ext{kind ax} = Q \rightarrow p \) by (auto dest:return-edges-only)
with (same-level-path-ax cs as') (as' = ax \# ax') (cs = c' \# cs')
have ax \in \text{get-return-edges c'} and same-level-path-ax cs' asx by auto
from (valid-edge c') (ax \in \text{get-return-edges c'} \langle a \in \text{get-return-edges c'}
have [simp]: ax = a by (rule get-return-edges-unique)
from (targetnode ax \rightarrow \textit{as'} \rightarrow \ast \) (Low) have targetnode a \rightarrow \textit{as'} \rightarrow \ast \ (Low)
by simp
from (upd-cs cs \langle a \# as \rangle = \emptyset) (kind a = Q \rightarrow p) (cs = c' \# cs')
(a \in \text{get-return-edges c'})
have upd-cs cs' as = \emptyset by simp
from (length s = Suc (length cs)) (cs = c' \# cs')
obtain cf cfx cfs where s = cf \# cfx \# cfs
by (cases s, auto, case-tac list, fastforce+)
from (length s' = Suc (length cs)) (cs = c' \# cs')
obtain cf' cfx' cfs' where s' = cf' \# cfx' \# cfs'
by (cases s', auto, case-tac list, fastforce+)
from res (cs = c' \# cs') (s = cf \# cfx \# cfs) (s' = cf' \# cfx' \# cfs')
have res1: \forall i < \text{length cs'}.
\forall V \in \textit{rv S} (CFG-node (sourcenode (cs' ! i))).
fst ((cfx \# cfs)) ! Suc i V = fst ((cf' \# cfs') ! Suc i) V
and \forall V \in \textit{rv S} (CFG-node (sourcenode c')).
(fst cfx) V = (fst cfx') V
by auto
from (valid-edge c') (a \in \text{get-return-edges c'})
obtain Qx rx px fsx where kind c' = Qx:rx \rightarrow px:fsx
by (fastforce dest!: only-call-get-return-edges)
have \forall V \in \textit{rv S} (CFG-node (targetnode a)).
V \in \textit{rv S} (CFG-node (sourcenode c'))
proof
fix V assume V \in \textit{rv S} (CFG-node (targetnode a))
from (valid-edge c') (a \in \text{get-return-edges c'})
obtain a' where edge:valid-edge a' sourcenode a' = sourcenode c'
targetnode a' = targetnode a intra-kind (kind a')
by -(drule call-return-node-edge, auto simp: intra-kind-def)
from (V \in \textit{rv S} (CFG-node (targetnode a))).
obtain as n' where targetnode a \rightarrow \textit{as'} \rightarrow \ast parent-node n'
and n' \in \textit{HRB-slice S} and V \in \textit{Use_{SDG}} n'
and \forall n'' \in \textit{valid-SDG-node n''} and parent-node n'' \in \textit{set (sourcenodes as)}
\rightarrow V \notin \textit{Def_{SDG}} n'' by (fastforce elim: rvE)
from (targetnode a \rightarrow \ast parent-node n') edge
have source-node c' \rightarrow a' \# \rightarrow \ast parent-node n'
  by (fastforce intro:Cons-path simp:intra-path-def)
from (valid-edge c') (kind c' = Qx:tx \rightarrow pxfsx) have Def (source-node c') = {}
  by (rule call-source-Def-empty)
hence \forall n'. valid-SDG-node n' \land parent-node n'' = source-node c'
  \rightarrow V \notin Def SDG n'' by (fastforce dest:SDG-Def-parent-Def)
with all (source-node a' = source-node c')
have \forall n''. valid-SDG-node n'' \land parent-node n'' \in set (source-nodes (a' \# as))
  \rightarrow V \notin Def SDG n'' by (fastforce simp:source-nodes-def)
with (source-node c' \rightarrow a' \# \rightarrow \ast parent-node n')
(n' \in HRB-slice S) (V \in Use_{SDG} n')
show V \in rv S (CFG-node (source-node c'))
  by (fastforce intro:rvI)
qed
show \?thesis
proof (cases source-node a \in [HRB-slice S] CFG)
case True
from (valid-edge c') (a \in get-return-edges c')
have get-proc (targetnode c') = get-proc (source-node a)
  by -(drule intra-proc-additional-edge,
    auto dest get-proc-intra simp:intra-kind-def)
moreover
from (valid-edge c') (kind c' = Qx:tx \rightarrow pxfsx)
have get-proc (targetnode c') = px by (rule get-proc-call)
moreover
from (valid-edge a) (kind a = Q \rightarrow pf)
have get-proc (source-node a) = p by (rule get-proc-return)
ultimately have [simp]: px = p by simp
from (valid-edge c') (kind c' = Qx:tx \rightarrow pxfsx)
obtain ins outs where (p, ins, outs) \in set procs
  by (fastforce dest!: callee-in-procs)
with (source-node a \in [HRB-slice S] CFG)
  (valid-edge a) (kind a = Q \rightarrow pf)
have slice-kind: slice-kind S a =
  Q \rightarrow pf cf c' rspp (targetnode a) (HRB-slice S) outs cf' cf
  by (rule slice-kind-Return-in-slice)
with (s = cf \# cf x \# cf s') (s' = cf' \# cf x' \# cf s')
have sz: transfer (slice-kind S a) s =
  (rspp targetnode a) (HRB-slice S) outs (fst cf x) (fst cf),
  snd cf x \# cf s
and sz': transfer (slice-kind S a) s' =
  (rspp targetnode a) (HRB-slice S) outs (fst cf x') (fst cf'),
  snd cf x' \# cf s'
by simp-all
with rvI have rv': \forall i < length cs'.
  \forall V \in rv S (CFG-node (source-node (cs' ! i))).
\[
\begin{align*}
\text{fst } &((\text{transfer } (\text{slice-kind } S \ a) \ s)) \Vert \text{Suc } i \ V = \\
\text{fst } &((\text{transfer } (\text{slice-kind } S \ a) \ s')) \Vert \text{Suc } i \ V
\end{align*}
\]
by fastforce

from slice-kind \( \forall i < \text{Suc } (\text{length } cs) \). snd \((s \Vert i) = \text{snd } (s' \Vert i) \) \(cs'\)

\[
\langle s = cf \# cf x \# cf s \rangle \langle s' = cf' \# cf x' \# cf s' \rangle
\]

have \(\text{snd } (\text{transfer } (\text{slice-kind } S \ a) \ s \Vert i) = \text{snd } (\text{transfer } (\text{slice-kind } S \ a) \ s' \Vert i)\)

apply auto apply \(\text{case-tac } i\) apply auto
by (erule-tac \(x = \text{Suc } (\text{Suc } \text{nat})\) in \text{allE}) auto

have \(\forall V \in rv S \) (\text{CFG-node } (\text{targetnode } a)\).
\(\text{rspp } (\text{targetnode } a) \) (\text{HRB-slice } S) outs
\(\text{fst cfz} \) (\(\text{fst cf}\)) \(V = \)
\(\text{rspp } (\text{targetnode } a) \) (\text{HRB-slice } S) outs
\(\text{fst cfz'} \) (\(\text{fst cf'}\)) \(V\)

proof

fix \(V\) assume \(V \in \text{rv } S \) (\text{CFG-node } (\text{targetnode } a)\)

show (\text{rspp } (\text{targetnode } a) \) (\text{HRB-slice } S) outs
\(\text{fst cfz} \) (\(\text{fst cf}\)) \(V = \)
\(\text{rspp } (\text{targetnode } a) \) (\text{HRB-slice } S) outs
\(\text{fst cfz'} \) (\(\text{fst cf'}\)) \(V\)

proof (cases \(V \in \text{set } (\text{ParamDefs } (\text{targetnode } a))\))

\text{case } \text{True}

then obtain \(i\) where \(i < \text{length } (\text{ParamDefs } (\text{targetnode } a))\)
and (\text{ParamDefs } (\text{targetnode } a)) \(i = V\)
by (fastforce simp: in-set-cone-nth)

from \(\text{valid-edge } a \) \(\langle \text{kind } a = Q \leftarrow p f \rangle \) \((p, \text{ins}, \text{outs}) \in \text{set procs}\)
have \(\text{length}(\text{ParamDefs } (\text{targetnode } a)) = \text{length } \text{outs}\)
by (fastforce intro: ParamDefs-return-target-length)

show \(\text{?thesis}\)

proof (cases \(\text{Actual-out } (\text{targetnode } a, i) \in \text{HRB-slice } S\))

\text{case } \text{True}

with \(i < \text{length } (\text{ParamDefs } (\text{targetnode } a))\) \(\langle \text{valid-edge } a \rangle\)
\(\langle \text{length}(\text{ParamDefs } (\text{targetnode } a)) = \text{length } \text{outs}\rangle\)
\(\langle (\text{ParamDefs } (\text{targetnode } a)) \langle i = V \rangle \langle \text{THEN } \text{sym} \rangle\)

have \(\text{rspp-eq } (\text{rspp } (\text{targetnode } a) \langle \text{HRB-slice } S \rangle \text{ outs } (\text{fst cfz} ) (\text{fst cf} )) \langle V \rangle = \)
\(\langle (\text{fst cf})(\text{outs} l)\rangle\)
\(\langle (\text{rspp } (\text{targetnode } a) \langle \text{HRB-slice } S \rangle \text{ outs } (\text{fst cfz'}) (\text{fst cf'})) \langle V \rangle = \)
\(\langle (\text{fst cf'})(\text{outs} l)\rangle\)

by (auto intro: rspp-Actual-out-in-slice)

from \(\langle \text{valid-edge } a \rangle \) \(\langle \text{kind } a = Q \leftarrow p f \rangle \) \((p, \text{ins}, \text{outs}) \in \text{set procs}\)
have \(\forall V \in \text{set } \text{outs}, V \in \text{Use } (\text{source node } a)\) by (fastforce dest: outs-in-Use)

have \(\forall V \in \text{Use } (\text{source node } a)\), \(V \in \text{rv } S \) (\text{CFG-node } m)

proof

fix \(V\) assume \(V \in \text{Use } (\text{source node } a)\)

from \(\langle \text{valid-edge } a \rangle \) \(\langle \text{source node } a = m \rangle\)
have parent-node (CFG-node m) −⇒ parent-node (CFG-node m)
by (fastforce intro:empty-path simp:intra-path-def)
with (sourcenode a ∈ [HRB-slice S]CFG)
V ∈ Use (sourcenode a) ; (sourcenode a = m) ; valid-edge a
show V ∈ rv S (CFG-node m)
by ¬(rule ref,
  auto intro!:CFG-Use-SDG-Use simp:SDG-to-CFG-set-def
sourcnodes-def)
qed
with (∀ V ∈ set outs. V ∈ Use (sourcenode a))
have ∃ V ∈ set outs. V ∈ rv S (CFG-node m) by simp
with (∀ V∈rv S (CFG-node m), state-val s V = state-val s′ V)
(s = cf#cfx#cfs) (s′ = cf′#cfx′#cfs′)
have ∃ V ∈ set outs. (fst cf) V = (fst cf′) V by simp
with (i < length (ParamDefs (targetnode a)))
  (length (ParamDefs (targetnode a)) = length outs)
have (fst cf)(outs!i) = (fst cf′)(outs!i) by fastforce
with rspp-eq show thesis by simp
next
case False
with (i < length (ParamDefs (targetnode a))) (valid-edge a)
  (length (ParamDefs (targetnode a)) = length outs)
  (ParamDefs (targetnode a))!i = V;[THEN sym]
have rspp-eq:(rspp (targetnode a)
  (HRB-slice S) outs (fst cf) (fst cf)) V =
  (fst cf)(ParamDefs (targetnode a))!i
  (rspp (targetnode a)
  (HRB-slice S) outs (fst cf′) (fst cf′) V =
  (fst cf′)(ParamDefs (targetnode a))!i)
by(auto intro!:rspp-Actual-out-notin-slice)
from (∀ V∈rv S (CFG-node (sourcenode c′)).
  (fst cf) V = (fst cf′) V)
  (V ∈ rv S (CFG-node (targetnode a)))
  (∀ V ∈ rv S (CFG-node (targetnode a)),
V ∈ rv S (CFG-node (sourcenode c′)))
   (ParamDefs (targetnode a))!i = V;[THEN sym]
have (fst cf) (ParamDefs (targetnode a) ! i) =
  (fst cf′) (ParamDefs (targetnode a) ! i) by fastforce
with rspp-eq show thesis by fastforce
qed
next
case False
with (∀ V∈rv S (CFG-node (sourcenode c′)).
  (fst cf) V = (fst cf′) V)
  (V ∈ rv S (CFG-node (targetnode a)))
  (∀ V ∈ rv S (CFG-node (targetnode a)),
V ∈ rv S (CFG-node (sourcenode c′)))
show thesis by (fastforce simp:rspp-def map-merge-def)
qed

21
qed

with \( \text{sx} \ \text{sx}' \)

have \( \text{rv} \ ' : \forall V \in \text{rv} \ S \ (\text{CFG-node (targetnode a)}), \)

state-val \((\text{transfer (slice-kind S a)} \ s)\ V = \)

state-val \((\text{transfer (slice-kind S a)} \ s')\ V \)

by fastforce

from \( \langle \text{preds (slice-kinds S (a \# as))} \ s \rangle \)

have \( \text{preds (slice-kinds S asx)} \)

\((\text{transfer (slice-kind S a)} \ s')\)

by\((\text{simp add:slice-kinds-def})\)

moreover

from \( \langle \text{preds (slice-kinds S as')} \ s' \ (\text{ax}\#\text{axas}) \rangle \)

have \( \text{preds (slice-kinds S asx)} \)

\((\text{transfer (slice-kind S a)} \ s')\)

by\((\text{simp add:slice-kinds-def})\)

moreover

from \( \langle \text{length s} = \text{Suc (length cs)} \rangle \langle \text{cs} = c' \# c's' \text{sx} \rangle \)

have \( \text{length (transfer (slice-kind S a)} \ s) = \text{Suc (length cs')} \)

by\((\text{simp simp add:s c cf # cf # cf})\)

moreover

from \( \langle \text{length s'} = \text{Suc (length cs)} \rangle \langle \text{cs} = c' \# c's' \text{sx}' \rangle \)

have \( \text{length (transfer (slice-kind S a)} \ s') = \text{Suc (length cs')} \)

by\((\text{simp simp add:s c cf # cf # cf})\)

moreover

from \( \langle \text{IH OF \langle \text{upd-CS cs as = [}\rangle (same-level-path-aux cs' asx) \rangle} \)

\((\forall c \text{\in set cs', valid-edge c)} \langle \text{targetnode a} \rightarrow CS \rightarrow \rightarrow \text{(Low)} \rangle \)

\((\text{targetnode } a \rightarrow \text{asx} \rightarrow \rightarrow \text{(Low)})\) \text{ res' snds rv' calculation)} \langle \text{as'} = \text{ax}\#\text{axas} \rangle\)

show \( \text{?thesis by (simp add:slice-kinds-def)} \)

next

case \( \text{False} \)

from \( \text{this } \text{kind a} = Q \rightarrow \text{pf} \)

have \( \text{slice-kind(slice-kind S a} = (\lambda \text{cf}. \text{True}) \rightarrow \lambda \text{cf cf'. cf'} \)

by\((\text{rule slice-kind-Return})\)

with \( (s = \text{cf # cf # cf}) (s' = \text{cf # cf # cf}) \)

have \( \langle \text{simp} \rangle \text{transfer (slice-kind S a)} \ s = \text{cf # cf # cf} \)

\( \langle \text{transfer (slice-kind S a)} \ s' = \text{cf # cf # cf} \text{ by simp-all} \rangle \)

from \( \text{slice-kind \forall i \leq \text{Suc (length cs)}, snd (s ! i) = snd (s' ! i)} \)

\( \langle \text{cs} = c' \# c's' \langle s = \text{cf # cf # cf} \langle s' = \text{cf # cf # cf} \rangle \rangle \)

have \( \text{snds} (\forall i \leq \text{Suc (length cs)} \rangle \)

\( \langle \text{snd (transfer (slice-kind S a)} \ s ! i) = \)

\( \langle \text{snd (transfer (slice-kind S a)} \ s' ! i) \rangle \text{ by fastforce} \)

from \( \text{rvs1 have } \text{rvs}' \forall i < \text{length cs}'. \)

\( \forall V \in \text{rv} \ S (\text{CFG-node (source-node (cs' ! i)})) \)

\( \text{fst ((transfer (slice-kind S a)} \ s) ! \text{Suc i} \ V = \)

\( \text{fst ((transfer (slice-kind S a)} \ s') ! \text{Suc i} \ V \)

by fastforce

from \( \forall V \in \text{rv} \ S (\text{CFG-node (target-node a)})) \)

\( \forall V \in \text{rv} \ S (\text{CFG-node (source-node c')}) \)

\( \forall V \in \text{rv} \ S (\text{CFG-node (source-node c')}} \).

22
(fst cfz) \( V = (fst cfz') V \)

**have** \( rv \cdot \forall V \in rv S \ (CFG-node \ (targetnode \ a)) \).  
state-val \ (transfer \ (slice-kind \ S \ a) \ s) \ V =  
state-val \ (transfer \ (slice-kind \ S \ a) \ s') \ V \textbf{ by simp}  

**from** \( \langle \text{preds \ (slice-kinds \ S \ (a \ # \ as)) \ s} \rangle \)  
**have** \( \text{preds \ (slice-kinds \ S \ asx)} \)  
(transfer \ (slice-kind \ S \ a) \ s')  
by(\(simp \ \text{add}:\text{slice-kinds-def}\))  
moreover  
**from** \( \langle \text{preds \ (slice-kinds \ S \ as') \ s'} \ \langle as' = ax#asx \rangle \)  
**have** \( \text{preds \ (slice-kinds \ S \ ax)} \)  
(transfer \ (slice-kind \ S \ a) \ s')  
by(\(simp \ \text{add}:\text{slice-kinds-def}\))  
moreover  
**from** \( \langle \text{length \ s = Suc \ (length \ cs)} \ \langle cs = c' \# \ cs' \rangle \)  
**have** \( \text{length \ (transfer \ (slice-kind \ S \ a) \ s) = Suc \ (length \ cs')} \)  
by(\(simp, simp \ \text{add}: s = cf \# cfz \# cfz'\))  
moreover  
**from** \( \langle \text{length \ s' = Suc \ (length \ cs)} \ \langle cs = c' \# \ cs' \rangle \)  
**have** \( \text{length \ (transfer \ (slice-kind \ S \ a) \ s') = Suc \ (length \ cs')} \)  
by(\(simp, simp \ \text{add}: s' = cf' \# cfz' \# cfz's')\)  
moreover  
**from** \( IH \langle OF \ \langle \text{ upd-cs cs' as = [] \ (same-level-path-aux cs' asx)} \ \rangle \ \langle \text{valid-edge c \ (targetnode \ a \ →* \ (-Low-)) \ res' snds rv' calculation} \ \langle as' = ax#asx \rangle \)  
**show** \( ?\text{thesis by}(\(simp \ \text{add}:\text{slice-kinds-def}\)) \)  
**qed**  
**qed**  
**qed**

