Slicing Guarantees Information Flow
Noninterference

Daniel Wasserrab
September 13, 2023

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 fulfil 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 =

get-proc get-return-edges procs Main Exit Def Use ParamDefs ParamUses
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 set and ParamUses :: 'node ⇒ '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:
[valid-edge a; sourcenode a = (-Entry-)]
⇒ targetnode a = (-Exit-) ∨ targetnode a = (-High-)
and High-target-Entry-edge:
∃ a. valid-edge a ∧ sourcenode a = (-Entry-) ∧ targetnode a = (-High-) ∧
kind a = (λs. True)✓
and Entry-predecessor-of-High:
[valid-edge a; targetnode a = (-High-)] ⇒ sourcenode a = (-Entry-)
and Exit-edge-Entry-or-Low: [valid-edge a; targetnode a = (-Exit-)]
⇒ sourcenode a = (-Entry-) ∨ sourcenode a = (-Low-)
and Low-source-Exit-edge:
∃ a. valid-edge a ∧ sourcenode a = (-Low-) ∧ targetnode a = (-Exit-) ∧
kind a = (λs. True)✓
and Exit-successor-of-Low:
[valid-edge a; sourcenode a = (-Low-)] ⇒ 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, (-Low-) = (-Exit-) 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-) \(\rightarrow\) \(n\) and \(\text{inner-node } n\)
obtains \(a^\prime\) as' where \(as = a^\prime\#as^\prime\) and \((-\text{High-}) \rightarrow\) \(n\)
and \(\text{kind } a' = (\lambda s. \text{True})\)

proof (atomize-elim)
from \((-\text{Entry-}) \rightarrow\) \(n\) \(<\text{inner-node } n\>)
show \(\exists a^\prime\) as'. \(as = a^\prime\#as^\prime\) \& \((-\text{High-}) \rightarrow\) \(n\) \& \(\text{kind } a' = (\lambda s. \text{True})\)

proof (induct \(n'\) \(\equiv\) (-Entry-) \(\equiv\) \(n\) rule: path.induct)
case (Cons-path \(n''\) as \(n^\prime a\))
from \((n^\prime = \rightarrow\) \(n\)\) \(\langle\text{inner-node } n\rangle\) have \(n'' \neq (-\text{Exit-})\)
        by (fastforce simp: inner-node-def)
with \((\text{valid-edge } a) \langle\text{source-node } a = (-\text{Entry-})\rangle \langle\text{target-node } a = n''\rangle\)
have \(n'' = (-\text{High-})\) by \((-\text{drule Entry-edge-Exit-or-High})\)
from High-target-Entry-edge
obtain \(a^\prime\) where \(\text{valid-edge } a^\prime\) and \(\text{source-node } a^\prime = (-\text{Entry-})\)
        and \(\text{target-node } a^\prime = (-\text{High-})\) and \(\text{kind } a^\prime = (\lambda s. \text{True})\)
        by blast
with \((\text{valid-edge } a) \langle\text{source-node } a = (-\text{Entry-})\rangle \langle\text{target-node } a = n''\rangle\)
        \(\langle n'' = (-\text{High-})\rangle\)
        have \(a = a^\prime\) by (auto dest: edge-det)
with \((n^\prime = \rightarrow\) \(n\)\) \(\langle n'' = (-\text{High-})\rangle \langle\text{kind } a' = (\lambda s. \text{True})\rangle\) show ?case
        by blast
qed fastforce

lemma Exit-path-Low-path:
assumes \(n = \rightarrow\) \(\langle\text{-Exit-}\rangle\) and \(\text{inner-node } n\)
obtains \(a^\prime\) as' where \(as = as^\prime\#a^\prime\) and \(n = \rightarrow\) \(\langle\text{-Low-}\rangle\)
and \(\text{kind } a' = (\lambda s. \text{True})\)

proof (atomize-elim)
from \((n = \rightarrow\) \(\langle\text{-Exit-}\rangle\))
show \(\exists as^\prime a^\prime\) as' \& \(n = \rightarrow\) \(\langle\text{-Low-}\rangle\) \& \(\text{kind } a^\prime = (\lambda s. \text{True})\)

proof (induct as rule: rev-induct)
case Nil
with \(<\text{inner-node } n\rangle\) show ?case
        by fastforce
next
case (snoe a^\prime as')
from \((n = \rightarrow\) \(\langle\text{-Exit-}\rangle\))
have \(n = \rightarrow\) \(\langle\text{source-node } a^\prime\rangle \text{ and valid-edge } a^\prime\) and \(\text{target-node } a^\prime = (-\text{Exit-})\)
        by (auto elim: path-split-snoe)
        \{ assume \(\text{source-node } a^\prime = (-\text{Entry-})\)
                with \((n = \rightarrow\) \(\langle\text{source-node } a^\prime\rangle\) have \(n = (-\text{Entry-})\)
                by (blast intro!: path-Entry-target)
        with \(<\text{inner-node } n\rangle\) have False by (simp add: inner-node-def) \}
with \((\text{valid-edge } a^\prime\rangle \langle\text{target-node } a^\prime = (-\text{Exit-})\rangle \text{ have }\text{source-node } a^\prime = (-\text{Low-})\)
        by (blast dest!: Exit-edge-Entry-or-Low)
from Low-source-Exit-edge
obtain ax where \(\text{valid-edge } ax\) and \(\text{source-node } ax = (-\text{Low-})\)
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
qed

lemma not-Low-High: V \notin L \Longrightarrow V \in H
using HighLowUNIV
by fastforce

lemma not-High-Low: V \notin H \Longrightarrow V \in 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-) \notin [HRB-slice S]_CFG
shows V ∈ L
proof –
from ⟨V ∈ rv S (CFG-node (-Entry-))⟩ obtain as n'
where ⟨-Entry-⟩ → as, parent-node n'
and n' ∈ HRB-slice S and V ∈ Use_{SDG} n'
and ∀ n'', valid-SDG-node n'' ∧ parent-node n'' ∈ set (sourcenodes as)
→ V \notin Def_{SDG} n'' by (fastforce elim:reE)
from ⟨⟨-Entry-⟩ → as, 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 ∈ Use_{SDG} n'⟩ have False
by (drule SDG-Use-parent-Use, simp add: Entry-empty)
thus ?thesis by simp
next
case Exit
  with \(< V \in \text{Use}_{SDG} n' \> \text{ have False}
  \text{ by } \neg (\text{drule SDG-Use-parent-Use, simp add: Exit-empty})
  \text{ thus } \lnot \text{thesis by simp}
next
case inner
  with \(< \text{Entry} \> - as \mapsto \ast \text{ parent-node } n' \> \text{ obtain } a' \text{ as' where as } = a' \# \text{as'}
  \text{ and } \(< \text{High} \> - as' \mapsto \ast \text{ parent-node } n' \>
  \text{ by (fastforce elim: Entry-path-High-path simp: intra-path-def})
  from \(< \text{Entry} \> - as \mapsto \ast \text{ parent-node } n' \> \langle as = a' \# \text{as'} \rangle
  \text{ have sourcenode } a' = \langle \text{Entry} \rangle \text{ by (fastforce elim: path.cases simp: intra-path-def})
  \text{ show } \lnot \text{thesis}
  \text{ proof (cases as' } = [\])
    case True