**lemma** \( \text{rv-Low-Use-Low}: \)  
**assumes** \( m \ →* \ \langle \text{-Low-} \rangle \) \( \text{and} \ m \ →* \ \langle \text{-Low-} \rangle \) \( \text{and} \ \text{get-proc} \ m = \text{Main} \)  
and \( \forall V \in rv S \ (CFG-node \ m). \ cf \ V = cf' \ V \)  
and \( \text{preds \ (slice-kinds \ S \ as')} [(cf, \text{undefined})]\)  
and \( \text{preds \ (slice-kinds \ S \ as')} [(cf', \text{undefined})]\)  
and \( \text{CFG-node \ (\langle \text{-Low-} \rangle) \in S} \)  
**shows** \( \forall V \in \text{Use \ (\langle \text{-Low-} \rangle)} \).  
state-val \ (\text{transfers}(\text{slice-kinds \ S \ as'}) [(cf', \text{undefined})]) \ V =  
state-val \ (\text{transfers}(\text{slice-kinds \ S \ as'}) [(cf', \text{undefined})]) \ V \)  
**proof**(cases as)  
**case** \( \text{Nil} \)  
**with** \( \langle m \ →* \ \langle \text{-Low-} \rangle \rangle \) \( \text{have} \) \( \text{valid-node} \ m \) \( \text{and} \ m = \langle \text{-Low-} \rangle \)  
by(auto intro:path-valid-node simp:vp-def)  
\{ fix \ V \ assume \ V \in \text{Use \ (\langle \text{-Low-} \rangle)} \)  
**moreover**  
**from** \( \langle \text{valid-node} \ m \rangle \ \langle m = \langle \text{-Low-} \rangle \rangle \) \( \text{have} \) \( \langle \text{-Low-} \rangle \ →* \ \langle \text{-Low-} \rangle \)  
by(\(fastforce \ \text{intro}:\text{empty-path simp:intra-path-def}\))  
23
moreover
from (valid-node m) (m = (Low-)) (CFG-node (Low-) ∈ S)
have (CFG-node (Low-) ∈ HRB-slice S)
  by(fastforce intro:HRB-slice-refl)
ultimately have \( V \in \text{rv} S \) (CFG-node m) using (m = (Low-))
  by(auto intro!:rule CFG-Use-SDG-Use simp:sourcenodes-def)

hence \( \forall V \in \text{Use} (\text{Low-}), V \in \text{rv} S \) (CFG-node m) by simp
show \?thesis
proof(cases \( L = {} \))
case True with UseLow show \?thesis by simp
next
case False
from (m → as′→∗ (Low-)) have m → as′→∗ (Low-) by(simp add:vp-def)
from (m → as′→∗ (Low-)) (m = (Low-)) have \( \text{as}' = [] \)
proof(induct m as' m''(Low- rule:path.induct)
case (Cons-path m'' as a m)
  from (valid-edge \( \text{a} \)) (sourcenode a = m) (m = (Low-))
  have targetnode a = (Exit-) by -(rule Exit-successor-of-Low,simp+)
    with (targetnode a = m'' → as'→∗ (Low-))
    have (Low-) = (Exit-) by -(drule path-Exit-source,auto)
    with False have False by -(drule Low-neq-Exit,simp)
thus ?case by simp
qed simp
with Nil \( \forall V \in \text{rv} S \) (CFG-node m). cf V = cf' V
\( \forall V \in \text{Use} (\text{Low-}), V \in \text{rv} S \) (CFG-node m)
show \?thesis by(fastforce simp:slice-kinds-def)
qed
next
case (Cons ax asx)
  with (m → as′→∗ (Low-)) have sourcenode ax = m and valid-edge ax
    and targetnode ax → asx→∗ (Low-)
    by(auto elim:path-split-Cons simp:vp-def)
show ?thesis
proof(cases \( L = {} \))
case True with UseLow show ?thesis by simp
next
case False
show ?thesis
proof(cases as')
case Nil
  with (m → as′→∗ (Low-)) have m = (Low-) by(fastforce simp:vp-def)
  with (valid-edge ax) (sourcenode ax = m) have targetnode ax = (Exit-)
    by -(rule Exit-successor-of-Low,simp+)
from Low-source-Exit-edge obtain a' where valid-edge a'
  and sourcenode a' = (Low-) and targetnode a' = (Exit-)
  and kind a' = (As. True) by blast
from (valid-edge ax) (sourcenode ax = m) (m = (Low-))
  (targetnode ax = (Exit-)) (valid-edge a') (sourcenode a' = (Low-))
  (targetnode a' = (Exit-))
have \( ax = a' \) by (fastforce dest:edge-det)

with (kind \( a' = (\lambda s. \text{True}) \)) have \( \text{kind } ax = (\lambda s. \text{True}) \) by simp

with \( \text{(targetnode } ax = (\text{-Exit}) \) \( \text{(targetnode } ax \rightarrow* (\text{-Low}) \))

have \( \text{(Low)} = (\text{-Exit}) \) by -(drule path-Exit-source,auto)

with False have False by -(drule Low-neq-Exit,simp)

thus \( \text{?thesis by simp} \)

next

case \( (\text{Cons } ax' \ ax') \)

from \( m \rightarrow* (\text{-Low}) \) have \( \text{valid-path-aux } \emptyset \) as and \( m \rightarrow* (\text{-Low}) \)

by (simp-all add:vp-def valid-path-def)

from this \( \langle \text{as } = ax#ax' \rangle \) (get-proc \( m = \text{Main} \))

have \( \text{same-level-path-aux } \emptyset \) as \( \land \) upd-cs \( \emptyset \) as = \( \emptyset \)

by -(rule vpa-Main-slapa[of \(-\) m (\text{-Low})],

(fastforce intro!:get-proc-Low simp:valid-call-list-def)+)

hence \( \text{same-level-path-aux } \emptyset \) as and upd-cs \( \emptyset \) as = \( \emptyset \) by simp-all

from \( m \rightarrow* (\text{-Low}) \) have \( \text{valid-path-aux } \emptyset \) as' and \( m \rightarrow* (\text{-Low}) \)

by (simp-all add:vp-def valid-path-def)

from this \( \langle \text{as'} = ax'\#ax' \rangle \) (get-proc \( m = \text{Main} \))

have \( \text{same-level-path-aux } \emptyset \) as' \( \land \) upd-cs \( \emptyset \) as' = \( \emptyset \)

by -(rule vpa-Main-slapa[of \(-\) m (\text{-Low})],

(fastforce intro!:get-proc-Low simp:valid-call-list-def)+)

hence \( \text{same-level-path-aux } \emptyset \) as' by simp

from \( \text{same-level-path-aux } \emptyset \) as \( \langle \text{upd-cs } \emptyset \) as = \( \emptyset \rangle \)

\( \langle \text{same-level-path-aux } \emptyset \) as' \( \langle \text{m } \rightarrow* \) (\text{-Low}) \( \rangle \), \( \langle \text{m } \rightarrow* \) (\text{-Low}) \( \rangle \)

\( \forall V \in \text{rv } \langle \text{CFG-node m} \rangle \). \( cf \rangle V = cf' V \langle \text{CFG-node } (\text{-Low}) \in S \rangle \)

\( \langle \text{preds } \langle \text{slice-kinds } S \rangle \) \( [(cf, \text{undefined})]\rangle \)

\( \langle \text{preds } \langle \text{slice-kinds } S \rangle \) \( [(cf', \text{undefined})]\rangle \)

show \( \text{?thesis by -(erule slpa-va-Low-Use-Low,auto} \)

qed

qed

\textbf{lemma} \ nonInterference-path-to-Low:

assumes \( [cf] \approx L \langle cf \rangle \) and \( (\text{-High}) \notin [\text{HRB-slice } S]_{\text{CFG}} \)

and \( \langle \text{CFG-node } (\text{-Low}) \in S \rangle \)

and \( (\text{-Entry}) \rightarrow* (\text{-Low}) \) and \( \langle \text{preds } \langle \text{kinds } as \rangle \) \( [(cf, \text{undefined})]\rangle \)

and \( (\text{-Entry}) \rightarrow* (\text{-Low}) \) and \( \langle \text{preds } \langle \text{kinds } as' \rangle \) \( [(cf', \text{undefined})]\rangle \)

shows \( map \text{fst } (\text{transfers } \langle \text{kinds } as \rangle \) \( [(cf, \text{undefined})]\rangle \approx_L \)

map \text{fst } (\text{transfers } \langle \text{kinds } as' \rangle \) \( [(cf', \text{undefined})]\rangle \)

\textbf{proof} –

from \( (\text{-Entry}) \rightarrow* (\text{-Low}) \) \( \langle \text{preds } \langle \text{kinds } as \rangle \) \( [(cf, \text{undefined})]\rangle \)

\( \langle \text{CFG-node } (\text{-Low}) \in S \rangle \)

obtain \( ax \) where \( \langle \text{preds } \langle \text{slice-kinds } S \rangle \) \( [(cf, \text{undefined})]\rangle \)

and \( V \in \text{Use } (\text{-Low}) \).

state-val \( (\text{transfers } \langle \text{slice-kinds } S \rangle \) \( [(cf, \text{undefined})]\rangle V \)

and \( \text{slice-edges } S \emptyset \) as = \( \text{slice-edges } S \emptyset \) ax
and transfers (kinds as) [(cf, undefined)] ≠ []
and (-Entry-) → asx→⋆ (-Low-
by (erule fundamental-property-of-static-slicing)
from (-Entry-) → as→⋆ (-Low-) ∙ preds (kinds as') [(cf', undefined)]
obtain asx' where preds (slice-kinds S asx') [(cf', undefined)]
and ∀ V ∈ Use (-Low-).
state-val (transfers(slice-kinds S asx') [(cf', undefined)]) V =
state-val (transfers(kinds as') [(cf', undefined)]) V
and slice-edges S [] as' =
slice-edges S [] asx'
and transfers (kinds as') [(cf', undefined)] ≠ []
and (-Entry-) → asx'→⋆ (-Low-
by (erule fundamental-property-of-static-slicing)
from [(cf) ≈L [cf'] ∙ (∃-High-) ∈ \[HRB-slice S\]_{CFG}
have ∀ V ∈ rv S (CFG-node (-Entry-)), cf V = cf' V
by (fastforce dest:lowEquivalence-relevant-nodes-Entry)
with (-Entry-) → asx→⋆ (-Low-) ∙ (-Entry-) → asx'→⋆ (-Low-
CFG-node (-Low-) ∈ S) ∙ preds (slice-kinds S asx') [(cf', undefined)]
preds (slice-kinds S asx') [(cf', undefined)]
have ∀ V ∈ Use (-Low-).
state-val (transfers(slice-kinds S asx) [(cf, undefined)]) V =
state-val (transfers(slice-kinds S asx') [(cf', undefined)]) V
by - (rule rv-Low-Use-Low,auto intro:get-proc-Entry)
with ∀ V ∈ Use (-Low-).
state-val (transfers(slice-kinds S asx) [(cf, undefined)]) V =
state-val (transfers(kinds as') [(cf', undefined)]) V
∀ V ∈ Use (-Low-).
state-val (transfers(slice-kinds S asx) [(cf, undefined)]) V =
state-val (transfers(kinds as') [(cf', undefined)]) V
transfers (kinds as) [(cf, undefined)] ≠ []
transfers (kinds as') [(cf', undefined)] ≠ []
show thesis by (fastforce simp:lowEquivalence-def UseLow neq-Nil-conv)
qed

theorem nonInterference-path:
assumes [cf] ≈L [cf'] and (-High-) ∉ \[HRB-slice S\]_{CFG}
and CFG-node (-Low-) ∈ S
and (-Entry-) → as→⋆ (-Exit-) and preds (kinds as) [(cf, undefined)]
and (-Entry-) → as'→⋆ (-Exit-) and preds (kinds as') [(cf', undefined)]
shows map fst (transfers (kinds as) [(cf, undefined)]) ≈L
map fst (transfers (kinds as') [(cf', undefined)])
proof -
from (-Entry-) → as→⋆ (-Exit-) obtain x xs where as = x#xs
and (-Entry-) = sourcenode x and valid-edge x
and targetnode x = x#xs (-Exit-)
apply (cases as = [])
apply (clarsimp simp:up-def,drule empty-path-nodes,drule Entry-noteq-Exit,simp)
by (fastforce elim: path-split-Cons simp: vp-def)
from (-Entry) \→ as \→ √* (-Exit) have valid-path as by (simp add: vp-def)
from (valid-edge x) have valid-node (targetnode x) by simp
hence inner-node (targetnode x)
proof (cases rule: valid-node-cases)
  case Entry
  with (valid-edge x) have False by (rule Entry-target)
thus ?thesis by simp
next
  case Exit
  with (targetnode x \→ xs \→∗ (-Exit)) have xs = []
  by -(drule path-Exit-source_auto)
from Entry-Exit-edge obtain z where valid-edge z
  and sourcenode z = (-Entry) and targetnode z = (-Exit)
  and kind z = (λs. False) by blast
from (valid-edge x) (valid-edge z) ((-Entry) = sourcenode x)
(sourcenode z = (-Entry)) Exit (targetnode z = (-Exit))
have x = z by (fastforce intro: edge-det)
with (preds (kinds as) [(cf, undefined)]) \→ as = x#xs \→ as = []
  (kind z = (λs. False) √)
  have False by (simp add: kinds-def)
thus ?thesis by simp
qed simp
with (targetnode x \→ xs \→∗ (-Exit)) obtain x' xs' where xs = xs'@[x']
  and targetnode x \→ xs' \→∗ (-Low) and kind x' = (λs. True) √
  by (fastforce elim: path-Low-path)
with (-Entry) = sourcenode x (valid-edge x)
have (-Entry) \→ x#xs' \→∗ (-Low) by (fastforce intro: Cons-path)
from (valid-path as) \→ as = x#xs \→ xs = xs'@[x']
  have valid-path (x#xs')
  by (simp add: valid-path-def del: valid-path-aux.simps)
  (rule valid-path-aux-split, simp)
with (-Entry) \→ x#xs' \→∗ (-Low) have (-Entry) \→ x#xs' \→∗ √ (-Low)
  by (simp add: vp-def)
from (as = x#xs) \→ xs = xs'@[x'] \→ by simp
with (preds (kinds as) [(cf, undefined)])
  have preds (kinds (x#xs')) [(cf, undefined)]
  by (simp add: kinds-def preds-split)
from (-Entry) \→ as' \→ √* (-Exit) obtain y ys where as' = y#ys
  and (-Entry) = sourcenode y and valid-edge y
  and targetnode y \→ ys \→∗ (-Exit)
  apply (cases as' = [])
  apply (clarsimp simp: vp-def, drule empty-path-nodes, drule Entry-noteq-Exit, simp)
  by (fastforce elim: path-split-Cons simp: vp-def)
from (-Entry) \→ as' \→ √* (-Exit) have valid-path as' by (simp add: vp-def)
from (valid-edge y) have valid-node (targetnode y) by simp
hence inner-node (targetnode y)
proof (cases rule: valid-node-cases)
  case Entry
with \( \text{valid-edge } y \) have \( \text{False by (rule Entry-target)} \)

thus \( \text{thesis by simp} \)

next

case Exit

with \( \text{targetnode } y \to ys \to* (\text{-Exit}) \) have \( y s = [] \)

by \(-(\text{drule path-Exit-source, auto})\)

from Entry-Exit-edge obtain \( z \) where \( \text{valid-edge } z \)

and \( \text{source-node } z = (\text{-Entry}) \) and \( \text{target-node } z = (\text{-Exit}) \)

and \( \text{kind } z = (\lambda s. \text{False}) \) by blast

from \( \text{valid-edge } y \) \( \text{(valid-edge } z \) \( \text{(Entry-) = source-node } y \)

\( \langle \text{source-node } z = (\text{-Entry}) \) \( \text{Exit } \langle \text{target-node } z = (\text{-Exit}) \rangle \)

have \( y = z \) by \( (\text{fastforce intro:edge-det}) \)

with \( \text{preds } (\text{kinds } as') \) \( \langle \text{cf', undefined} \rangle \)

\( \langle \text{as'} = y \# ys' \) \( \langle y s = [] \rangle \) \( \text{by simp} \)

thus \( \text{thesis by simp} \)

qed

simp

with \( \text{targetnode } y \to ys \to* (\text{-Exit}) \) obtain \( y' \) \( y's \) where \( y s = ys'@[y'] \)

and \( \text{targetnode } y \to ys' \to* (\text{-Low}) \) and \( \text{kind } y' = (\lambda s. \text{True}) \)

by \( (\text{fastforce elim:Exit-path-Low-path}) \)

with \( \text{(Entry-) = source-node } y \) \( \text{(valid-edge } y \)

have \( \text{(-Entry-) = y \# ys' \to* (\text{-Low}) by (fastforce intro:Cons-path) \)

from \( \text{(valid-path as'b) \( \langle \text{xs' = ys' \) \( \langle y s = ys'@[y'] \rangle \)

have \( \text{valid-path } (y \# ys') \)

by \( (\text{simp add:valid-path-def del:valid-path-aux,simps}) \)

(\text{rule valid-path-aux-split,simp}) \)

with \( \text{(Entry-) = y \# ys' \to* (\text{-Low}) \) \( \langle \text{Entry-} \) \( \text{have (Entry-) = y \# ys' \to* (\text{-Low}) \)

by \( (\text{simp add:up-def}) \)

from \( \langle \text{as'} = y \# ys' \) \( \langle y s = ys'@[y'] \rangle \) \( \text{have as'} = (y \# ys')@[y'] \) by simp

with \( \text{preds } (\text{kinds } as') \) \( \langle \text{cf', undefined} \rangle \)

have \( \text{preds } (\text{kinds } (y \# ys')) \) \( \langle \text{cf', undefined} \rangle \)

by \( (\text{simp add:kinds-def pred-splat}) \)

from \( \langle \text{cf} \rangle \approx_L [\text{cf'} \langle \text{-High} \rangle] \in [\text{HRB-slice } S]_{\text{CFG}} \)

\( \langle \text{CFG-node } (\text{-Low}) \in S \rangle \)

\( \langle \text{(-Entry-) = x \# xs \to* (\text{-Low}) \rangle \) \( \langle \text{preds } (\text{kinds } (x \# xs')) \rangle \) \( \langle \text{cf', undefined} \rangle \)

\( \langle \text{(-Entry-) = y \# ys' \to* (\text{-Low}) \rangle \) \( \langle \text{preds } (\text{kinds } (y \# ys')) \rangle \) \( \langle \text{cf', undefined} \rangle \)

have \( \text{map fst } (\text{transfers } (\text{kinds } (x \# xs')) \) \( \langle \text{cf', undefined} \rangle \rangle \approx_L \)

\( \text{map fst } (\text{transfers } (\text{kinds } (y \# ys')) \) \( \langle \text{cf', undefined} \rangle \) \)

by \( (\text{rule nonInterference-path-to-Low}) \)

with \( \langle \text{as' = x \# xs} \) \( \langle \text{xs = xs'@[x']} \rangle \) \( \langle \text{kind } x' = (\lambda s. \text{True}) \rangle \)

\( \langle \text{as'} = y \# ys' \) \( \langle y s = ys'@[y'] \rangle \) \( \langle \text{kind } y' = (\lambda s. \text{True}) \rangle \)

show \( \text{thesis} \)

apply \( (\text{cases transfers } (\text{map kind } (x \# xs')) \) \( (\text{transfer } (\text{kind } x) \) \( \langle \text{cf', undefined} \rangle) \)

apply \( (\text{auto simp add:kinds-def transfers-split}) \)

by \( (\text{auto simp add:kinds-def transfers-split}) \)

qed

end
The second theorem assumes that we have a operational semantics, whose evaluations are written \(<c,s> \Rightarrow <c',s'>\) and which conforms to the CFG. The correctness theorem then states that if no high variable influenced a low variable and the initial states were low equivalent, the resulting states are again low equivalent:

**locale** NonInterferenceInter =
NonInterferenceInterGraph sourcenode targetnode kind valid-edge Entry
get-proc get-return-edges procs Main Exit Def Use ParamDefs ParamUses
H L High Low +
SemanticsProperty sourcenode targetnode kind valid-edge Entry get-proc
get-return-edges procs Main Exit Def Use ParamDefs ParamUses sem identifies
for sourcenode :: 'edge ⇒ 'node and targetnode :: 'edge ⇒ 'node
and kind :: 'edge ⇒ ('var,'val,'ret,'pname) edge-kind
and valid-edge :: 'edge ⇒ bool
and Entry :: 'node (''Entry'-') and get-proc :: 'node ⇒ 'pname
and get-return-edges :: 'edge ⇒ 'edge set
and procs :: ('pname × 'var list × 'var list) list and Main :: 'pname
and Exit::'node (''Exit'')
and Def :: 'node ⇒ 'var set and Use :: 'node ⇒ 'var set
and ParamDefs :: 'node ⇒ 'var list and ParamUses :: 'node ⇒ 'var set list
and sem :: 'com ⇒ ('var → 'val) list ⇒ 'com ⇒ ('var → 'val) list ⇒ bool
(((1,1)) ⇒ (1,-)) [0,0,0,0] 81
and identifies :: 'node ⇒ 'com ⇒ bool (- ≡ [50,0] 80)
and H :: 'var set and L :: 'var set
and High :: 'node (''High'') and Low :: 'node (''Low'') +
fixes final :: 'com ⇒ bool
assumes final-edge-Low: [final c; n ≡ c]
⇒ ∃ a. valid-edge a ∧ sourcenode a = n ∧ targetnode a = (-Low-) ∧ kind a = ↑id
begin

The following theorem needs the explicit edge from (-High-) to n. An
approach using a init predicate for initial statements, being reachable from
(-High-) via a (λs. True) edge, does not work as the same statement could
be identified by several nodes, some initial, some not. E.g., in the program
while (True) Skip;;Skip two nodes identify this initial statement: the
initial node and the node within the loop (because of loop unrolling).

**theorem** nonInterference:
assumes [cf1] ≡L [cf2] and (-High-) \∉ \{HRB-slice S\} CFG
and CFG-node (-Low-) ∈ S
and valid-edge a and sourcenode a = (-High-) and targetnode a = n
and kind a = (λs. True) and n ≡ c and final c'
and (c,[cf1]) ⇒ (c',s1) and (c,[cf2]) ⇒ (c',s2)
shows s1 ≡L s2
proof –
from High-target-Entry-edge obtain az where valid-edge ax
and sourcenode ax = (-Entry-) and targetnode ax = (-High-
and kind ax = (λs. True) by blast

29
from (n ≡ c) \((c, [cf]) \Rightarrow (c', [s])\) 
obtain n₁ as₁ cfs₁ where \(n \rightarrow as₁ \Rightarrow^* n₁\) and \(n₁ \equiv c'\)
  and \(\text{preds} (\text{kinds} as₁) \{[cf₁, \text{undefined}]\}\)
  and \(\text{transfers} (\text{kinds} as₁) \{[cf₁, \text{undefined}]\} = \text{cfs₁}\) and \(\text{map} \text{fst} \text{cfs₁} = s₁\)
by (fastforce dest fundamental-property)

from \((n \rightarrow as₁ \Rightarrow^* n₁)\) valid-edge \(a\) \((\text{source} node a = (-\text{High})): \text{target} node a = n:\)

\(\text{kind} a = (\lambda s. \text{True})\)\)\(\text{by} (\text{fastforce intro: Cons-path simp: vp-def valid-path-def})\)

from \((\text{final} c) \langle n₁ \equiv c'\rangle\)

\obtain a₁ where valid-edge \(a₁\) and \(\text{source} node a₁ = n₁\)
  and \(\text{target} node a₁ = (-\text{Low})\) and \(\text{kind} a₁ = \uparrow \text{id}\) by (fastforce dest: final-edge-Low)

\(\text{hence} n₁ - [a₁] \Rightarrow^* (-\text{Low})\) by (fastforce intro: path-edge)

with (-\text{High} \(\rightarrow \text{a} # as₁ \Rightarrow^* n₁)\)
  have (-\text{High} \(\rightarrow \text{a} # as₁ \Rightarrow^* n₁\)) as \((-\text{Low})\)
  by (fastforce intro: path-Append simp: vp-def)

with \((\text{valid-edge} ax)\) \((\text{source} node ax = (-\text{Entry}))\) \((\text{target} node ax = (-\text{High}))\)

\(\text{have} (-\text{Entry} \(\rightarrow ax \# ((\text{a} # as₁) \Rightarrow [a₁]) \Rightarrow^* (-\text{Low})\) by \(-\text{rule Cons-path})\)
moreover

from \((-\text{High} \rightarrow \text{a} # as₁ \Rightarrow^* n₁)\) have valid-path-aux \[] (\text{a} # as₁)
  by (\text{simp add: vp-def valid-path-def})

with \((\text{kind} a₁ = \uparrow \text{id})\) have valid-path-aux \[] (\text{a} # as₁) \Rightarrow [a₁])
  by (fastforce intro: valid-path-aux-Append)

with \((\text{kind} ax = (\lambda s. \text{True})\)\)
  have valid-path-aux \[] (ax # ((\text{a} # as₁) \Rightarrow [a₁]))
  by simp

ultimately have \((-\text{Entry} \rightarrow ax \# ((\text{a} # as₁) \Rightarrow [a₁]) \Rightarrow^* (-\text{Low})\)
  by (simp add: vp-def valid-path-def)

from \((\text{valid-edge} a)\) \((\text{kind} a = (\lambda s. \text{True})\)\)
  \((\text{source} node a = (-\text{High}))\)
  \((\text{target} node a = n)\)

\(\text{have} \text{get-proc} n = \text{get-proc} \ (-\text{High})\)
  \((\text{by (fastforce dest: get-proc intra simp: intra-kind-def)})\)

with \((\text{get-proc} \text{High})\) \((\text{have} \text{get-proc} n = \text{Main} \text{ by simp})\)

from \((\text{valid-edge} a)\) \((\text{source} node a₁ = n₁)\)
  \((\text{target} node a₁ = (-\text{Low}))\)
  \((\text{kind} a₁ = \uparrow \text{id})\)

\(\text{have} \text{get-proc} n₁ = \text{get-proc} \ (-\text{Low})\)
  \((\text{by (fastforce dest: get-proc intra simp: intra-kind-def)})\)

with \((\text{get-proc} \text{Low})\) \((\text{have} \text{get-proc} n₁ = \text{Main} \text{ by simp})\)

from \((n \rightarrow as₁ \Rightarrow^* n₁)\) \((\text{have} \ n \rightarrow as₁ \Rightarrow^* n₁)\)

\((\text{by cases as₁})\)

\((\text{auto dest: vpa-main-slsa intro: get-proc n₁ = Main} \ \text{get-proc} n = \text{Main}) \ \text{simp: vp-def valid-path-def valid-call-list-def slp-def}
\text{same-level-path-def simp del: valid-path-aux.simp})\)

then obtain \((cfz r)\) where \((cfz. \text{transfers} \ (\text{map} \ \text{kind} as₁) \ \{(cf₁, \text{undefined})\}) = \{(cfz, r)\}\)

\((\text{by (fastforce elim: slp-callstack-length-equal simp: kinds-def)})\)

from \((\text{kind} ax = (\lambda s. \text{True})\)\)

\((\text{preds} \ (\text{kinds} as₁) \ \{(cf₁, \text{undefined})\}) \ \text{kind} a₁ = \uparrow \text{id} \ cfz\)

have \(\text{preds} \ (\text{kinds} \ (ax # ((\text{a} # as₁) \Rightarrow [a₁]))) \ \{(cf₁, \text{undefined})\}\)

\((\text{by auto simp: kinds-def preds-split})\)

from \((n \equiv c) \ \langle c, [cf₂]\rangle \Rightarrow (c', [s₂])\)
obtain $n_2, a_2$, $cfs_2$ where $n \rightarrow a_2.\top n_2$ and $n_2 \triangleq c'$
and preds (kinds $a_2$) [[[$cfs_2$, undefined]]]
and transfers (kinds $a_2$) [[[$cfs_2$, undefined]]] = $cfs_2$ and map fst $cfs_2 = s_2$
by (fastforce dest: fundamental-property)
from $(n \rightarrow a_2.\top n_2)$ valid-edge $a_1$ source-node $a_1 = \langle -High- \rangle$ target-node $a_1 = n_0$
(kind $a = (\lambda s. \text{True})$)
have $(\langle -High- \rangle - ax \# ((a \# a_2) \circ [a_2]) \Rightarrow \langle -Low- \rangle)$ by (fastforce intro: Cons-path simp: vp-def valid-path-def)
from $(\text{final } c \Downarrow (n_2 \triangleq c'))$
obtain $a_2$ where valid-edge $a_2$ and source-node $a_2 = n_2$
and target-node $a_2 = \langle -Low- \rangle$ and kind $a_2 = \sqcup \text{id}$ by (fastforce dest: final-edge-Low)
hence $n_2 - a_2 \Rightarrow \langle -Low- \rangle$ by (fastforce intro: path-edge)
with $(\langle -High- \rangle - ax \# ((a \# a_2) \circ [a_2]) \Rightarrow \langle -Low- \rangle)$ valid-edge $ax = \langle -Entry- \rangle$ source-node $ax = \langle -High- \rangle$
have $(\langle -Entry- \rangle - ax \# ((a \# a_2) \circ [a_2]) \Rightarrow \langle -Low- \rangle)$ by (rule Cons-path)
otherwise from $(\langle -High- \rangle - ax \# ((a \# a_2) \circ [a_2]) \Rightarrow \langle -Low- \rangle)$ valid-path-aux \[\langle ax \# ((a \# a_2) \circ [a_2]) \Rightarrow \langle -Low- \rangle \rangle$
by (simp add: vp-def valid-path-def)
with $\langle \text{kind } a_2 = \sqcup \text{id}, \text{valid-path-aux} \langle ax \# ((a \# a_2) \circ [a_2]) \Rightarrow \langle -Low- \rangle \rangle \rangle$
by (fastforce intro: valid-path-aux-Append)
with $\langle \text{kind } ax = (\lambda s. \text{True}) \rangle$ valid-path-aux \[\langle ax \# ((a \# a_2) \circ [a_2]) \Rightarrow \langle -Low- \rangle \rangle \]
by (simp ultimately have $(\langle -Entry- \rangle - ax \# ((a \# a_2) \circ [a_2]) \Rightarrow \langle -Low- \rangle)$
by (simp add: vp-def valid-path-def)
from $(\langle -Entry- \rangle - ax \# ((a \# a_2) \circ [a_2]) \Rightarrow \langle -Low- \rangle)$ valid-edge $a_1$ source-node $a_1 = \langle -High- \rangle$
(kind $a_1 = n_0$)
have get-proc $n = \text{get-proc} (\langle -High- \rangle)$
by (fastforce dest: get-proc-intra simp: intra-kind-def)
with get-proc-High have get-proc $n = \text{Main}$ by simp
from $(\langle -Entry- \rangle - ax \# ((a \# a_2) \circ [a_2]) \Rightarrow \langle -Low- \rangle)$ \[\langle \text{target-node } a_1 = n \rangle \]
have get-proc $n_2 = \text{get-proc} (\langle -Low- \rangle)$
by (fastforce dest: get-proc-intra simp: intra-kind-def)
with get-proc-Low have get-proc $n_2 = \text{Main}$ by simp
from $(n \rightarrow a_2.\top n_2)$ have $(n \rightarrow a_2.\bot n_2)$
by (cases $a_2$
(auto dest: upa-Main-slap intro: get-proc $n_2 = \text{Main}$ get-proc $n = \text{Main} \quad \text{simp: vp-def valid-path-def valid-call-list-def slp-def}
\quad \text{same-level-path-def simp del: valid-path-aux-simps})$
then obtain $cfx', r'$
where $cfx': \text{transfers \ (map kind } a_2) \langle \langle cfx_2, \text{undefined} \rangle \rangle = \langle \langle cfx', r' \rangle \rangle$
by (fastforce elim: slp-callstack-length-equal simp: kinds-def)
from $\langle \text{kind } ax = (\lambda s. \text{True}) \rangle \\langle \text{kind } a = (\lambda s. \text{True}) \rangle$
\[\langle \text{preds (kinds } a_2) \langle \langle cfs_2, \text{undefined} \rangle \rangle \rangle \langle \text{kind } a_2 = \sqcup \text{id} \rangle \quad \text{cfx'}$
have preds (kinds $\langle ax \# ((a \# a_2) \circ [a_2]) \rangle$ $\langle cfs_2, \text{undefined} \rangle$)
by (auto simp: kinds-def preds-split)
from $(\langle cfs_2 \rangle \approx L \langle cfs_2 \rangle)$ $(\langle -High- \rangle \not\in \text{HRB-slice } S \circ \text{CFG}) \langle \text{CFG-node (} \langle -Low- \rangle \rangle \in S \rangle$
$(\langle -Entry- \rangle - ax \# ((a \# a_2) \circ [a_2]) \Rightarrow \langle -Low- \rangle)$.
In this section, we show how a valid CFG from the slicing framework in [8] can be lifted to fulfill all properties of the NonInterferenceIntraGraph locale. Basically, we redefine the hitherto existing Entry and Exit nodes as new High and Low nodes, and introduce two new nodes NewEntry and NewExit. Then, we have to lift all functions to operate on this new graph.

3.1 Liftings

3.1.1 The datatypes

datatype 'node LDCFG-node = Node 'node
  | NewEntry
  | NewExit

type-synonym ('edge, 'node, 'var, 'val, 'ret, 'pname) LDCFG-edge =
  'node LDCFG-node × (('var, 'val, 'ret, 'pname) edge-kind) × 'node LDCFG-node

3.1.2 Lifting basic definitions using 'edge and 'node

inductive lift-valid-edge :: ('edge ⇒ bool) ⇒ ('edge ⇒ 'node) ⇒ ('edge ⇒ 'node) ⇒ bool
  ⇒
  ('edge ⇒ ('var, 'val, 'ret, 'pname) edge-kind) ⇒ 'node ⇒ 'node ⇒
  ('edge, 'node, 'var, 'val, 'ret, 'pname) LDCFG-edge ⇒ bool
for valid-edge::'edge ⇒ bool and src::'edge ⇒ 'node and trg::'edge ⇒ 'node and knd::'edge ⇒ ('var,'val,'ret,'pname) edge-kind and E::'node and X::'node

where lve-edge:
[valid-edge a; src a ≠ E ∨ trg a ≠ X;
 e = (Node (src a),knd a,Node (try a))]
⇒ lift-valid-edge valid-edge src trg knd E X e

| lve-Entry-edge:
 e = (NewEntry,(λs. True),Node E)
⇒ lift-valid-edge valid-edge src trg knd E X e

| lve-Exit-edge:
 e = (Node X,(λs. True),NewExit)
⇒ lift-valid-edge valid-edge src trg knd E X e

| lve-Entry-Exit-edge:
 e = (NewEntry,(λs. False),NewExit)
⇒ lift-valid-edge valid-edge src trg knd E X e

lemma [simp]¬ lift-valid-edge valid-edge src trg knd E X (Node E,et,Node X) by (auto elim:lift-valid-edge.cases)

fun lift-get-proc :: ('node ⇒ 'pname) ⇒ 'pname ⇒ 'node LDCFG-node ⇒ 'pname
where lift-get-proc get-proc Main (Node n) = get-proc n
| lift-get-proc get-proc Main NewEntry = Main
| lift-get-proc get-proc Main NewExit = Main

inductive-set lift-get-return-edges :: ('edge ⇒ 'edge set) ⇒ ('edge ⇒ bool) ⇒ ('edge ⇒ 'node) ⇒ ('edge ⇒ 'node) ⇒ ('edge ⇒ ('var,'val,'ret,'pname) edge-kind) ⇒ ('edge,'node,'var,'val,'ret,'pname) LDCFG-edge
⇒ ('edge,'node,'var,'val,'ret,'pname) LDCFG-edge set
for get-return-edges :: 'edge ⇒ 'edge set and valid-edge :: 'edge ⇒ bool and src::'edge ⇒ 'node and trg::'edge ⇒ 'node and knd::'edge ⇒ ('var,'val,'ret,'pname) edge-kind and e::('edge,'node,'var,'val,'ret,'pname) LDCFG-edge
where lift-get-return-edgesI:
[ e = (Node (src a),knd a,Node (try a)); valid-edge a; a' ∈ get-return-edges a;
 e' = (Node (src a'),knd a',Node (try a'))]
⇒ e' ∈ lift-get-return-edges get-return-edges valid-edge src trg knd e

3.1.3 Lifting the Def and Use sets

inductive-set lift-Def-set :: ('node ⇒ 'var set) ⇒ 'node ⇒ 'node ⇒
'var set ⇒ 'var set ⇒ ('node LDCFG-node × 'var) set

for Def::('node ⇒ 'var set) and E::'node and X::'node
and H::'var set and L::'var set

where lift-Def-node:
V ∈ Def n ⇒ (Node n, V) ∈ lift-Def-set Def E X H L

| lift-Def-High:
V ∈ H ⇒ (Node E, V) ∈ lift-Def-set Def E X H L

abbreviation lift-Def :: ('node ⇒ 'var set) ⇒ 'node ⇒ 'node ⇒
'var set ⇒ 'var set ⇒ ('node LDCFG-node ⇒ 'var set)

where lift-Def Def E X H L n ≡ \{ (V, (n, V)) ∈ lift-Def-set Def E X H L \}

inductive-set lift-Use-set :: ('node ⇒ 'var set) ⇒ 'node ⇒ 'node ⇒
'var set ⇒ 'var set ⇒ ('node LDCFG-node ⇒ 'var set)

for Use::'node ⇒ 'var set and E::'node and X::'node
and H::'var set and L::'var set

where lift-Use-node:
V ∈ Use n ⇒ (Node n, V) ∈ lift-Use-set Use E X H L

| lift-Use-High:
V ∈ H ⇒ (Node E, V) ∈ lift-Use-set Use E X H L

| lift-Use-Low:
V ∈ L ⇒ (Node X, V) ∈ lift-Use-set Use E X H L

abbreviation lift-Use :: ('node ⇒ 'var set) ⇒ 'node ⇒ 'node ⇒
'var set ⇒ 'var set ⇒ ('node LDCFG-node ⇒ 'var set)

where lift-Use Use E X H L n ≡ \{ (V, (n, V)) ∈ lift-Use-set Use E X H L \}

fun lift-ParamUses :: ('node ⇒ 'var set list) ⇒ 'node LDCFG-node ⇒ 'var set list

where lift-ParamUses ParamUses (Node n) = ParamUses n

| lift-ParamUses ParamUses NewEntry = []
| lift-ParamUses ParamUses NewExit = []

fun lift-ParamDefs :: ('node ⇒ 'var list) ⇒ 'node LDCFG-node ⇒ 'var list

where lift-ParamDefs ParamDefs (Node n) = ParamDefs n

| lift-ParamDefs ParamDefs NewEntry = []
| lift-ParamDefs ParamDefs NewExit = []
3.2 The lifting lemmas

3.2.1 Lifting the CFG locales

abbreviation src :: ('edge,'node,'var,'val,'ret,'pname) LDCFG-edge ⇒ 'node LDCFG-node
where src a ≡ fst a

abbreviation trg :: ('edge,'node,'var,'val,'ret,'pname) LDCFG-edge ⇒ 'node LDCFG-node
where trg a ≡ snd(snd a)

abbreviation knd :: ('edge,'node,'var,'val,'ret,'pname) LDCFG-edge ⇒ ('var,'val,'ret,'pname) edge-kind
where knd a ≡ fst(snd a)

lemma lift-CFG:
assumes wf:CFGExit-wf sourcenode targetnode kind valid-edge Entry get-proc get-return-edges procs Main Exit Def Use ParamDefs ParamUses
and pd:Postdomination sourcenode targetnode kind valid-edge Entry get-proc get-return-edges procs Main Exit
shows CFG src trg knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewEntry
(lift-get-proc get-proc Main)
(lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind)
procs Main
proof –
interpret CFGExit-wf sourcenode targetnode kind valid-edge Entry get-proc get-return-edges procs Main Exit Def Use ParamDefs ParamUses
by(erule wf)
interpret Postdomination sourcenode targetnode kind valid-edge Entry get-proc get-return-edges procs Main Exit
by(erule pd)
show ?thesis
proof
fix a assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and trg a = NewEntry
thus False by(fastforce elim:lift-valid-edge.cases)
next
show lift-get-proc get-proc Main NewEntry = Main by simp
next
fix a Q r p fs
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and knd a = Q:r→p fs and src a = NewEntry
thus False by(fastforce elim:lift-valid-edge.cases)
next
fix a a'
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a'
and src a = src a' and trg a = trg a'
thus a = a'
proof (induct rule: lift-valid-edge.induct)
case lve-edge thus ⟨case by (erule lift-valid-edge.cases, auto dest: edge-det)⟩
qed (auto elim: lift-valid-edge.cases)

next
  fix a Q r f
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
    and knd a = Q: r→Main f
  thus False by (fastforce elim: lift-valid-edge.cases dest: Main-no-call-target)

next
  fix a Q' f'
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
    and knd a = Q': Main f'
  thus False by (fastforce elim: lift-valid-edge.cases dest: Main-no-return-source)

next
  fix a Q r p fs
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
    and knd a = Q: r→p fs
  thus ∃ ins outs. (p, ins, outs, f) ∈ set procs
    by (fastforce elim: lift-valid-edge.cases intro: callee-in-procs)

next
  fix a Q r p f
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
    and knd a = Q: r→Main f
  thus lift-get-proc get-proc Main (src a) = lift-get-proc get-proc Main (trg a)
    by (fastforce elim: lift-valid-edge.cases intro: get-proc-intra
      simp: get-proc-Entry get-proc-Exit)

next
  fix a Q r p f
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
    and knd a = Q: r→p fs
  thus lift-get-proc get-proc Main (trg a) = p
    by (fastforce elim: lift-valid-edge.cases intro: get-proc-call)

next
  fix a Q' p f'
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
    and knd a = Q': Main f'
  thus lift-get-proc get-proc Main (src a) = p
    by (fastforce elim: lift-valid-edge.cases intro: get-proc-return)

next
  fix a Q r p f
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
    and knd a = Q: r→p fs
  then obtain ax where valid-edge ax and knd ax = Q: r→p fs
    and sourcenode ax ∉ Entry ∨ targetnode ax ∉ Exit
    and src a = Node (sourcenode ax) and trg a = Node (targetnode ax)
    by (fastforce elim: lift-valid-edge.cases)
  from valid-edge ax (kind ax = Q: r→p fs)
  have all:∀ a'. valid-edge a' ∧ targetnode a' = targetnode ax →
    (∃ Qx rx fsx. kind a' = Qx: rx→p fsx)
    by (auto dest: call-edges-only)
\{ \text{fix } a' \}
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a'
    and \( \text{try } a' = \text{try } a \)
\text{hence } \exists Q_x \, \forall f_x \, f_{sx}. \text{kind } a' = Q_x:rx \mapsto pf_{sx}
\text{proof} (\text{induct rule:lift-valid-edge.induct})
    \text{case } (\text{live-edge } ax')
      \text{note} [\text{simpl}] = (e = (\text{Node } (\text{sourcenode } ax'), \text{kind } ax', \text{Node } (\text{targetnode } ax')))
      \text{from } (\text{try } e = \text{try } a); (\text{try } a = \text{Node } (\text{targetnode } ax'))
      \text{have } \text{targetnode } ax' = \text{targetnode } ax \text{ by simpl}
      \text{with } (\text{valid-edge } ax') \text{ all have } \exists Q_x \, \forall f_x \, \exists ax' = Q_x:rx \mapsto pf_{sx} \text{ by blast}
      \text{thus } \text{?case by simpl}
\text{next}
    \text{case } (\text{live-Entry-edge } e)
      \text{from } (e = (\text{NewEntry}, (\lambda s. \text{True}) \vDash, \text{Node Entry})); (\text{try } e = \text{try } a)
      (\text{try } a = \text{Node } (\text{targetnode } ax'))
      \text{have } \text{targetnode } ax = \text{Entry} \text{ by simpl}
      \text{with } (\text{valid-edge } ax) \text{ have False by (rule Entry-target)}
      \text{thus } \text{?case by simpl}
\text{next}
    \text{case } (\text{live-Exit-edge } e)
      \text{from } (e = (\text{Node Exit}, (\lambda s. \text{True}) \vDash, \text{NewExit})); (\text{try } e = \text{try } a)
      \text{have } \text{targetnode } ax' = \text{Node } (\text{targetnode } ax) \text{ by simpl}
      \text{thus } \text{?case by simpl}
\text{next}
    \text{case } (\text{live-Entry-Exit-edge } e)
      \text{from } (e = (\text{NewEntry}, (\lambda s. \text{False}) \vDash, \text{NewExit})); (\text{try } e = \text{try } a)
      \text{have } \text{targetnode } ax' = \text{Node } (\text{targetnode } ax) \text{ by simpl}
      \text{thus } \text{?case by simpl}
\text{qed } \}
\text{thus } \forall a'. \text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit } a' \land
    \text{try } a' = \text{try } a \longrightarrow (\exists Q_x \, \forall f_x \, \text{kind } a' = Q_x:rx \mapsto pf_{sx} \text{ by simp})
\text{next}
\text{fix } a \, Q' \, p \, f' \text{'}
\text{assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit } a
    \text{and } \text{kind } a = Q' \mapsto pf' \text{'}
\text{then obtain } ax \text{ where valid-edge ax and kind } ax = Q' \mapsto pf' \text{'}
    \text{and sourcenode } ax \neq \text{Entry } \lor \text{targetnode } ax \neq \text{Exit}
    \text{and } \text{src } a = \text{Node } (\text{sourcenode } ax) \text{ and } \text{try } a = \text{Node } (\text{targetnode } ax)
\text{by (fastforce elim:lift-valid-edge.cases)}
\text{from } (\text{valid-edge } ax); (\text{kind } ax = Q' \mapsto pf')
\text{have all } \forall a'. \text{valid-edge } a' \land \text{sourcenode } a' = \text{sourcenode } ax \longrightarrow
    (\exists Q_x \, f_x. \text{kind } a' = Q_x:rx \mapsto pf_x) \text{ by (auto dest:return-edges-only)}
\text{\{ fix } a' \}
\text{assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit } a'
    \text{and } \text{src } a' = \text{src } a
\text{hence } \exists Q_x \, f_x. \text{kind } a' = Q_x \mapsto pf_x
\text{proof} (\text{induct rule:lift-valid-edge.induct})
case (lve-edge ax' e)
  note [simp] = \( e = (\text{Node} (\text{sourcenode ax'}), \text{kind ax'}, \text{Node} (\text{targetnode ax'})) \)
  from \( \langle \text{src e} = \text{src a} \rangle \langle \text{src a} = \text{Node} (\text{sourcenode ax}) \rangle \)
  have sourcenode ax' = sourcenode ax by simp
  with \( \langle \text{valid-edge ax'} \rangle \) all have \( \exists Qx fx. \text{kind ax'} = Qx \mapsto_p fx \) by blast
  thus \?case by simp
next
case (lve-Entry-edge e)
  from \( \langle e = (\text{NewEntry}, (\lambda s. \text{True}) \mapsto_p, \text{Node Entry}) \rangle \langle \text{src e} = \text{src a} \rangle \langle \text{src a} = \text{Node} (\text{sourcenode ax}) \rangle \) have False by simp
  thus \?case by simp
next
case (lve-Exit-edge e)
  from \( \langle e = (\text{Node Exit}, (\lambda s. \text{True}) \mapsto_p, \text{NewExit}) \rangle \langle \text{src e} = \text{src a} \rangle \langle \text{src a} = \text{Node} (\text{sourcenode ax}) \rangle \) have False by (rule Exit-source)
  thus \?case by simp
next
case (lve-Entry-Exit-edge e)
  from \( \langle e = (\text{NewEntry}, (\lambda s. \text{False}) \mapsto_p, \text{NewExit}) \rangle \langle \text{src e} = \text{src a} \rangle \langle \text{src a} = \text{Node} (\text{sourcenode ax}) \rangle \) have False by simp
  thus \?case by simp
next
\}
thus \( \forall a'. \text{lift-valid-edge} \text{ valid-edge sourcenode targetnode kind Entry Exit a' } \land \condition{src a' = src a } \mapsto (\exists Qx fx. \text{kind a'} = Qx \mapsto_p fx) \) by simp
next
fix a Q r p fs
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
  and knd a = Q:r \mapsto_p fs
thus lift-get-return-edges get-return-edges valid-edge
  source node target node kind a \neq \{\}
proof (induct rule: lift-valid-edge.induct)
case (lve-edge ax e)
  from \( \langle e = (\text{Node} (\text{sourcenode ax}), \text{kind ax}, \text{Node} (\text{targetnode ax})) \rangle \langle knd e = Q:r \mapsto_p fs \rangle \)
  have \( \text{kind ax} = Q:r \mapsto_p fs \) by simp
  with \( \langle \text{valid-edge ax} \rangle \) have get-return-edges ax \neq \{\}
    by (rule get-return-edge-call)
  then obtain ax' where ax' \in \text{get-return-edges ax} by blast
  with \( \langle e = (\text{Node} (\text{sourcenode ax}), \text{kind ax}, \text{Node} (\text{targetnode ax})) \rangle \langle \text{valid-edge ax} \rangle \)
  have \( \langle \text{Node} (\text{sourcenode ax'}), \text{kind ax'}, \text{Node} (\text{targetnode ax'}) \rangle \in \text{lift-get-return-edges get-return-edges valid-edge} \text{ source node target node kind e} \) by (fastforce intro: lift-get-return-edgesI)
  thus \?case by fastforce
qed simp-all
next
fix a a'
assume a' ∈ lift-get-return-edges get-return-edges valid-edge
targetnode kind a
and lift-valid-edge valid-edge targetnode kind Entry Exit a
thus lift-valid-edge valid-edge targetnode kind Entry Exit a'
proof (induct rule: lift-get-return-edges.induct)
case (lift-get-return-edgesI ax a' e')
  from ⟨valid-edge ax⟩ (a' ∈ get-return-edges ax) have valid-edge a'
    by (rule get-return-edges-valid)
from ⟨valid-edge ax⟩ (a' ∈ get-return-edges ax) obtain Q r p fs
  where kind ax = Q: r ↪→ p fs
    by (fastforce dest!: only-call-get-return-edges)
with ⟨valid-edge ax⟩ (a' ∈ get-return-edges ax) obtain Q' f'
  where kind a' = Q' ↪→ p f'
from ⟨valid-edge a'⟩ (kind a' = Q' ↪→ p f') have get-proc(sourcenode a') = p
    by (rule get-proc-return)
have sourcenode a' ≠ Entry
proof
  assume sourcenode a' = Entry
  with get-proc-Entry ⟨get-proc(sourcenode a') = p⟩ have p = Main by simp
  with ⟨kind a' = Q' ↪→ p f'⟩ have kind a' = Q' ↪→ Main f'
    by (fastforce dest!: call-return-edges)
with ⟨valid-edge a'⟩ show False
  by (rule Main-no-return-source)
qed
with ⟨e' = (Node (sourcenode a'), kind a', Node (targetnode a'))⟩
  ⟨valid-edge a'⟩ show ?case
    by (fastforce intro:lve-edge)
qed
next
fix a a'
assume a' ∈ lift-get-return-edges get-return-edges valid-edge
targetnode kind a
and lift-valid-edge valid-edge targetnode kind Entry Exit a
thus ∃ Q r p fs. kind a = Q: r ↪→ p fs
proof (induct rule: lift-get-return-edges.induct)
case (lift-get-return-edgesI ax a' e')
  from ⟨a = (Node (sourcenode ax), kind ax, Node (targetnode ax))⟩
    ⟨valid-edge ax⟩ show ?case
      by simp
  ⟨valid-edge a'⟩
  show ?case
    by (fastforce intro:lve-edge)
qed
next
fix a Q r p fs a'
assume a' ∈ lift-get-return-edges get-return-edges valid-edge
targetnode kind a and kind a = Q: r ↪→ p fs
and lift-valid-edge valid-edge targetnode kind Entry Exit a
thus ∃ Q' f'. kind a' = Q' ↪→ p f'
proof (induct rule: lift-get-return-edges.induct)
case (lift-get-return-edgesI ax a' e')
  from ⟨a = (Node (sourcenode ax), kind ax, Node (targetnode ax))⟩

\( \text{knd} \ a = Q:r \rightarrow p fs \)

\textbf{have} kind \( \text{ax} = Q:r \rightarrow p fs \) by simp

\textbf{with} \( \text{valid-edge} \ \text{ax} \) \( a' \in \text{get-return-edges} \ \text{ax} \) \textbf{have} \( \exists \ Q ' f', \ \text{kind} \ a' = Q'\leftarrow p f' \)

\textbf{by} \( -( \text{rule call-return-edges} ) \)

\textbf{with} \( e' = (\text{Node} (\text{source} \ \text{node} a'), \ \text{kind} \ a', \ \text{Node} (\text{target} \ \text{node} a')) \)

\textbf{show} \( ? \text{case} \ \text{by} \ \text{simp} \)

\textbf{qed}

next

\textbf{fix} \ a Q' p f'

\textbf{assume} \( \text{lift-valid-edge} \ \text{valid-edge} \ \text{source} \ \text{node} \ \text{target} \ \text{node} \ \text{kind} \ \text{Entry} \ \text{Exit} \ \text{a} \)

and \( \text{knd} \ a = Q'\leftarrow p f' \)

\textbf{thus} \( \exists !a'. \ \text{lift-valid-edge} \ \text{valid-edge} \ \text{source} \ \text{node} \ \text{target} \ \text{node} \ \text{kind} \ \text{Entry} \ \text{Exit} \ a' \ \land \)

\( (\exists Q \ r fs. \ \text{knd} \ a' = Q:r \rightarrow p fs) \ \land \ a \in \text{lift-get-return-edges} \ \text{get-return-edges} \)

\( \text{valid-edge} \ \text{source} \ \text{node} \ \text{target} \ \text{node} \ \text{kind} \ a' \)

\textbf{proof}(\text{induct rule:} \ \text{lift-valid-edge}\_\text{induct})

\textbf{case} \( \text{lve-edge} \ a \ e \)

\textbf{from} \( \langle e = (\text{Node} (\text{source} \ \text{node} a), \ \text{kind} \ a, \ \text{Node} (\text{target} \ \text{node} a)) \rangle \)

\( \langle \text{knd} \ e = Q'\leftarrow p f' \rangle \ \textbf{have} \ \text{knd} \ a = Q'\leftarrow p f' \) \ \text{by} \ \text{simp} \)

\textbf{with} \( \text{valid-edge} \ a \)

\textbf{have} \( \exists !a'. \ \text{valid-edge} \ a' \ \land \ (\exists Q \ r fs. \ \text{knd} \ a' = Q:r \rightarrow p fs) \ \land \ a \in \text{get-return-edges} \ a' \)

\text{by} \( \text{(rule return-needs-call)} \)

\textbf{then obtain} \( a' Q \ r fs \) \textbf{where} \( \text{valid-edge} \ a' \ \text{and} \ \text{knd} \ a' = Q:r \rightarrow p fs \)

and \( a \in \text{get-return-edges} \ a' \)

and \( \text{imp} : \forall x. \ \text{valid-edge} \ a \ \land \ (\exists Q \ r fs. \ \text{knd} \ x = Q:r \rightarrow p fs) \ \land \)

\( a \in \text{get-return-edges} \ x \ \longrightarrow x = a' \)

\text{by} \( \text{(fastforce elim:ex1E)} \)

\textbf{let} \( ? e' = (\text{Node} (\text{source} \ \text{node} a'), \text{kind} \ a', \text{Node} (\text{target} \ \text{node} a')) \)

\textbf{have} \( \text{source} \ \text{node} \ a' \neq \text{Entry} \)

\textbf{proof}

\textbf{assume} \( \text{source} \ \text{node} \ a' = \text{Entry} \)

\textbf{with} \( \text{valid-edge} \ a' \langle \text{kind} \ a' = Q:r \rightarrow p fs \rangle \)

\textbf{show} \( \text{False} \) \textbf{by} \( \text{(rule Entry-no-call-source)} \)

\textbf{qed}

\textbf{with} \( \text{valid-edge} \ a' \)

\textbf{have} \( \text{lift-valid-edge} \ \text{valid-edge} \ \text{source} \ \text{node} \ \text{target} \ \text{node} \ \text{kind} \ \text{Entry} \ \text{Exit} \ ?e' \)

\textbf{by} \( \text{(fastforce intro:} \ \text{lift-valid-edge}\_\text{lve-edge)} \)

\textbf{moreover}

\textbf{from} \( \langle \text{kind} \ a' = Q:r \rightarrow p fs \rangle \ \textbf{have} \ ? e' = Q:r \rightarrow p fs \) \textbf{by} \ \text{simp}

\textbf{moreover}

\textbf{from} \( \langle e = (\text{Node} (\text{source} \ \text{node} a), \ \text{kind} \ a, \ \text{Node} (\text{target} \ \text{node} a)) \rangle \)

\( \langle \text{valid-edge} \ a' \rangle \ a \in \text{get-return-edges} \ a' \)

\textbf{have} \( e \in \text{lift-get-return-edges} \ \text{get-return-edges} \ \text{valid-edge} \)

\( \text{source} \ \text{node} \ \text{target} \ \text{node} \ ? e' \) \textbf{by} \( \text{(fastforce intro:} \ \text{lift-get-return-edgesI)} \)

\textbf{moreover}

\{ \ \textbf{fix} \ x \}

\textbf{assume} \( \text{lift-valid-edge} \ \text{valid-edge} \ \text{source} \ \text{node} \ \text{target} \ \text{node} \ \text{kind} \ \text{Entry} \ \text{Exit} \ x \)

and \( \exists Q \ r fs. \ \text{knd} \ x = Q:r \rightarrow p fs \)

and \( e \in \text{lift-get-return-edges} \ \text{get-return-edges} \ \text{valid-edge} \)
sourcenode targetnode kind x
from \langle \text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit x} \rangle
\exists Q s fs. kind x = Q;r\rightarrow p fs
obtain y where valid-edge y
and x = (Node (sourcenode y), kind y, Node (targetnode y))
by (fastforce elim:lift-valid-edge.cases)
with \langle e \in \text{lift-get-return-edges get-return-edges valid-edge}
\text{sourcenode targetnode kind x \& valid-edge a}
\langle e = (Node (sourcenode a), kind a, Node (targetnode a)) \rangle \rangle
have \ x = ?e'
proof (induct rule:lift-get-return-edges.induct)
case (lift-get-return-edgesI ax ax' e)
from \langle \text{valid-edge ax} \rangle \langle ax' \in \text{get-return-edges ax} \rangle
have valid-edge ax'
by (rule get-return-edges-valid)
from \langle e = (Node (sourcenode ax'), kind ax', Node (targetnode ax')) \rangle
\langle e = (Node (sourcenode a), kind a, Node (targetnode a)) \rangle
have sourcenode a = sourcenode ax' and targetnode a = targetnode ax'
by simp-all
with \langle \text{valid-edge a} \rangle \langle \text{valid-edge ax} \rangle
have \ [\text{simp}: a = ax' \rangle
by (rule edge-det)
from ax = (Node (sourcenode ax), kind ax, Node (targetnode ax))
\langle \exists Q s fs. kind x = Q;r\rightarrow p fs \rangle
have \exists Q s fs. kind ax = Q;r\rightarrow p fs
by simp
with \langle \text{valid-edge ax} \rangle \langle ax' \in \text{get-return-edges ax} \rangle
imp
have ax = a' by fastforce
with \langle ax = (Node (sourcenode ax), kind ax, Node (targetnode ax)) \rangle
show \ \langle \text{thesis} \rangle
by simp
qed \ }
ultimately show \ \langle \text{case by(blast intro:exI)} \rangle
qed simp-all
next
fix a a'
assume a' \in \text{lift-get-return-edges get-return-edges valid-edge sourcenode}
targetnode kind a
and \text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a}
thus \exists a''. \text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a''} \land
src a'' = trg a \land trg a'' = src a' \land
\text{kind a''} = (\lambda cf. \text{False}) \leftrightarrow
proof (induct rule:lift-get-return-edges.induct)
case (lift-get-return-edgesI ax a' e')
from \langle \text{valid-edge ax} \rangle \langle a' \in \text{get-return-edges ax} \rangle
obtain ax' where valid-edge ax' and sourcenode ax' = targetnode ax
and targetnode ax' = sourcenode a' and kind ax' = (\lambda cf. \text{False}) \leftrightarrow
by (fastforce dest:intra-proc-additional-edge)
let \ \langle ex = (Node (sourcenode ax'), kind ax', Node (targetnode ax')) \rangle
have targetnode ax \neq \text{Entry}
proof
assume targetnode ax = Entry
with \langle \text{valid-edge ax} \rangle
show False
by (rule Entry-target)
qed
with \langle \text{sourcenode ax' = targetnode ax} \rangle
have sourcenode ax' \neq \text{Entry}
by simp
with \langle \text{valid-edge ax'} \rangle
have \text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit ?ex}
by (fastforce intro: lve-edge)
with \( e' = (\text{Node} (\text{sourcenode} a'), \text{kind} a', \text{Node} (\text{targetnode} a')) \)
\( a = (\text{Node} (\text{sourcenode} ax), \text{kind} ax, \text{Node} (\text{targetnode} ax)) \)
\( e' = (\text{Node} (\text{sourcenode} a'), \text{kind} a', \text{Node} (\text{targetnode} a')) \)
\( \langle \text{sourcenode} ax', \text{targetnode} ax' \rangle \langle \text{targetnode} ax' = \text{sourcenode} a' \rangle \)
\( \langle \text{kind} ax' = (\lambda e' \cdot \text{False}) \rangle \)
show ?case by simp
qed

next
fix \( a a' \)
assume \( a' \in \text{lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind} \)
and \( \text{lift-valid-edge valid-edge sourcenode targetnode kind} \text{Entry Exit} a \)
thus \( \exists a''. \text{lift-valid-edge valid-edge sourcenode targetnode kind} \text{Entry Exit} a'' \land \langle \text{src} a'' = \text{src} a \land \text{trg} a'' = \text{trg} a' \land \text{kind} a'' = (\lambda e' \cdot \text{False}) \rangle \)
proof (induct rule: lift-get-return-edges.induct)
case (lift-get-return-edgesI \( ax ax' ax'' \))
from \( \langle \text{valid-edge} ax \rangle \langle a' \in \text{get-return-edges} ax \rangle \)
obtain \( ax' \) where \( \text{valid-edge} ax' \land \text{sourcenode} ax' = \text{sourcenode} ax \)
and \( \text{targetnode} ax' = \text{targetnode} a' \land \text{kind} ax' = (\lambda e' \cdot \text{False}) \rangle \)
by (fastforce dest: call-return-node-edge)
let \( \exists ax = (\text{Node} (\text{sourcenode} ax'), \text{kind} ax', \text{Node} (\text{targetnode} ax')) \)
from \( \langle \text{valid-edge} ax \rangle \langle a' \in \text{get-return-edges} ax \rangle \)
obtain \( Q r p fs \) where \( \text{kind} ax = Q \cdot r \rightarrow_p fs \)
by (fastforce dest!: only-call-get-return-edges)
have \( \text{sourcenode} ax \neq \text{Entry} \)
proof
assume \( \text{sourcenode} ax = \text{Entry} \)
with \( \langle \text{valid-edge} ax \rangle \langle \text{kind} ax = Q \cdot r \rightarrow_p fs \rangle \) show False
by (rule Entry-no-call-source)
qed

with \( \langle \text{sourcenode} ax' = \text{sourcenode} ax \rangle \) have \( \text{sourcenode} ax' \neq \text{Entry} \) by simp
with \( \langle \text{valid-edge} ax' \rangle \)

have \( \text{lift-valid-edge valid-edge sourcenode targetnode kind} \text{Entry Exit} \exists ax \)
by (fastforce intro: lve-edge)
with \( \langle e' = (\text{Node} (\text{sourcenode} a'), \text{kind} a', \text{Node} (\text{targetnode} a')) \rangle \)
\( a = (\text{Node} (\text{sourcenode} ax), \text{kind} ax, \text{Node} (\text{targetnode} ax)) \)
\( e' = (\text{Node} (\text{sourcenode} a'), \text{kind} a', \text{Node} (\text{targetnode} a')) \)
\( \langle \text{sourcenode} ax' = \text{sourcenode} ax \rangle \langle \text{targetnode} ax' = \text{targetnode} a' \rangle \)
\( \langle \text{kind} ax' = (\lambda e' \cdot \text{False}) \rangle \)
show ?case by simp
qed

next
fix \( a Q r p fs \)
assume \( \text{lift-valid-edge valid-edge sourcenode targetnode kind} \text{Entry Exit} a \)
and \( \text{kind} a = Q \cdot r \rightarrow_p fs \)
thus \( \exists a'. \text{lift-valid-edge valid-edge sourcenode targetnode kind} \text{Entry Exit} a' \land \langle \text{src} a' = \text{src} a \land \text{intra-kind} (\text{kind} a') \rangle \)
proof (induct rule: lift-valid-edge.induct)
case (lve-edge a e)
  from (e = (Node (sourcenode a), kind a, Node (targetnode a))) (knd e = Q;r→pfs)
  have kind a = Q;r→pfs by simp
  with (valid-edge a) have ∃!a'. valid-edge a' ∧ sourcenode a' = sourcenode a
  \land intra-kind(kind a') by (rule call-only-one-intra-edge)
then obtain a' where valid-edge a' \land sourcenode a' = sourcenode a
  and intra-kind(kind a')
  and imp:∀x. valid-edge x \land sourcenode x = sourcenode a \land intra-kind(kind x)
  → x = a' by (fastforce elim:ex1E)
let ?e' = (Node (sourcenode a'), kind a', Node (targetnode a'))
have sourcenode a ≠ Entry
proof
  assume sourcenode a = Entry
  with (valid-edge a) (kind a = Q;r→pfs) show False
  by (rule Entry-no-call-source)
qed
with (sourcenode a' = sourcenode a) have sourcenode a' ≠ Entry by simp
with (valid-edge a')
have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit ?e'
  by (fastforce intro: lift-valid-edge.lve-edge)
moreover
from (e = (Node (sourcenode a), kind a, Node (targetnode a)))
  (sourcenode a' = sourcenode a)
have src ?e' = src e by simp
moreover
moreover
  \{ fix x
  assume lift-valid-edge valid-edge source node target node kind Entry Exit x
  and src x = src e and intra-kind (knd x)
  from (lift-valid-edge valid-edge source node target node kind Entry Exit x)
  have x = ?e'
  proof (induct rule: lift-valid-edge.cases)
    case (lve-edge ax ex)
    from (intra-kind (knd x)) (x = ex) (src x = src e)
      (ex = (Node (sourcenode ax), kind ax, Node (targetnode ax)))
        (e = (Node (sourcenode a), kind a, Node (targetnode a)))
    have intra-kind (kind ax) and sourcenode ax = sourcenode a by simp-all
      with (valid-edge ax) imp have ax = a' by fastforce
      with (x = ex) (ex = (Node (sourcenode ax), kind ax, Node (targetnode ax)))
    show ?case by simp
    next
    case (lve-Entry-edge ex)
    with (src x = src e)
      (e = (Node (sourcenode a), kind a, Node (targetnode a)))
  
};
have False by simp
thus ?case by simp

next
case (lve-Exit-edge ex)
with (src x = src e)
  \langle e = (Node (sourcenode a), kind a, Node (targetnode a)) \rangle
have sourcenode a = Exit by simp
with \langle valid-edge a \rangle have False by (rule Exit-source)
thus ?case by simp

next
case (lve-Entry-Exit-edge ex)
with \langle src x = src e \rangle
  \langle e = (Node (sourcenode a), kind a, Node (targetnode a)) \rangle
have False by simp
thus ?case by simp

qed }
ultimately show ?case by (blast intro:ex1I)

qed simp-all

next
fix a Q' p f'
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and knd a = Q'←p f'
thus \exists!a'. lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a' ∧
  try a' = try a ∧ intra-kind (knd a')

proof (induct rule:lift-valid-edge.induct)
case (lve-edge a e)
  from \langle e = (Node (sourcenode a), kind a, Node (targetnode a)) \rangle \langle knd e =
    Q'←p f' \rangle
  have knd a = Q'←p f' by simp
  with \langle valid-edge a \rangle have \exists!a'. valid-edge a' ∧ targetnode a' = targetnode a ∧
    intra-kind (knd a') by (rule return-only-one-intra-edge)
  then obtain a' where valid-edge a' and targetnode a' = targetnode a
    and intra-kind (knd a')
    and imp:\forall x. valid-edge x ∧ targetnode x = targetnode a ∧ intra-kind (kind x)
    \rightarrow x = a' by (fastforce elim:ex1E)
  let ?e' = (Node (sourcenode a'), kind a', Node (targetnode a'))
  have targetnode a ≠ Exit
  proof
    assume targetnode a = Exit
    with \langle valid-edge a \rangle \langle knd a = Q'←p f' \rangle show False
    by (rule Exit-no-return-target)
  qed
  with \langle targetnode a' = targetnode a \rangle have targetnode a' ≠ Exit by simp
  with \langle valid-edge a' \rangle have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit ?e'
    by (fastforce intro.lift-valid-edge.lve-edge)
  moreover
  from \langle e = (Node (sourcenode a), kind a, Node (targetnode a)) \rangle

44
\[
\langle \text{targetnode } a' = \text{targetnode } a \rangle
\]

have \( \text{trg } ?e' = \text{trg } e \) by simp

moreover

from \( \langle \text{intra-kind}(\text{kind } a') \rangle \) have \( \text{intra-kind}(\text{kind } ?e') \) by simp

moreover

\{ fix \( x \)
assume \( \text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit } a \)
and \( \text{trg } x = \text{trg } e \) and \( \text{intra-kind}(\text{kind } x) \)
from \( \langle \text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit } x \rangle \) have \( x = ?e' \)
proof (induct rule: lift-valid-edge.cases)
  case \( \langle \text{bve-edge } ax ex \rangle \)
  from \( \langle \text{intra-kind}(\text{kind } x) \rangle \) \( \langle \text{trg } x = \text{trg } e \rangle \)
  \( \langle \text{ex} = (\text{Node } (\text{sourcenode } ax), \text{kind } ax, \text{Node } (\text{targetnode } ax)) \rangle \)
  \( \langle e = (\text{Node } (\text{sourcenode } a), \text{kind } a, \text{Node } (\text{targetnode } a)) \rangle \)
  have \( \text{intra-kind}(\text{kind } ax) \) and \( \text{targetnode } ax = \text{targetnode } a \) by simp-all
  with \( \langle \text{valid-edge } ax \rangle \) imp have \( ax = a' \) by fastforce
  with \( \langle \text{ex} = (\text{Node } (\text{sourcenode } ax), \text{kind } ax, \text{Node } (\text{targetnode } ax)) \rangle \)
  show \( ?\text{case} \) by simp
  next
  case \( \langle \text{bve-Entry-edge } ex \rangle \)
  with \( \langle \text{trg } x = \text{trg } e \rangle \)
  \( \langle e = (\text{Node } (\text{sourcenode } a), \text{kind } a, \text{Node } (\text{targetnode } a)) \rangle \)
  have \( \text{targetnode } a = \text{Entry} \) by simp
  with \( \langle \text{valid-edge } ax \rangle \) have \( \text{False} \) by (rule Entry-target)
  thus \( ?\text{case} \) by simp
  next
  case \( \langle \text{bve-Exit-edge } ex \rangle \)
  with \( \langle \text{trg } x = \text{trg } e \rangle \)
  \( \langle e = (\text{Node } (\text{sourcenode } a), \text{kind } a, \text{Node } (\text{targetnode } a)) \rangle \)
  have \( \text{False} \) by simp
  thus \( ?\text{case} \) by simp
  next
  case \( \langle \text{bve-Entry-Exit-edge } ex \rangle \)
  with \( \langle \text{trg } x = \text{trg } e \rangle \)
  \( \langle e = (\text{Node } (\text{sourcenode } a), \text{kind } a, \text{Node } (\text{targetnode } a)) \rangle \)
  have \( \text{False} \) by simp
  thus \( ?\text{case} \) by simp
  qed \}
ultimately show \( ?\text{case} \) by (blast intro:exI)
qed simp-all

next
fix \( a a' Q_1 r_1 p f s_1 Q_2 r_2 f s_2 \)
assume \( \text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit } a \)
and \( \text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit } a' \)
and \( \text{kind } a = Q_1 r_1 \rightarrow_p f s_1 \) and \( \text{kind } a' = Q_2 r_2 \rightarrow_p f s_2 \)
then obtain \( x x' \) where valid-edge \( x \)
and \( a:a = (\text{Node } (\text{sourcenode } x), \text{kind } x, \text{Node } (\text{targetnode } x)) \) and valid-edge

45
\[
x' \quad \text{and} \quad a' = (Node (sourcenode x'), \text{kind } x', Node (targetnode x'))
\]

by (auto elim!: lift-valid-edge \text{cases})

with \( \text{kind } a = Q_1: r_1 \hookrightarrow_p s_1 \) \( \text{and} \quad a' = Q_2: r_2 \hookrightarrow_p s_2 \)

have \( \text{kind } x = Q_1: r_1 \hookrightarrow_p s_1 \) \( \text{and} \quad \text{kind } x' = Q_2: r_2 \hookrightarrow_p s_2 \) by simp-all

with \( \text{valid-edge } x \) \( \text{valid-edge } x' \) have \( \text{targetnode } x = \text{targetnode } x' \)

by (rule same-proc-call-unique-target)

with \( a a' \) show \( \text{trg } a = \text{trg } a' \) by simp

next

from unique-callers show distinct-fst procs .

next

fix \( p \) ins outs

assume \( (p, \text{ins}, \text{outs}) \in \text{set } \text{procs} \)

from distinct-formal-ins[\text{OF this}] show distinct ins .

next

fix \( p \) ins outs

assume \( (p, \text{ins}, \text{outs}) \in \text{set } \text{procs} \)

from distinct-formal-outs[\text{OF this}] show distinct outs .

qed

qed

\textbf{lemma lift-CFG-wf:}

\textbf{assumes} \( \text{wf: CFGExit-wf } \text{sourcenode } \text{targetnode } \text{kind } \text{valid-edge } \text{Entry } \text{get-proc } \text{get-return-edges } \text{procs } \text{Main } \text{Exit } \text{Def } \text{Use } \text{ParamDefs } \text{ParamUses} \)

\textbf{and} \( \text{pd: Postdomination } \text{sourcenode } \text{targetnode } \text{kind } \text{valid-edge } \text{Entry } \text{get-proc } \text{get-return-edges } \text{procs } \text{Main } \text{Exit} \)

\textbf{shows} \( \text{CFG-wf } \text{src } \text{trg } \text{knd} \)

\( (\text{lift-valid-edge } \text{valid-edge } \text{sourcenode } \text{targetnode } \text{kind } \text{Entry } \text{Exit} ) \) \( \text{NewEntry} \)

\( (\text{lift-get-proc } \text{get-proc } \text{Main}) \)

\( (\text{lift-get-return-edges } \text{get-return-edges } \text{valid-edge } \text{sourcenode } \text{targetnode } \text{kind}) \)

\( \text{procs } \text{Main} \) \( (\text{lift-Def } \text{Def } \text{Entry } \text{Exit } \text{H } \text{L} ) \) \( (\text{lift-Use } \text{Use } \text{Entry } \text{Exit } \text{H } \text{L}) \)

\( (\text{lift-ParamDefs } \text{ParamDefs}) \) \( (\text{lift-ParamUses } \text{ParamUses}) \)

\textbf{proof} –

\textbf{interpret} \( \text{CFGExit-wf } \text{sourcenode } \text{targetnode } \text{kind } \text{valid-edge } \text{Entry } \text{get-proc } \text{get-return-edges } \text{procs } \text{Main } \text{Exit } \text{Def } \text{Use } \text{ParamDefs } \text{ParamUses} \)

by (rule \text{wf})

\textbf{interpret} \( \text{Postdomination } \text{sourcenode } \text{targetnode } \text{kind } \text{valid-edge } \text{Entry } \text{get-proc } \text{get-return-edges } \text{procs } \text{Main } \text{Exit} \)

by (rule \text{pd})

\textbf{interpret} \( \text{CFG: CFG } \text{src } \text{trg } \text{knd} \)

\( \text{lift-valid-edge } \text{valid-edge } \text{sourcenode } \text{targetnode } \text{kind } \text{Entry } \text{Exit } \text{NewEntry} \)

\( \text{lift-get-proc } \text{get-proc } \text{Main}) \)

\( \text{lift-get-return-edges } \text{get-return-edges } \text{valid-edge } \text{sourcenode } \text{targetnode } \text{kind}) \)

\( \text{procs } \text{Main} \) \( (\text{lift-Def } \text{Def } \text{Entry } \text{Exit } \text{H } \text{L} ) \) \( (\text{lift-Use } \text{Use } \text{Entry } \text{Exit } \text{H } \text{L}) \)

\( (\text{lift-ParamDefs } \text{ParamDefs}) \) \( (\text{lift-ParamUses } \text{ParamUses}) \)

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

\textbf{proof}

show \( \text{lift-Def } \text{Def } \text{Entry } \text{Exit } \text{H } \text{L } \text{NewEntry} = \{\} \) ∧
lift-Use Use Entry Exit H L NewEntry = {}
by (fastforce elim: lift-Use-set.cases lift-Def-set.cases)

next
fix a Q r p fs ins outs
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and knd a = Q::p r→p fs and (p, ins, outs) ∈ set procs
thus length (lift-ParamUses ParamUses (src a)) = length ins
proof (induct rule: lift-valid-edge.induct)
  case (lve-edge a e)
    from ⟨e = (Node (sourcenode a), kind a, Node (targetnode a))⟩ ⟨knd e = Q::p r→p fs⟩
    have kind a = Q::p r→p fs and src e = Node (sourcenode a) by simp-all
    with ⟨valid-edge a ⟩ ⟨(p, ins, outs) ∈ set procs⟩
    have length (ParamUses (sourcenode a)) = length ins
    by (rule ParamUses-call-source-length)
    with ⟨src e = Node (sourcenode a)⟩ show ?case by simp
  qed simp-all

next
fix a assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
thus distinct (lift-ParamDefs ParamDefs (try a))
proof (induct rule: lift-valid-edge.induct)
  case (lve-edge a e)
    from ⟨valid-edge a ⟩ have distinct (ParamDefs (targetnode a))
    by (rule distinct-ParamDefs)
    with ⟨e = (Node (sourcenode a), kind a, Node (targetnode a))⟩
    show ?case by simp
  qed

next
  case (lve-Entry-edge e)
  have ParamDefs Entry = []
  proof (rule ccontr)
    assume ParamDefs Entry ≠ []
    then obtain V Vs where ParamDefs Entry = V # Vs
    by (cases ParamDefs Entry) auto
    hence V ∈ set (ParamDefs Entry) by fastforce
    hence V ∈ Def Entry by (fastforce intro: ParamDefs-in-Def)
    with Entry-empty show False by simp
  qed
  with ⟨e = (NewEntry, (λs. True), Node Entry)⟩ show ?case by simp
  qed simp-all

next
fix a Q’ p f’ ins outs
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and knd a = Q’::p f’ r→p fs and (p, ins, outs) ∈ set procs
thus length (lift-ParamDefs ParamDefs (trg a)) = length outs
proof (induct rule: lift-valid-edge.induct)
  case (lve-edge a e)
    from ⟨e = (Node (sourcenode a), kind a, Node (targetnode a))⟩ ⟨knd e = Q’::p f’⟩
    have kind a = Q’::p f’ and trg e = Node (targetnode a) by simp-all

47
with \(\langle \text{valid-edge } a \rangle \cdot \langle p, \text{ins}, \text{outs} \rangle \in \text{set procs} \)

have \(\text{length(ParamDefs (targetnode } a) = \text{length outs} \)
by \(- \text{(rule ParamDefs-return-target-length)} \)
with \(\langle \text{try e = Node (targetnode } a) \rangle \) show \(\text{?case by simp} \)
qed simp-all

next
fix \(n \) \(V \)
assume \(\text{CFG.CFG.valid-node src try} \)
\((\text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit} a) \)
and \(V \in \text{set (lift-ParamDefs ParamDefs } n) \)

hence \((n = \text{NewEntry}) \vee n = \text{NewExit}) \vee (\exists m. n = \text{Node } m \land \text{valid-node } m) \)
by(\text{auto elim:lift-valid-edge.cases simp:CFG.valid-node-def})

thus \(V \in \text{lift-Def Def Entry Exit } H \) \(L \) \(n \) \(\text{apply} \)
proof(\text{erule disjE}+)

assume \(n = \text{NewEntry} \)
with \(\langle V \in \text{set (lift-ParamDefs ParamDefs } n) \rangle \) show \(\text{?thesis by simp} \)

next
assume \(n = \text{NewExit} \)
with \(\langle V \in \text{set (lift-ParamDefs ParamDefs } n) \rangle \) show \(\text{?thesis by simp} \)

next
assume \(\exists m. n = \text{Node } m \land \text{valid-node } m \)
then obtain \(m \) where \(n = \text{Node } m \land \text{valid-node } m \) by blast
from \((n = \text{Node } m ; V \in \text{set (lift-ParamDefs ParamDefs } n) \)

have \(V \in \text{set (ParamDefs } m) \) by simp
with \(\langle \text{valid-node } m \rangle \) have \(V \in \text{Def } m \) by(\text{rule ParamDefs-in-Def})
with \(\langle n = \text{Node } m \rangle \) show \(\text{?thesis by (fastforce intro:lift-Def-node)} \)
qed

next
fix \(a \) \(Q \) \(r \) \(p \) \(fs \) \(\text{ins outs } V \)
assume \(\text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit } a \)
and \(knd a = Q; r \rightarrow p; fs \) \(\in \text{set procs} \) and \(V \in \text{set ins} \)

thus \(V \in \text{lift-Def Def Entry Exit } H \) \(L \) \(\langle \text{try a} \) \rangle
proof(\text{induct rule:lift-valid-edge.induct})

\text{case (lev-edge } a \ e \)
from \((e = \langle \text{Node (sourcenode } a), \text{kind } a, \text{Node (targetnode } a) \rangle) \)
\(\langle \text{knd } e = Q; r \rightarrow p; fs \) \)

have \(\text{knd } a = Q; r \rightarrow p; fs \) by simp
from \(\langle \text{valid-edge } a \rangle \) \(\langle \text{knd } a = Q; r \rightarrow p; fs \rangle \) \(\langle p, \text{ins}, \text{outs} \rangle \in \text{set procs} \) \(\langle V \in \text{set ins} \) \)

have \(V \in \text{Def (targetnode } a) \) by(\text{rule ins-in-Def})
from \((e = \langle \text{Node (sourcenode } a), \text{kind } a, \text{Node (targetnode } a) \rangle) \)

have \(\text{try } e = \text{Node (targetnode } a) \) by simp
with \(\langle V \in \text{Def (targetnode } a) \rangle \) show \(\text{?case by (fastforce intro:lift-Def-node)} \)
qed simp-all

next
fix \(a \) \(Q \) \(r \) \(p \) \(fs \)
assume \(\text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit } a \)
and \(\text{knd } a = Q; r \rightarrow p; fs \)
thus \( \text{lift-Def} \text{ Def Entry Exit H L (src a)} = \{\} \)

proof (induct rule: \text{lift-valid-edge}.induct)

\begin{enumerate}
  \item case (\text{loc-edge} a e)
    \begin{enumerate}
      \item show ?case
    \end{enumerate}
  \end{enumerate}

proof (rule ccontr)

\begin{enumerate}
  \item assume \( \text{lift-Def} \text{ Def Entry Exit H L (src e)} \neq \{\} \)
    \begin{enumerate}
      \item then obtain \( x \) where \( x \in \text{lift-Def} \text{ Def Entry Exit H L (src e)} \) by blast
    \end{enumerate}
  \end{enumerate}

\begin{enumerate}
  \item from \( \langle e = (\text{Node (sourcenode a), kind a, Node (targetnode a)}); \text{kind e} = \rangle \)
    \begin{enumerate}
      \item have \( \text{sourcenode a} \neq \text{Entry} \)
    \end{enumerate}
  \end{enumerate}

qed

next

fix \( n \ V \)

assume \( \text{CFG.CFG valido-node src try} \)

(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) \( n \)

and \( V \in \bigcup \text{set (lift-ParamUses ParamUses n)} \)

hence \( ((n = \text{NewEntry}) \lor n = \text{NewExit}) \lor (\exists m. n = \text{Node m} \land \text{valid-node m}) \)

by (auto elim: lift-valid-edge_cases simp: \( \text{CFG.valid-node-def} \))

thus \( V \in \text{lift-Use Use Entry Exit H L n apply} \)

proof (erule disjE)+

\begin{enumerate}
  \item assume \( n = \text{NewEntry} \)
    \begin{enumerate}
      \item with \( \langle V \in \bigcup \text{set (lift-ParamUses ParamUses n)} \rangle \) show ?thesis by simp
    \end{enumerate}
  \end{enumerate}

next

\begin{enumerate}
  \item assume \( n = \text{NewExit} \)
    \begin{enumerate}
      \item with \( \langle V \in \bigcup \text{set (lift-ParamUses ParamUses n)} \rangle \) show ?thesis by simp
    \end{enumerate}
  \end{enumerate}

next

\begin{enumerate}
  \item assume \( \exists m. n = \text{Node m} \land \text{valid-node m} \)
  \item then obtain \( m \) where \( n = \text{Node m} \land \text{valid-node m} \) by blast
  \item from \( \langle V \in \bigcup \text{set (ParamUses m)} \rangle \) show ?thesis by simp
  \item have \( V \in \bigcup \text{set (ParamUses m)} \) by simp
    \begin{enumerate}
      \item with \( \langle \text{valid-node m} \rangle \) have \( V \in \text{Use m} \) by (rule ParamUses-in-Use)
      \item with \( \langle n = \text{Node m} \rangle \) show ?thesis by (fastforce intro: lift-Use-node)
    \end{enumerate}
  \end{enumerate}

qed

next
fix a Q p f ins outs V
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and knd a = Q<–p<–f and (p, ins, outs) ∈ set procs and V ∈ set outs
thus V ∈ lift-Use Use Entry Exit H L (src a)
proof(induct rule:lift-valid-edge.induct)
  case (lee-edge a e)
    have knd a = Q<–p<–f by simp
    from valid-edge a ⟨knd a = Q<–p<–f⟩ ⟨(p, ins, outs) ∈ set procs⟩ ⟨V ∈ set outs⟩
    have V ∈ Use (sourcenode a) by(rule outs-in-Use)
    from ⟨e = (Node (sourcenode a), kind a, Node (targetnode a))⟩
    have src e = Node (sourcenode a) by simp
    with ⟨V ∈ Use (sourcenode a)⟩ show ?case by(fastforce intro:lift-Use-node)
  qed simp-all
next
fix a V s
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and V ∉ lift-Def Def Entry Exit H L (src a) and intra-kind (knd a)
and pred (knd a) s
thus state-val (transfer (knd a) s) V = state-val s V
proof(induct rule:lift-valid-edge.induct)
  case (lee-edge a e)
    from ⟨e = (Node (sourcenode a), kind a, Node (targetnode a))⟩
    ⟨intra-kind (knd e)⟩ ⟨pred (knd e) s⟩
    have intra-kind (knd a) and pred (knd a) s
    and knd e = kind a and src e = Node (sourcenode a) by simp-all
    from ⟨V ∉ lift-Def Def Entry Exit H L (src e)⟩ ⟨src e = Node (sourcenode a)⟩
    have V ∉ Def (sourcenode a) by (auto dest: lift-Def-node)
    from valid-edge a ⟨V ∉ Def (sourcenode a)⟩ ⟨intra-kind (knd a)⟩
    ⟨pred (knd a) s⟩
    have state-val (transfer (knd a) s) V = state-val s V
    by(rule CFG-intra-edge-no-Def-equal)
    with ⟨knd e = kind a⟩ show ?case by simp
next
  case (lee-Entry-edge e)
    from ⟨e = (NewEntry, (λs. True)⟨s⟩, Node Entry)⟩ ⟨pred (knd e) s⟩
    show ?case by(cases s) auto
next
  case (lee-Exit-edge e)
    from ⟨e = (Node Exit, (λs. True)⟨s⟩, NewExit)⟩ ⟨pred (knd e) s⟩
    show ?case by(cases s) auto
next
  case (lee-Entry-Exit-edge e)
    from ⟨e = (NewEntry, (λs. False)⟨s⟩, NewExit)⟩ ⟨pred (knd e) s⟩
    have False by(cases s) auto
    thus ?case by simp
qed
next

fix a s s'

assume assms:lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit

\forall V \in \text{lift-Use} \ Use \text{Entry} \text{Exit} H L (\text{src} a). \ state-val s V = state-val s' V

intra-kind (kind a) \ pred (kind a) s pred (kind a) s'

show \forall V \in \text{lift-Def} \text{Def} \text{Entry} \text{Exit} H L (\text{src} a).

state-val (\text{transfer} (\text{kind} a) s) V = state-val (\text{transfer} (\text{kind} a) s') V

proof

fix V assume V \in \text{lift-Def} \text{Def} \text{Entry} \text{Exit} H L (\text{src} a)

with assms

show state-val (\text{transfer} (\text{kind} a) s) V = state-val (\text{transfer} (\text{kind} a) s') V

proof (induct rule:lift-valid-edge.induct)

case (lbe-edge a e)

from (e = (\text{Node} (\text{sourcenode} a), \text{kind} a, \text{Node} (\text{targetnode} a))):

\langle \text{intra-kind} (\text{kind} e) \rangle \text{ have intra-kind} (\text{kind} a) \text{ by simp}

show ?thesis

proof (cases Node (\text{source} a) = \text{Node} \text{Entry})

case True

hence \text{source} a = \text{Entry} \text{ by simp}

from Entry-Exit-edge obtain a' where valid-edge a'

and \text{source} a' = \text{Entry} \text{ and target} a' = \text{Exit}

and kind a' = (\lambda s. \text{False}) by blast

have \exists Q. kind a = (Q) by simp

proof (cases targetnode a = \text{Exit})

case True

with \langle \text{valid-edge} a \rangle \langle \text{valid-edge} a' \rangle \langle \text{source} a = \text{Entry} \rangle

\langle \text{source} a' = \text{Entry} \rangle \langle \text{target} a' = \text{Exit} \rangle

have a = a' by (fastforce dest:edge-det)

with \langle \text{kind} a' = (\lambda s. \text{False}) \rangle show ?thesis by simp

next

case False

with \langle \text{valid-edge} a \rangle \langle \text{valid-edge} a' \rangle \langle \text{source} a = \text{Entry} \rangle

\langle \text{source} a' = \text{Entry} \rangle \langle \text{target} a' = \text{Exit} \rangle

\langle \text{intra-kind} (\text{kind} a) \rangle \langle \text{kind} a' = (\lambda s. \text{False}) \rangle

show ?thesis by (auto dest:deterministic simp:intra-kind-def)

qed

from True (V \in \text{lift-Def} \text{Def} \text{Entry} \text{Exit} H L (\text{src} e)) \text{ Entry-empty}

\langle e = (\text{Node} (\text{source} a), \text{kind} a, \text{Node} (\text{target} a)) \rangle

have V \in H by (fastforce elim:lift-Def-set.cases)

from True \langle e = (\text{Node} (\text{source} a), \text{kind} a, \text{Node} (\text{target} a)) \rangle

\langle \text{source} a \neq \text{Entry} \lor \text{target} a \neq \text{Exit} \rangle

have \forall V \in H. V \in \text{lift-Use} \ Use \text{Entry} \text{Exit} H L (\text{src} e)

by (fastforce intro:lift-Use-High)

with \forall V \in \text{lift-Use} \ Use \text{Entry} \text{Exit} H L (\text{src} e).

state-val s V = state-val s' V; (V \in H)

have state-val s V = state-val s' V by simp

with \langle e = (\text{Node} (\text{source} a), \text{kind} a, \text{Node} (\text{target} a)) \rangle

\exists Q. \text{ kind} a = (Q) \langle \text{pred} (\text{kind} e) s \rangle \langle \text{pred} (\text{kind} e) s' \rangle

51
show ?thesis by{cases s,auto,cases s',auto}

next
case False
{ fix V' assume V' ∈ Use (sourcenode a)
  with (e = (Node (sourcenode a), kind a, Node (targetnode a)))
  have V' ∈ lift-Use Use Entry Exit H L (src e)
    by (fastforce intro:lift-Use-node)
}
with (∀ V∈lift-Use Use Entry Exit H L (src e).
  state-val s V = state-val s' V)
have ∀ V∈Use (sourcenode a). state-val s V = state-val s' V
by fastforce
from ⟨valid-edge a⟩ this ⟨pred (kind a) s⟩ ⟨pred (kind e) s'⟩
⟨e = (Node (sourcenode a), kind a, Node (targetnode a))⟩
⟨intra-kind (kind e)⟩
have ∀ V ∈ Def (sourcenode a). state-val (transfer (kind a) s) V =
  state-val (transfer (kind a) s') V
by (erule CFG-intra-edge-transfer-uses-only-Use,auto)
from (V ∈ lift-Def Def Entry Exit H L (src e), False
  ⟨e = (Node (sourcenode a), kind a, Node (targetnode a))⟩)
have V ∈ Def (sourcenode a) by (fastforce elim:lift-Def-set.cases)
with (∀ V ∈ Def (sourcenode a). state-val (transfer (kind a) s) V =
  state-val (transfer (kind a) s') V)
⟨e = (Node (sourcenode a), kind a, Node (targetnode a))⟩
show ?thesis by simp
qed
next
case (lve-Entry-edge e)
from (V ∈ lift-Def Def Entry Exit H L (src e)
  ⟨e = (NewEntry, (λs. True), Node Entry)⟩)
have False by (fastforce elim:lift-Def-set.cases)
thus ?case by simp
next
case (lve-Exit-edge e)
from (V ∈ lift-Def Def Entry Exit H L (src e)
  ⟨e = (Node Exit, (λs. True), NewExit)⟩)
have False
by (fastforce elim:lift-Def-set.cases intro!:Entry-noteq-Exit simp:Exit-empty)
thus ?case by simp
next
case (lve-Entry-Exit-edge e)
thus ?case by (cases s) auto
qed
qed
next
fix a s s'
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and pred (kind a) s and snd (hd s) = snd (hd s')
and ∀ V∈lift-Use Use Entry Exit H L (src a). state-val s V = state-val s' V
and length s = length s'
thus pred (knd a) s'

proof (induct rule: lift-valid-edge.induct)
case (lee-edge a e)
from ⟨ e = (Node (sourcenode a), kind a, Node (targetnode a)), pred (knd e) s' ⟩

have pred (kind a) s and src e = Node (sourcenode a) by simp-all
from ⟨ src e = Node (sourcenode a), \( \forall V \in \text{lift-Use Use Entry Exit H L (src e). state-val s V = state-val s' V} \) ⟩

have \( \forall V \in \text{Use (sourcenode a). state-val s V = state-val s' V} \) by (auto dest: lift-Use-node)
from ⟨ valid-edge a ⟩ ⟨ pred (knd a) s ⟩ ⟨ snd (hd s) = snd (hd s') ⟩
this ⟨ length s = length s' ⟩
have pred (kind a) s' by (rule CFG-edge-Uses-pred-equal)
with ⟨ e = (Node (sourcenode a), kind a, Node (targetnode a)) ⟩
show ?case by simp
next

case (lee-Entry-edge e)
thus ?case by (cases s') auto
next

case (lee-Exit-edge e)
thus ?case by (cases s') auto
next

case (lee-Entry-Exit-edge e)
thus ?case by (cases s) auto
qed simp-all

next
fix a Q r p fs ins outs
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and knd a = Q: r → p fs and (p, ins, outs) ∈ set procs
thus length fs = length ins

proof (induct rule: lift-valid-edge.induct)
case (lee-edge a e)
from ⟨ e = (Node (sourcenode a), kind a, Node (targetnode a)), knd e = Q: r → p fs ⟩

have kind a = Q: r → p fs by simp
from ⟨ valid-edge a, knd a = Q: r → p fs ⟩ ⟨ (p, ins, outs) ∈ set procs ⟩
show ?case by (rule CFG-call-edge-length)
qed simp-all

next
fix a Q r p fs a' Q' r' p' fs' s s'
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and knd a = Q: r → p fs and knd a' = Q': r' → p' fs'
and lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a'
and src a = src a' and pred (knd a) s and pred (knd a') s
from ⟨ lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a ⟩
\( knd a = Q: r → p fs \) \( \langle \text{pred (knd a)} \rangle s \)

obtain x where a:a = (Node (sourcenode x), kind x, Node (targetnode x))
and valid-edge x and src a = Node (sourcenode x)
and kind \( x = Q \mapsto r \mapsto p \mapsto \) and pred (kind \( x \))

by (fastforce elim: lift-valid-edge_cases)

from lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a'  
  \( \text{knd} a' = Q \mapsto r \mapsto p \mapsto fs' \) \( \langle \text{pred (kind} a' \rangle s \)

obtain \( x' \) where \( a':a = (\text{Node (sourcenode} x'), \text{kind} x', \text{Node (targetnode} x')) \)
and valid-edge \( x' \) and \( \text{sroenode} a' = \text{Node (sourcenode} x') \)
and kind \( x' = Q \mapsto r \mapsto p \mapsto fs' \) and pred (kind \( x' \)) s

by (fastforce elim: lift-valid-edge_cases)

from \( \langle \text{src} a = \text{Node (sourcenode} x) \rangle \langle \text{src} a' = \text{Node (sourcenode} x') \rangle \)
\( \text{src} a = \text{src} a' \)

have sourcenode \( x = \text{sourcenode} x' \) by simp

from valid-edge \( x \) (kind \( x = Q \mapsto r \mapsto p \mapsto fs \) \( \langle \text{kind} x' = Q \mapsto r \mapsto p \mapsto fs' \rangle \)
\( \langle \text{sourcenode} x = \text{sourcenode} x' \rangle \langle \text{pred (kind} x) s \rangle \langle \text{pred (kind} x) s \rangle \)

have \( x = x' \) by (rule CFG-call-determ)

with \( a \) a' show \( a = a' \) by simp

next

fix \( a \) \( r \) \( p \) \( fs \) \( i \) \( \text{ins} \) \( \text{outs} \) \( s \) \( s' \)

assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and \( \text{knd} a = Q \mapsto r \mapsto p \mapsto fs \) \( \langle \text{i < length} \text{ins} \rangle \text{and} \) \( (p, \text{ins}, \text{outs}) \in \text{set procs} \)
and pred (kind \( a \)) \( s \) and pred (kind \( a \)) \( s' \)
and \( \forall V \in \text{ lift-ParamUses ParamUses (src} a) \) \( ! \) i. state-val \( s \) \( V \) = state-val \( s' \)

V

thus \( \text{params} fs \) (state-val \( s) ! \) i = \( \text{CFG.params} fs \) \( \text{(state-val} s') ! \) i

proof (induct rule: lift-valid-edge_induct)

case (lev-edge a e)

from \( e = (\text{Node (sourcenode} a), \text{kind} a, \text{Node (targetnode} a)) \) \( \langle \text{kind} e = Q \mapsto r \mapsto p \mapsto fs \rangle \)
\( \langle \text{pred (kind} e) s \rangle \langle \text{pred (kind} e) s' \rangle \)

have \( \text{knd} a = Q \mapsto r \mapsto p \mapsto fs \) \( \langle \text{pred (kind} a) s \rangle \langle \text{pred (kind} a) s' \rangle \)
and \( \text{src} e = \text{Node (sourcenode} a) \)

by simp_all

from \( \forall V \in \text{ lift-ParamUses ParamUses (src} e) \) \( ! \) i. state-val \( s \) \( V \) = state-val \( s' \) \( V \)

\( \langle \text{src} e = \text{Node (sourcenode} a) \rangle \)

have \( \forall V \in \text{ ParamUses (sourcenode} a) \) ! i. state-val \( s \) \( V \) = state-val \( s' \) \( V \)

by simp

with \( \langle \text{valid-edge} a \rangle \langle \text{kind} a = Q \mapsto r \mapsto p \mapsto fs \rangle \langle i < \text{length} \text{ins} \rangle \)
\( \langle (p, \text{ins}, \text{outs}) \in \text{set procs} \rangle \langle \text{pred (kind} a) s \rangle \langle \text{pred (kind} a) s' \rangle \)

show \( \forall \) case by (rule CFG-call-edge-params)

qed simp_all

next

fix \( a \) \( Q \) \( p \) \( f' \) \( \text{ins} \) \( \text{outs} \) \( \text{cf} \)

assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and \( \text{knd} a = Q \mapsto r \mapsto p \mapsto f' \) \( \langle \text{(p, ins, outs)} \in \text{set procs} \rangle \)

thus \( f' \) \( \text{cf} \) \( \text{cf} = \text{cf'} (\text{lift-ParamDefs ParamDefs} \text{(try} a) [:=>] \text{map} \text{cf} \text{outs}) \)

proof (induct rule: lift-valid-edge_induct)

case (lev-edge a e)

from \( e = (\text{Node (sourcenode} a), \text{kind} a, \text{Node (targetnode} a)) \) \( \langle \text{kind} e = Q \mapsto r \mapsto p \mapsto f' \rangle \)

54
have kind \( a = Q' \leftrightarrow p' \) and \( \text{trg} e = \text{Node} \ (\text{targetnode} a) \) by simp-all
from \( \langle \text{valid-edge} a \rangle \ (\text{kind} a = Q' \leftrightarrow p' \) \( (p, \text{ins}, \text{outs}) \in \text{set procs} \)
have \( f' \ cf \ cf' = cf' \ (\text{ParamDefs} \ (\text{targetnode} a) [:=] \text{map} cf \ \text{outs}) \)
by \( \text{(rule CFG-return-edge-fun)} \)
with \( \langle \text{trg} e = \text{Node} \ (\text{targetnode} a) \rangle \) show ?case by simp
qed simp-all
next

fix \( a a' \)
assume \( \text{lift-valid-edge} \ (\text{valid-edge} \ \text{source} \ \text{node} \ \text{target} \ \text{node} \ \text{kind} \ \text{Entry} \ \text{Exit} \ a \)
and \( \text{lift-valid-edge} \ (\text{valid-edge} \ \text{source} \ \text{node} \ \text{target} \ \text{node} \ \text{kind} \ \text{Entry} \ \text{Exit} \ a' \)
and \( \text{src} a = \text{src} a' \) and \( \text{trg} a \neq \text{trg} a' \)
and \( \text{intra-kind} \ (\text{knd} a) \) and \( \text{intra-kind} \ (\text{knd} a') \)
thus \( \exists Q Q', \ \text{knd} a = (Q) \land \text{knd} a' = (Q') \land \\
\forall s. (Q s \rightarrow \neg Q' s) \land (Q' s \rightarrow \neg Q s) \)
proof \( \langle \text{(induct rule:lift-valid-edge.induct)} \rangle \)
case \( \text{lve-edge} a e \) from \( \langle \text{lift-valid-edge} \ (\text{valid-edge} \ \text{source} \ \text{node} \ \text{target} \ \text{node} \ \text{kind} \ \text{Entry} \ \text{Exit} \ a) \)
\( \text{valid-edge} a \) \langle \( e = \langle \text{Node} \ (\text{source} a), \text{kind} a, \text{Node} \ (\text{target} a) \rangle \rangle \langle \text{src} e = \text{src} a' \) : \( \text{try} e \neq \text{try} a' \) \langle \text{intra-kind} \ (\text{knd} e) \rangle \langle \text{intra-kind} \ (\text{knd} a') \rangle \rangle 
show ?case
proof \( \langle \text{(induct rule:lift-valid-edge.induct)} \rangle \)
case \( \text{lve-edge} \) thus ?case by \( \langle \text{auto dest:deterministic} \rangle \)
next

case \( \text{lve-Exit-edge} e' \) from \( \langle \text{lift-valid-edge} \ (\text{valid-edge} \ \text{source} \ \text{node} \ \text{target} \ \text{node} \ \text{kind} \ \text{Entry} \ \text{Exit} \ a) \)
\( \text{valid-edge} a' \) \langle \( e' = \langle \text{Node} \ (\text{source} a), \text{kind} a, \text{Node} \ (\text{target} a) \rangle \rangle \langle \text{src} e = \text{src} a' \) \langle \text{try} e = \text{try} a' \) \langle \text{intra-kind} \ (\text{knd} e) \rangle \langle \text{intra-kind} \ (\text{knd} a') \rangle \rangle 
have \( \text{source} \text{enode} a = \text{Exit} \) by simp
with \( \langle \text{valid-edge} a \rangle \) have \( \text{False} \) by \( \langle \text{rule Exit-source} \rangle \)
thus ?case by simp
qed auto
qed \( \langle \text{fastforce elim:lift-valid-edge.cases} \rangle + \)
qed

lemma lift-CFGExit:
assumes \( \text{wf:CFGExit-wf} \ (\text{source} \ \text{node} \ \text{target} \ \text{node} \ \text{kind} \ \text{valid-edge} \ \text{Entry} \ \text{get-proc} \)
\( \text{get-return-edges} \ \text{procs} \ \text{Main} \ \text{Exit} \ \text{Def} \ \text{Use} \ \text{ParamDefs} \ \text{ParamUses} \)
and \( \text{pd:Postdomination} \ (\text{source} \ \text{node} \ \text{target} \ \text{node} \ \text{kind} \ \text{valid-edge} \ \text{Entry} \ \text{get-proc} \)
\( \text{get-return-edges} \ \text{procs} \ \text{Main} \ \text{Exit} \)
shows \( \text{CFGExit} \ (\text{src} \ \text{trg} \ \text{knd} \ \langle \text{lift-valid-edge} \ (\text{valid-edge} \ \text{source} \ \text{node} \ \text{target} \ \text{node} \ \text{kind} \ \text{Entry} \ \text{Exit}) \ \text{NewEntry} \ \langle \text{lift-get-proc} \ \text{get-proc} \ \text{Main} \text{NewExit} \ \langle \text{lift-get-return-edges} \ \text{valid-edge} \ \text{source} \ \text{node} \ \text{target} \ \text{node} \ \text{kind} \ \text{procs} \ \text{Main} \ \text{NewExit} \ \langle \text{interpret} \ \text{CFGExit-wf} \ (\text{source} \ \text{node} \ \text{target} \ \text{node} \ \text{kind} \ \text{valid-edge} \ \text{Entry} \ \text{get-proc} \)
\( \text{get-return-edges} \ \text{procs} \ \text{Main} \ \text{Exit} \ \text{Def} \ \text{Use} \ \text{ParamDefs} \ \text{ParamUses} \)
by \( \langle \text{rule wf} \rangle \)
interpret Postdomination sourcenode targetnode kind valid-edge Entry get-proc get-return-edges procs Main Exit
by(rule pd)

interpret CFG:CFG src trg knd
lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit NewEntry
lift-get-proc get-proc Main
lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind procs Main
by(fastforce intro:lift-CFG wf pd)

show \ ?thesis

proof
  fix a assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
  and src a = NewExit
  thus False by(fastforce elim:lift-valid-edge_cases)

next
show lift-get-proc get-proc Main NewExit = Main by simp

next
fix a Q p f
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and knd a = Q↩p f and trg a = NewExit
thus False by(fastforce elim:lift-valid-edge_cases)

next
show \exists a. lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a ∧
src a = NewEntry ∧ trg a = NewExit ∧ knd a = (\lambda s. False)✓
  by(fastforce intro:lev-Entry-Exit-edge)

qed

lemma lift-CFGExit-wf:
assumes wf:CFGExit-wf sourcenode targetnode kind valid-edge Entry get-proc get-return-edges procs Main Exit Def Use ParamDefs ParamUses
and pd:Postdomination sourcenode targetnode kind valid-edge Entry get-proc get-return-edges procs Main Exit

shows CFGExit-wf src trg knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewEntry
(lift-get-proc get-proc Main)
(lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind)
procs Main NewExit (lift-Def Def Entry Exit H L) (lift-Use Use Entry Exit H L)
(lift-ParamDefs ParamDefs) (lift-ParamUses ParamUses)

proof –
interpret CFGExit-wf sourcenode targetnode kind valid-edge Entry get-proc get-return-edges procs Main Exit Def Use ParamDefs ParamUses
by(rule wf)

interpret Postdomination sourcenode targetnode kind valid-edge Entry get-proc get-return-edges procs Main Exit
by(rule pd)

interpret CFG-wf:CFG-wf src trg knd
lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit NewEntry

56
lift-get-proc get-proc Main
lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind
procs Main lift-Def Def Entry Exit H L lift-Use Use Entry Exit H L
lift-ParamDefs ParamDefs lift-ParamUses ParamUses
by (fastforce intro: lift-CFG-wf wf pd)
interpret CFGExit:CFGExit src trg knd
lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit NewEntry
lift-get-proc get-proc Main
lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind
procs Main NewExit
by (fastforce intro: lift-CFGExit wf pd)
show ?thesis
proof
  show lift-Def Def Entry Exit H L NewExit = {}
  lift-Use Use Entry Exit H L NewExit = {}
  by (fastforce elim: lift-Def-set.cases lift-Use-set.cases)
qed
qed

3.2.2 Lifting the SDG

lemma lift-Postdomination:
assumes wf:CFGExit-wf sourcenode targetnode kind valid-edge Entry get-proc
get-return-edges procs Main Exit Def Use ParamDefs ParamUses
and pd:Postdomination sourcenode targetnode kind valid-edge Entry get-proc
get-return-edges procs Main Exit
and inner:CFGExit.inner-node sourcenode targetnode valid-edge Entry Exit nx
shows Postdomination src trg knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewEntry
(lift-get-proc get-proc Main)
(lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind)
procs Main NewExit
proof –
interpret CFGExit-wf sourcenode targetnode kind valid-edge Entry get-proc
get-return-edges procs Main Exit Def Use ParamDefs ParamUses
by (rule wf)
interpret Postdomination sourcenode targetnode kind valid-edge Entry get-proc
get-return-edges procs Main Exit
by (rule pd)
interpret CFGExit:CFGExit src trg knd
lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit NewEntry
lift-get-proc get-proc Main
lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind
procs Main NewExit
by (fastforce intro: lift-CFGExit wf pd)
fix m assume valid-node m
then obtain a where valid-edge a and m = sourcenode a ∨ m = targetnode a
by (auto simp: valid-node-def)
from \( \langle m = \text{sourcenode } a \lor m = \text{targetnode } a \rangle \) 

have \( \text{CFG CFG.valid-node src trg} \)

(\( \text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit} \) \(\text{Node } m\))

proof

assume \( m = \text{sourcenode } a \)

show \( ?\text{thesis} \)

proof (cases \( m = \text{Entry} \))

\hfill case True

have \( \text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit} \)

(\(\text{NewEntry},(\lambda s. \text{True}),\text{Node Entry} \)) \(\text{by} (\text{fastforce intro:lve-Entry-edge})\)

with \( m = \text{Entry} \): show \( ?\text{thesis} \) \(\text{by} (\text{fastforce simp:CFGExit.valid-node-def})\)

next

\hfill case False

with \( m = \text{sourcenode } a \) \(\text{valid-edge } a \)

have \( \text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit} \)

(\(\text{Node } (\text{sourcenode } a),\text{kind } a,\text{Node } (\text{targetnode } a) \))

\(\text{by} (\text{fastforce intro:lve-edge})\)

with \( m = \text{sourcenode } a \): show \( ?\text{thesis} \) \(\text{by} (\text{fastforce simp:CFGExit.valid-node-def})\)

qed

next

assume \( m = \text{targetnode } a \)

show \( ?\text{thesis} \)

proof (cases \( m = \text{Exit} \))

\hfill case True

have \( \text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit} \)

(\(\text{Node Exit},(\lambda s. \text{True}),\text{NewExit} \)) \(\text{by} (\text{fastforce intro:lve-Exit-edge})\)

with \( m = \text{Exit} \): show \( ?\text{thesis} \) \(\text{by} (\text{fastforce simp:CFGExit.valid-node-def})\)

next

\hfill case False

with \( m = \text{targetnode } a \) \(\text{valid-edge } a \)

have \( \text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit} \)

(\(\text{Node } (\text{sourcenode } a),\text{kind } a,\text{Node } (\text{targetnode } a) \))

\(\text{by} (\text{fastforce intro:lve-edge})\)

with \( m = \text{targetnode } a \): show \( ?\text{thesis} \) \(\text{by} (\text{fastforce simp:CFGExit.valid-node-def})\)

qed

}\}

note \( \text{lift-valid-node} = \text{this} \)

\{\text{fix } n \text{ as } n' \text{ cs } m \text{ m'}\}

\hfill assume \( \text{valid-path-aux } cs \text{ as } \text{and } m \Rightarrow s \Rightarrow \text{m'} \text{ and } \forall c \in \text{set } cs. \text{ valid-edge } c \)

\hfill and \( m \neq \text{Entry } \lor m' \neq \text{Exit} \)

\hfill hence \( \exists cs' \text{ es. } \text{CFG CFG.valid-path-aux knod} \)

(lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind) \(\text{cs' es} \land \)

list-all2 \((\lambda c c'. c' = (\text{Node } (\text{sourcenode } c),\text{kind } c,\text{Node } (\text{targetnode } c))) \text{ cs cs'} \land \text{CFG CFG.path src trg} \)

(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) \(\text{(Node } m \text{ es } (\text{Node } m'))\)

\text{proof} (\text{induct } \text{arbitrary:} m \text{ rule:} \text{vpa-induct})

\hfill case \( \text{vpa-empty } cs \)
from \( m \rightarrow^* m' \) have \[ simp \] \( m = m' \) by fastforce
from \( m \rightarrow^* m' \) have valid-node \( m \) by (rule path-valid-node)
generate \( cs' \) where \( cs' = \)
map (\( \lambda c. \) Node (sourcenode \( c \)), kind \( c \), Node (targetnode \( c \))) \( cs \) by simp

hence list-all2
\((\lambda c. c'. c' = (\text{Node (sourcenode} \( c \)), \text{kind} \( c \), \text{Node (targetnode} \( c \)))) \( cs \) \( cs' \)
by (simp add: list-all2-cov-all-nth)

with \( \langle \text{valid-node} \( m \rangle \) show \?case
apply (rule-tac \( x = cs' \) in exI)
apply (rule-tac \( x = [] \) in exI)
by (fastforce intro: CFGExit.empty-path lift-valid-node)

next

case (vpa-intra \( cs \ a \ a s \) )

note \( IH = \emptyset \ (m \rightarrow^* m'; \ \forall c \in \text{set} \ cs. \ \text{valid-edge} \ c; \ m \neq \text{Entry} \lor m' \neq \text{Exit}) \)

\( \Rightarrow \)
\( \exists cs' \ es. \ \text{CFG.valid-path-aux knd} \)
\((\text{lift-get-return-edges} \ \text{get-return-edges} \ \text{valid-edge} \ \text{sourcenode targetnode kind}) \ cs' \ es \land \)
\( \text{list-all2} \ (\lambda c. c'. c' = (\text{Node (sourcenode} \( c \)), \text{kind} \( c \), \text{Node (targetnode} \( c \)))) \ cs \ cs' \land \text{CFG.path src trg} \)
\( \text{lift-valid-edge} \ \text{valid-edge} \ \text{sourcenode} \ \text{targetnode kind} \ \text{Entry Exit} \)
\((\text{Node} \ m) \ es (\text{Node} \ m') \)
from \( m \ a \neq a \rightarrow^* m' \) have \( m = \text{sourcenode} \ a \) and valid-edge \( a \)
andtargetnode \( a \ a s \rightarrow^* m' \) by (auto elim: path-split-Cons)

show \?case

proof (cases \( \text{sourcenode} \ a = \text{Entry} \land \text{targetnode} \ a = \text{Exit} \))
case True
with \( \langle m = \text{sourcenode} \ a \rangle \langle m \neq \text{Entry} \lor m' \neq \text{Exit} \rangle \)
\( \text{have} \ m' \neq \text{Exit} \) by simp
from True have targetnode \( a = \text{Exit} \) by simp
with \( \langle \text{targetnode} \ a = a \rightarrow^* m' \rangle \) have \( m' = \text{Exit} \)
by \( \text{drule} \ \text{path-Exit-source}, \text{auto} \)
with \( \langle m' \neq \text{Exit} \rangle \) have False by simp

thus \?thesis by simp

next

case False
\
let \( ?e = (\text{Node (sourcenode} \( a \)), \text{kind} \( a \), \text{Node (targetnode} \( a \))) \)
from False (valid-edge \( a \))
\( \text{have} \ \text{lift-valid-edge} \ \text{valid-edge} \ \text{sourcenode} \ \text{targetnode kind} \ \text{Entry Exit} \ ?e \)
by (fastforce intro: lve-edge)
\( \text{have} \ \text{targetnode} \ a \neq \text{Entry} \)

proof
assume targetnode \( a = \text{ Entry} \)
with \( \text{valid-edge} \ a \) show False by (rule Entry-target)

qed

hence targetnode \( a \neq \text{Entry} \lor m' \neq \text{Exit} \) by simp
from \( IH \ [\langle \text{targetnode} \ a = a \rightarrow^* m' \rangle \ \forall c \in \text{set} \ cs. \ \text{valid-edge} \ c \) this]

obtain \( cs' \ es \)
where \( \text{valid-path} : \text{CFG.valid-path-aux knd} \)
lift-get-return-edges get-return-edges valid-edge sourcenode
targetnode kind) cs cs’
and list:list-all2
(λc c’. c’ = (Node (sourcenode c), kind c, Node (targetnode c))) cs cs’
and path:CFG.path src trg
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(Node (targetnode a)) cs cs’ by blast
from (intra-kind (kind a)) valid-path have CFG.valid-path-aux knd
(lift-get-return-edges get-return-edges valid-edge sourcenode
targetnode kind) cs’ (∀e#es) by (fastforce simp:intra-kind-def)
moreover
from path (m = sourcenode a)
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit ?e)
have CFG.path src trg
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(Node m) (∃e#es) (Node m’) by (fastforce intro:CFGExit.Cons-path)
ultimately show ?thesis using list by blast
qed
next
case (vpa-Call cs a as Q r p fs)
note IH = (∀m. [m = as→∗ m’; ∀c∈set (a ≠ cs). valid-edge c; m ≠ Entry] ∨ m’ ≠ Exit] →
∃cs’ es. CFG.valid-path-aux knd
(lift-get-return-edges get-return-edges valid-edge sourcenode
targetnode kind) cs’ es ∧
list-all2 (λc c’. c’ = (Node (sourcenode c), kind c, Node (targetnode c)))
(a ≠ cs) cs’ ∧ CFG.path src trg
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(Node m) es (Node m’)
from (m = sourcenode a)
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
have valid-edge a by simp
proof
assume sourcenode a = Entry
with (valid-edge a) (kind a = Q:r→p|fs)
show False by (rule Entry-no-call-source)
qed
with (valid-edge a)
have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit ?e
by (fastforce intro:lve-edge)
have targetnode a ≠ Entry
proof
assume targetnode a = Entry
with (valid-edge a) show False by (rule Entry-target)
qed
hence targetnode a ≠ Entry ∨ m’ ≠ Exit by simp
from IH \([OF \langle targetnode a \rightarrow s \mapsto m \mapsto \forall c \in set \ (a \neq c), \ valid-edge c \rangle \ this] \)

obtain \(cs' \ es\)

where valid-path:CFG.valid-path-aux knd

(lift-get-return-edges get-return-edges valid-edge sourcenode

targetnode kind) cs' es

and list:list-all2

(\(\lambda c . c' = (\text{Node (sourcenode c), kind c, Node (targetnode c)}) \)) \((a \neq cs) \ cs'\)

and path:CFG.path src try

(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)

(Node (targetnode a)) es (Node m') by blast

from list obtain \(csx\) where \(cs' = cs \# csx\)

and \(ex:ex = (\text{Node (sourcenode a), kind a, Node (targetnode a)})\)

and list':list-all2

(\(\lambda c . c' = (\text{Node (sourcenode c), kind c, Node (targetnode c)}) \)) cs csx

by (fastforce simp:list-all2-Cons1)

from valid-path cx \((cs' = cs \# csx) \ (kind a = Q;r\mapsto p;fs)\)

have CFG.valid-path-aux knd

(lift-get-return-edges get-return-edges valid-edge sourcenode

targetnode kind) csx (?e#es) by simp

moreover

from path \(m = sourcenode a\)

(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit ?e)

have CFG.path src try

(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)

(Node m') (?e#es) (Node m') by (fastforce intro:CFGExit.Cons-path)

ultimately show ?case using list' by blast

next

case (vpa-ReturnEmpty cs a as Q p f)

note IH = (\(\forall m . \ m \mapsto a \# \ m' \mapsto \forall c \in set \ [\text{valid-edge c; m \neq Entry} \lor m' \neq Exit]\))

\(\exists cs' \ es. \ CFG.\text{valid-path-aux knd}

(lift-get-return-edges get-return-edges valid-edge sourcenode

targetnode kind) cs' es \land\)

list-all2 (\(\lambda c . c' = (\text{Node (sourcenode c), kind c, Node (targetnode c)}) \))

\(\square cs' \land CFG.\text{path src try}

(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)

(Node m) es (Node m')

from \(m \mapsto a \# \ mapsto m' \mapsto \text{have m = sourcenode a and valid-edge a}\)

and targetnode a \(-\mapsto a\# \ mapsto m' \mapsto\text{(auto elim:path-split-Cons)}\)

let \(\forall c = (\text{Node (sourcenode a),kind a,Node (targetnode a)})\)

have targetnode a \# Exit

proof

assume targetnode a = Exit

with \(\langle \text{valid-edge a} \rangle \langle \text{kind a = Q;r}\mapsto p;fs\rangle\) show False by (rule Exit-no-return-target)

qed

with \(\langle \text{valid-edge a} \rangle\)

have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit ?e

by (fastforce intro:live-edge)

have targetnode a \# Exit
proof 
  assume targetnode a = Entry 
  with ⟨valid-edge a⟩ show False by (rule Entry-target) 
qed 
hence targetnode a ≠ Entry ∨ m' ≠ Exit by simp 
from IH[OF ⟨targetnode a − as−→ m'⟩ - this] obtain es 
  where valid-path:CFG.valid-path-aux knd 
    (lift-get-return-edges get-return-edges valid-edge sourcenode 
      targetnode kind) [] es 
    and path:CFG.path src try 
    (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) 
    (Node (targetnode a)) es (Node m') by auto 
from valid-path ⟨kind a = Q←pf⟩ 
  have CFG.valid-path-aux knd 
    (lift-get-return-edges get-return-edges valid-edge sourcenode 
      targetnode kind) [] (?e#es) by simp 
moreover 
from path ⟨m = sourcenode a⟩ 
  ⟨lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit ?e⟩ 
  have CFG.path src try 
    (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) 
    (Node m) (?e#es) (Node m') by (fastforce intro:CFGExit.Cons-path) 
ultimately show ?case using ⟨cs = []⟩ by blast 
next 
case (vpa-ReturnCons cs a as Q p f c' cs') 
  note IH = (\∀m. [m − as−→ m' ; ∀c∈set cs'. valid-edge c ; m ≠ Entry ∨ m'] 
    =⇒ \∃csx es.CFG.valid-path-aux knd 
      (lift-get-return-edges get-return-edges valid-edge sourcenode 
        targetnode kind) csx es ∧ 
      list-all2 (λc c'. c' = (Node (sourcenode c), kind c, Node (targetnode c))) 
      cs' csx ∧ CFG.path src try 
      (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) 
      (Node m) es (Node m')) 
from ⟨m − a # as−→ m'⟩ have m = sourcenode a and valid-edge a 
  and targetnode a − as−→ m' by (auto elim:path-split-Cons) 
from ∀c∈set cs. valid-edge c ; ⟨cs = c' ≠ cs'⟩ 
  have valid-edge c' and ∀c∈set cs'. valid-edge c by simp-all 
let ?e = (Node (sourcenode a),kind a,Node (targetnode a)) 
have targetnode a ≠ Exit 
proof 
  assume targetnode a = Exit 
  with ⟨valid-edge a⟩ ⟨kind a = Q←pf⟩ show False by (rule Exit-no-return-target) 
qed 
with ⟨valid-edge a⟩ have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit ?e 
  by (fastforce intro:lve-edge) 
have targetnode a ≠ Entry 
proof 

62
assume targetnode \( a = \text{Entry} \)

with \( \langle \text{valid-edge } a \rangle \) show False by (rule Entry-target)

qed

hence targetnode \( a \neq \text{Entry} \lor m' \neq \text{Exit} \) by simp

from IH[OF \( \langle \text{targetnode } a \rangle \rightarrow m' \lor \forall c \in \text{set } cs', \text{valid-edge } c \rangle \) this]

obtain csx es

where \( \text{valid-path:CFG}.\text{valid-path-aux } knd \) (lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind) csx es

and \( \text{list: list-all2} \)

\((\lambda c. c' = (\text{Node } (\text{sourceinode } c), \text{kind } c, \text{Node } (\text{targetnode } c))) \) cs' csx

and \( \text{path:CFG}.\text{path src trg} \)

(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)

(\( \text{Node } (\text{targetnode } a) \rangle \) es (\( \text{Node } m' \rangle \) by blast

from \( \langle \text{valid-edge } c' \rangle (a \in \text{get-return-edges } c' \rangle \)

have \( ?e \in \text{lift-get-return-edges } \text{get-return-edges } \text{valid-edge } \text{sourceinode } \text{targetnode } \text{kind } \text{Entry } \text{Exit} \)

\( \text{targetnode } \text{kind } (\text{Node } (\text{sourceinode } c'), \text{kind } c', \text{Node } (\text{targetnode } c')) \)

by (fastforce intro: lift-get-return-edgesI)

with \( \text{valid-path } \langle \text{kind } a = Q' \rangle \)

have \( \text{CFG}.\text{valid-path-aux } knd \)

(lift-get-return-edges get-return-edges valid-edge sourceinode targetnode kind)

((\( \text{Node } (\text{sourceinode } c'), \text{kind } c', \text{Node } (\text{targetnode } c') \rangle) \# csx) \ (\?e \# es)

by simp

moreover

from \( \text{list } \langle \text{cs } = c' \# cs' \rangle \)

have \( \text{list-all2} \)

\((\lambda c. c' = (\text{Node } (\text{sourceinode } c), \text{kind } c, \text{Node } (\text{targetnode } c))) \) cs

((\( \text{Node } (\text{sourceinode } c'), \text{kind } c', \text{Node } (\text{targetnode } c') \rangle) \# csx)

by simp

moreover

from \( \text{path } \langle m = \text{sourceinode } a \rangle \)

(lift-valid-edge valid-edge sourceinode targetnode kind Entry Exit \( ?e \) \)

have \( \text{CFG}.\text{path src trg} \)

(lift-valid-edge valid-edge sourceinode targetnode kind Entry Exit)

(\( \text{Node } m \rangle \) \( ?e \# es \) \( \text{Node } m' \rangle \) by (fastforce intro: CFGExit.Cons-path)

ultimately show \( ?\text{case using } \langle \text{kind } a = Q' \rangle \) by blast

qed

hence \( \text{lift-valid-path: } \forall m \rightarrow m' \rightarrow m \neq \text{Entry } \lor m' \neq \text{Exit} [\]

\[\Rightarrow \exists es. \text{ CFG}.\text{CFG}.\text{valid-path'} src trg knd \]

(lift-valid-edge valid-edge sourceinode targetnode kind Entry Exit)

(lift-get-return-edges get-return-edges valid-edge sourceinode targetnode kind)

(\( \text{Node } m \rangle \) es (\( \text{Node } m' \rangle \))

by (fastforce simp: vp-def valid-path-def CFGExit.vp-def CFGExit.valid-path-def)

show \( ?\text{thesis} \)

proof

fix \( n \) assume \( \text{CFG}.\text{CFG}.\text{valid-node } src trg \)

(lift-valid-edge valid-edge sourceinode targetnode kind Entry Exit) \( n \)

hence \((n = \text{NewEntry}) \lor n = \text{NewExit} ) \lor (\exists m. n = \text{Node } m \land \text{valid-node } m) \)
by (auto elim: lift-valid-edge.cases simp: CFGExit.valid-node-def)
thus 3 as. CFG.CFG.valid-path' src trg knd
(lift-valid-edge valid-edge source-node target-node kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge source-node target-node kind)
NewEntry as n apply --
proof (erule disjE)+
assume n = NewEntry
hence CFG.CFG.valid-path' src trg knd
(lift-valid-edge valid-edge source-node target-node kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge source-node target-node kind)
NewEntry [] n
by (fastforce intro: CFGExit.empty-path
  simp: CFGExit.vp-def CFGExit.valid-path-def)
thus ?thesis by blast
next
assume n = NewExit
have lift-valid-edge valid-edge source-node target-node kind Entry Exit
  (NewEntry, (\lambda s. False) _\_, NewExit) by (fastforce intro: lve-Entry-Exit-edge)
  hence CFG.CFG.path src trg
  (lift-valid-edge valid-edge source-node target-node kind Entry Exit)
  NewEntry [(NewEntry, (\lambda s. False) _\_, NewExit)] NewExit
  by (fastforce dest: CFGExit.path-edge)
  with (n = NewExit) have CFG.CFG.valid-path' src trg knd
  (lift-valid-edge valid-edge source-node target-node kind Entry Exit)
  (lift-get-return-edges get-return-edges valid-edge source-node target-node kind)
  NewEntry [(NewEntry, (\lambda s. False) _\_, NewExit)] n
  by (fastforce simp: CFGExit.vp-def CFGExit.valid-path-def)
thus ?thesis by blast
next
assume \exists m. n = Node m \land valid-node m
then obtain m where n = Node m and valid-node m by blast
from (valid-node m)
show ?thesis
proof (cases m rule: valid-node-cases)
  case Entry
  have lift-valid-edge valid-edge source-node target-node kind Entry Exit
    (NewEntry, (\lambda s. True) _\_, Node Entry) by (fastforce intro: lve-Entry-Entry-edge)
  with (m = Entry) (n = Node m) have CFG.CFG.path src trg
    (lift-valid-edge valid-edge source-node target-node kind Entry Exit)
    NewEntry [(NewEntry, (\lambda s. True) _\_, Node Entry)] n
    by (fastforce intro: CFGExit.Cons-path CFGExit.empty-path
      simp: CFGExit.valid-node-def)
  thus ?thesis by (fastforce simp: CFGExit.vp-def CFGExit.valid-path-def)
next
  case Exit
  from inner obtain ax where valid-edge ax and intra-kind (kind ax)
    and inner-node (source-node ax)
    and target-node ax = Exit by (erule inner-node-Exit-edge)
  hence lift-valid-edge valid-edge source-node target-node kind Entry Exit
(Node (sourcenode ax), kind ax, Node Exit)
by (auto intro: lift-valid-edge lve-edge simp: inner-node-def)

hence CFG.path src try
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(Node (sourcenode ax)) [[(Node (sourcenode ax), kind ax, Node Exit)]
(Node Exit)
by (fastforce intro: CFGExit.Cons-path CFGExit.empty-path
    simp: CFGExit.valid-node-def)

with (intra-kind (kind ax))

have slp-edge: CFG._CFG.same-level-path' src try knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge sourcenode
    targetnode kind)
(Node (sourcenode ax)) [[(Node (sourcenode ax), kind ax, Node Exit)]
(Node Exit)
by (fastforce simp: CFGExit.slp-def CFGExit.same-level-path-def
    intra-kind-def)

have sourcenode ax ≠ Exit

proof
assume sourcenode ax = Exit
with ⟨valid-edge ax⟩ show False by (rule Exit-source)
qed

have slp-edge: CFG._CFG.same-level-path' src try knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge sourcenode
    targetnode kind)
(Node (sourcenode ax)) [[(Node (sourcenode ax), kind ax, Node Exit)]
(Node Exit)
by (fastforce intro: lve-Entry-edge)

hence CFG.path src try
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(Node (NewEntry, (λs. True), Node Entry))
by (fastforce intro: lve-Entry-edge)

hence slp-edge': CFG._CFG.same-level-path' src try knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge sourcenode
    targetnode kind)
(Node (NewEntry))[[(NewEntry, (λs. True), Node Entry)]
(Node Entry)
by (fastforce simp: CFGExit.slp-def CFGExit.same-level-path-def)

from ⟨inner-node (sourcenode ax)⟩ have valid-node (sourcenode ax)
    by (rule inner-is-valid)
then obtain asx where Entry → asx→ₐ⁺ sourcenode ax
by (fastforce dest: Entry-path)
with (sourcenode ax ≠ Exit)

have ∃ es, CFG._CFG.valid-path' src try knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge sourcenode
    targetnode kind) (Node Entry) es (Node (sourcenode ax))
by (fastforce intro: lift-valid-path)
then obtain es where CFG._CFG.valid-path' src try knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge sourcenode
    targetnode kind) (Node Entry) es (Node (sourcenode ax))
by blast

65
with slp-edge have CFG.CFG.valid-path’ src trg knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge sourcenode
targetnode kind)
(Node Entry) (es@[((Node (sourcenode ax),kind ax,Node Exit))]) (Node Exit)
by -(rule CFGExit.ep-slp-Append)
with slp-edge’ have CFG.CFG.valid-path’ src trg knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge sourcenode

targetnode kind) NewEntry
([(NewEntry,(λs. True)\(\),Node Entry)])@
(es@[((Node (sourcenode ax),kind ax,Node Exit))]) (Node Exit)
by(rule CFGExit.slp-vp-Append)
with ⟨m = Exit⟩ ⟨n = Node m⟩ show ?thesis by simp blast
next
case inner
have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
(NewEntry,(λs. True)\(\),Node Entry) by (fastforce intro:lv-ex-Entry-edge)
hence CFG.path src trg
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(NewEntry) [(NewEntry,(λs. True)\(\),Node Entry)] (Node Entry)
by (fastforce simp:CFGExit.Cons-path CFGExit.empty-path
simpl:CFGExit.valid-node-def)
hence slp-edge:CFG.CFG.same-level-path’ src trg knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge sourcenode

targetnode kind)
(NewEntry) [(NewEntry,(λs. True)\(\),Node Entry)] (Node Entry)
by (fastforce simp:CFGExit.slp-def CFGExit.same-level-path-def)
from (valid-node m) obtain as where Entry − as→ \(\)* m
by (fastforce dest:lv-Entry-path)
with ⟨inner-node m⟩
have ∃ es. CFG.CFG.valid-path’ src trg knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge sourcenode

targetnode kind) (Node Entry) es (Node m)
by (fastforce intro:lv-lift-valid-path simp:inner-node-def)
then obtain es where CFG.CFG.valid-path’ src trg knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge sourcenode

targetnode kind) (Node Entry) es (Node m) by blast
with slp-edge have CFG.CFG.valid-path’ src trg knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge sourcenode

targetnode kind) NewEntry [(NewEntry,(λs. True)\(\),Node Entry)]@es
(Node m)
by (rule CFGExit.slp-vp-Append)
with ⟨n = Node m⟩ show ?thesis by simp blast
qed
qed

next

fix n assume CFG.CFG.valid-node src try
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) n
hence ((n = NewEntry) ∨ n = NewExit) ∨ (∃m. n = Node m ∧ valid-node m)

by(auto elim:lift-valid-edge.cases simp:CFGExit.valid-node-def)

thus ∃as. CFG.CFG.valid-path′ src try knd
(lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind)

proof(erule disjE)+

assume n = NewEntry
have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
(NewEntry,(λs. False),NewExit) by(fastforce intro:lve-Entry-Exit-edge)

hence CFG.CFG.path src try
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
NewEntry [(NewEntry,(λs. False),NewExit)] NewExit
by(fastforce dest:CFGExit.path-edge)

with (n = NewEntry) have CFG.CFG.valid-path′ src try knd
(lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind)
n [(NewEntry,(λs. False),NewExit)] NewExit
by(fastforce simp:CFGExit.empty-path simp:CFGExit.valid-path-def)

thus ?thesis by blast

next

assume n = NewExit

hence CFG.CFG.valid-path′ src try knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind)
n [] NewExit

by(fastforce intro:CFGExit.empty-path simp:CFGExit.valid-path-def)

thus ?thesis by blast

next

assume ∃m. n = Node m ∧ valid-node m

then obtain m where n = Node m and valid-node m by blast
from (valid-node m)
show ?thesis
proof(cases m rule:valid-node-cases)

case Entry

from inner obtain ax where valid-edge ax and intra-kind (kind ax)
and inner-node (targetnode ax) and sourcenode ax = Entry
by(erule inner-node-Entry-edge)

hence lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
(Node Entry,kind ax,Node (targetnode ax))
by(auto intro:lift-valid-edge.lve-edge simp:inner-node-def)

hence CFG.path src try
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(Node Entry) [(Node Entry, kind ax, Node (targetnode ax))]
(Node (targetnode ax))
by (fastforce intro: CFGExit.Cons-path CFGExit.empty-path
simp: CFGExit.valid-node-def)
with ⟨intra-kind (kind ax)⟩
have slp-edge: CFG_CFG.same-level-path' src trg knd
(lift-valid-edge valid-edge source-node target-node kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge source-node
target-node kind)
(Node Entry) [(Node Entry, kind ax, Node (targetnode ax))]
(Node (targetnode ax))
by (fastforce simp: CFGExit.slp-def CFGExit.same-level-path-def
intra-kind-def)
have targetnode ax ≠ Entry
proof
  assume targetnode ax = Entry
  with ⟨valid-edge ax⟩ show False by (rule Entry-target)
qed
have lift-valid-edge valid-edge source-node target-node kind Entry Exit
(Node Exit, (λs. True) , NewExit) by (fastforce intro: lift-Exit-edge)

hence CFG path src trg
(lift-valid-edge valid-edge source-node target-node kind Entry Exit)
(Node Exit) [(Node Exit, (λs. True), NewExit)] NewExit
by (fastforce intro: CFGExit.Cons-path CFGExit.empty-path
simp: CFGExit.valid-node-def)

hence slp-edge': CFG_CFG.same-level-path' src trg knd
(lift-valid-edge valid-edge source-node target-node kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge source-node
target-node kind)
(Node Exit) [(Node Exit, (λs. True), NewExit)] NewExit
by (fastforce simp: CFGExit.slp-def CFGExit.same-level-path-def)
from ⟨inner-node (targetnode ax)⟩ have valid-node (targetnode ax)
by (rule inner-is-valid)
then obtain asx where
  targetnode ax − asx → ,* Exit
by (fastforce dest: Exit-path)
with ⟨targetnode ax ≠ Entry⟩
have ∃ es. CFG_CFG.valid-path' src trg knd
(lift-valid-edge valid-edge source-node target-node kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge source-node
target-node kind) (Node (targetnode ax)) es (Node Exit)
by (fastforce intro: lift-valid-path)
then obtain es where
  CFG_CFG.valid-path' src trg knd
(lift-valid-edge valid-edge source-node target-node kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge source-node
target-node kind) (Node (targetnode ax)) es (Node Exit) by blast
with slp-edge have CFG_CFG.valid-path' src trg knd
(lift-valid-edge valid-edge source-node target-node kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge source-node
target-node kind)
(Node Entry) \(((\text{Node Entry}, \text{kind ax}, \text{Node (targetnode ax)}))@es\) (Node Exit)

by (rule CFGExit.slp-vp-Append)
with slp-edge' have CFG.CFG.valid-path' src try knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind) (Node Entry)
\(((\text{Node Entry}, \text{kind ax}, \text{Node (targetnode ax)}))@es)@\[\text{(Node Exit},(\text{\lambda s. True})_\rightarrow,\text{NewExit})\] NewExit
by (rule CFGExit.slp-vp-Append)

next

\text{case Exit}
have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
(Node Exit,(\lambda s. True)_\rightarrow,\text{NewExit}) by (fastforce intro:lev-Exit-edge)
with \(m = \text{Exit}\) \((n = \text{Node m})\) have CFG.CFG.path src try
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
\((\text{Node Exit},(\lambda s. True)_\rightarrow,\text{NewExit})\) NewExit
by (fastforce intro:CFGExit.Cons-path CFGExit.empty-path simp:CFGExit.valid-node-def)

thus \(?thesis\) by (fastforce simp:CFGExit.vp-def CFGExit.valid-path-def)

\text{next}

\text{case inner}

have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
(Node Exit,(\lambda s. True)_\rightarrow,\text{NewExit}) by (fastforce intro:lev-Exit-edge)

hence CFG.path src try
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(Node Exit) \[\text{((Node Exit},(\lambda s. True)_\rightarrow,\text{NewExit})\] NewExit
by (fastforce intro:CFGExit.Cons-path CFGExit.empty-path simp:CFGExit.valid-node-def)

hence slp-edge:CFG.CFG.same-level-path' src try knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind)
(Node Exit) \[\text{((Node Exit},(\lambda s. True)_\rightarrow,\text{NewExit})\] NewExit
by (fastforce simp:CFGExit.slp-def CFGExit.same-level-path-def)

from \(\text{valid-node m}\) obtain as \text{where}\ \forall m \rightarrow (\text{\lambda \ast \rightarrow \ast}) \text{Exit}
by (fastforce dest:Exit-path)

\text{with (inner-node m)}

have \(\exists es, \text{CFG.CFG.valid-path'} src try knd\)
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind) (Node m) es (Node Exit)
by (fastforce intro:lift-valid-path simp:inner-node-def)

then obtain es as \text{where}\ CFG.CFG.valid-path' src try knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind) (Node m) es (Node Exit) by blast

with slp-edge have CFG.CFG.valid-path' src try knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge sourcenode
targetnode kind) (Node m) (es@[\(\text{Node Exit,} (\lambda s. \text{True}) \cup, \text{NewExit}\)]) NewExit
by \(\text{rule CFGExit, cp-slp-Append}\) with \((n = \text{Node m})\) show \(\text{thesis}\) by simp blast
qed
qed
next
fix \(n \, n'\)
assume \(\text{method-exit1:CFGExit.CFGExit.method-exit src knd}\)
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewExit \(n\)
and \(\text{method-exit2:CFGExit.CFGExit.method-exit src knd}\)
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewExit \(n'\)
and \(\text{lift-eq:lift-get-proc get-proc Main n = lift-get-proc get-proc Main n'}\)
from \(\text{method-exit1}\) show \(n = n'\)
proof (rule CFGExit.method-exit-cases)
assume \(n = \text{NewExit}\)
from \(\text{method-exit2}\) show \(\text{thesis}\)
proof (rule CFGExit.method-exit-cases)
assume \(n' = \text{NewExit}\)
with \((n = \text{NewExit})\) show \(\text{thesis}\) by simp
next
fix \(a \, Q \, f \, p\)
assume \(n' = \text{src a}\)
and \(\text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a}\)
and \(\text{knd a = Q \leftarrow p f}\)

hence \(\text{lift-get-proc get-proc Main } (\text{src a}) = p\)
by \(\text{rule CFGExit.get-proc-return}\)
with \(\text{CFGExit.get-proc-Exit lift-eq } (\text{n' = src a})\) \((n = \text{NewExit})\)
have \(p = \text{Main}\) by simp
with \((\text{knd a = Q \leftarrow p f})\) have \(\text{knd a = Q \leftarrow p f}\) \(\text{Main}\) by simp
with \((\text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a})\)
have \(\text{False}\) by (rule CFGExit.Main-no-return-source)
thus \(\text{thesis}\) by simp
qed
next
fix \(a \, Q \, f \, p\)
assume \(n = \text{src a}\)
and \(\text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a}\)
and \(\text{knd a = Q \leftarrow p f}\)
then obtain \(x\) where \(\text{valid-edge x and src a = Node } (\text{sourcenode x})\)
and \(\text{knd x = Q \leftarrow p f}\)
by (fastforce elim:lift-valid-edge, cases)

hence \(\text{method-exit } (\text{sourcenode x})\) by (fastforce simp:method-exit-def)
from \(\text{method-exit2}\) show \(\text{thesis}\)
proof (rule CFGExit.method-exit-cases)
assume \(n' = \text{NewExit}\)
from \((\text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a})\)
\((\text{knd a = Q \leftarrow p f})\)
have \(\text{lift-get-proc get-proc Main (src a) = p}\)
  by -(rule CFGExit.get-proc-return)
with CFGExit.get-proc-Exit lift-eq (n = src a) (n' = NewExit)
have \(p = \text{Main}\) by simp
with \(\langle \text{knd a = Q左手\(\rightarrow\)f}\rangle\) have \(\text{knd a = Q左手\(\rightarrow\)Mainf}\) by simp
with \(\langle \text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a}\rangle\) have \(\text{False}\) by (rule CFGExit.Main-no-return-source)
thus \(?\text{thesis}\) by simp

next

fix \(a' Q' f' p'\)
assume \(n' = \text{src a'}\)
  and \(\text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a'}\)
  and \(\text{knd a'} = Q'左手\(\rightarrow\)p'f'\)
then obtain \(x'\) where \(\text{valid-edge x'}\) and \(\text{src a'} = \text{Node (sourcenode x')}\)
  and \(\text{knd x'} = Q'左手\(\rightarrow\)p'f'\)
  by (fastforce elim: lift-valid-edge.cases)
hence \(\text{method-exit (sourcenode x')}\) by (fastforce simp: method-exit-def)
with \(\langle \text{method-exit (sourcenode x)}\rangle\) lift-eq (n = src a) (n' = src a')
  (\(\text{src a = Node (sourcenode x)}\)) (\(\text{src a' = Node (sourcenode x')}\))
have \(\text{sourcenode x = sourcenode x'}\) by (fastforce intro: method-exit-unique)
with (\(\text{src a = Node (sourcenode x)}\)) (\(\text{src a' = Node (sourcenode x')}\))
  (\(n = \text{src a}\)) (\(n' = \text{src a'}\))
show \(?\text{thesis}\) by simp
qed
qed
qed

lemma lift-SDG:
assumes SDG:SDG sourcenode targetnode kind valid-edge Entry get-proc
get-return-edges procs Main Exit Def Use ParamDefs ParamUses
and inner:CFGExit.inner-node sourcenode targetnode kind valid-edge Entry Exit nx
shows SDG src try knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewEntry
(lift-get-proc get-proc Main)
(lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind)
procs Main NewExit (lift-Def Def Entry Exit H L) (lift-Use Entry Exit H L)
(lift-ParamDefs ParamDefs) (lift-ParamUses ParamUses)

proof

interpret SDG sourcenode targetnode kind valid-edge Entry get-proc
get-return-edges procs Main Exit Def Use ParamDefs ParamUses
by (rule SDG)
have wf:CFGExit-wf sourcenode targetnode kind valid-edge Entry get-proc
get-return-edges procs Main Exit Def Use ParamDefs ParamUses
by (unfold-locales)
have pd:Postdomination sourcenode targetnode kind valid-edge Entry get-proc
get-return-edges procs Main Exit
by (unfold-locales)
interpret \texttt{wf}':\texttt{CFGExit-wf} src trg knd
\begin{itemize}
\item lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit NewEntry
\item lift-get-proc get-proc Main
\item lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind
\end{itemize}
\item procs Main NewExit lift-Def Def Entry Exit H L lift-Use Use Entry Exit H L
\item lift-ParamDefs ParamDefs lift-ParamUses ParamUses
\item by (fastforce intro:lift-CFGExit-wf wf pd)
interpret \texttt{pd}':\texttt{Postdomination} src trg knd
\begin{itemize}
\item lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit NewEntry
\item lift-get-proc get-proc Main
\item lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind
\end{itemize}
\item procs Main NewExit
\item by (fastforce intro:lift-Postdomination wf pd inner)
show ?thesis by (unfold-locales)
qed

3.2.3 Low-deterministic security via the lifted graph

\textbf{lemma Lift-NonInterferenceGraph:}
\begin{itemize}
\item fixes valid-edge and sourcenode and targetnode and kind and Entry and Exit
\item and get-proc and get-return-edges and procs and Main
\item and Def and Use and ParamDefs and ParamUses and H and L
\item defines lve: lve \equiv lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
\item and lget-proc:lget-proc \equiv lift-get-proc get-proc Main
\item and lget-return-edges:lget-return-edges \equiv lift-get-return-edges get-return-edges sourcenode targetnode kind
\item and lDef:lDef \equiv lift-Def Def Entry Exit H L
\item and lUse:lUse \equiv lift-Use Use Entry Exit H L
\item and lParamDefs:lParamDefs \equiv lift-ParamDefs ParamDefs
\item and lParamUses:lParamUses \equiv lift-ParamUses ParamUses
\item assumes SDG: SDG sourcenode targetnode kind valid-edge Entry get-proc
\item get-return-edges procs Main Exit Def Use ParamDefs ParamUses
\item and inner: CFGExit.inner-node sourcenode targetnode valid-edge Entry Exit \nec
\item and H \cap L = \{\} and H \cup L = UNIV
\end{itemize}
\item shows NonInterferenceInterGraph src trg knd lve NewEntry lget-proc
\item lget-return-edges procs Main NewExit lDef lUse lParamDefs lParamUses H L
\item (Node Entry) (Node Exit)
proof
\begin{itemize}
\item interpret SDG sourcenode targetnode kind valid-edge Entry get-proc
\item get-return-edges procs Main Exit Def Use ParamDefs ParamUses
\item by (rule SDG)
\item interpret SDG': SDG src trg knd lve NewEntry lget-proc lget-return-edges
\item procs Main NewExit lDef lUse lParamDefs lParamUses
\item by (fastforce intro:lift-SDG SDG inner simp: lve lget-proc lget-return-edges lDef
\item lUse lParamDefs lParamUses)
\end{itemize}
\item show ?thesis
proof
\begin{itemize}
\item fix a assume lve a and src a = NewEntry
\item thus try a = NewExit \lor try a = Node Entry
\end{itemize}
by (fastforce elim: lift-valid-edge.cases simp:lve)
next
show ∃ a. lve a ∧ src a = NewEntry ∧ trg a = Node Entry ∧ knd a = (λs. True)
by (fastforce intro:lve-Entry-edge simp:lve)
next
fix a assume lve a and trg a = Node Entry
from ⟨lve a⟩
have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
by (simp add:lve)
from this ⟨trg a = Node Entry⟩
show src a = NewEntry
proof (induct rule: lift-valid-edge.induct)
  case (lve-edge a e)
  from ⟨e = (Node (sourcenode a), kind a, Node (targetnode a))⟩
  ⟨trg e = Node Entry⟩
  have targetnode a = Entry by simp
  with ⟨valid-edge a⟩ have False by (rule Entry-target)
  thus ?case by simp
qed simp-all
next
fix a assume lve a and trg a = NewExit
thus src a = NewEntry ∨ src a = Node Exit
by (fastforce elim: lift-valid-edge.cases simp:lve)
next
show ∃ a. lve a ∧ src a = NewExit ∧ trg a = NewExit ∧ knd a = (λs. True)
by (fastforce intro:lve-Exit-edge simp:lve)
next
fix a assume lve a and src a = Node Exit
from ⟨lve a⟩
have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
by (simp add:lve)
from this ⟨src a = Node Exit⟩
show trg a = NewExit
proof (induct rule: lift-valid-edge.induct)
  case (lve-edge a e)
  from ⟨e = (Node (sourcenode a), kind a, Node (targetnode a))⟩
  ⟨src e = Node Exit⟩
  have sourcenode a = Exit by simp
  with ⟨valid-edge a⟩ have False by (rule Exit-source)
  thus ?case by simp
qed simp-all
next
from lDef show lDef (Node Entry) = H
by (fastforce elim: lift-Def-set.cases intro:lift-Def-High)
next
from Entry-noteq-Exit lUse show lUse (Node Entry) = H
by (fastforce elim: lift-Use-set.cases intro:lift-Use-High)
next
from Entry-noteq-Exit lUse show lUse (Node Exit) = L
  by (fastforce elim:lift-Use-set.cases intro:lift-Use-Low)
next
  from \H \cap L = \{} \show \H \cap L = \{} \.
next
  from \H \cup L = \text{UNIV} \show \H \cup L = \text{UNIV} .
qed
qed
end

References


interference Java bytecode verifier. In ESOP 2007, volume 4421 of


object-sensitive information flow control based on program dependence
2009.

[5] F. Kammlüer. Formalizing non-interference for a simple bytecode lan-

[6] A. Sabelfeld and D. Sands. A per model of secure information flow in
sequential programs. Higher Order Symbolic Computation, 14(1):59–91,

[7] G. Snelting and D. Wasserrab. A correctness proof for the Vol-
pano/Smith security typing system. In G. Klein, T. Nipkow, and
entries/VolpanoSmith.shtml, September 2008. Formal proof develop-
ment.


74
[9] D. Wasserrab. Backing up slicing: Verifying the interprocedural two-
isa-afp.org/entries/HRB-Slicing.shtml, September 2009. Formal proof
development.

[10] D. Wasserrab, D. Lohner, and G. Snelting. On PDG-based noninterfer-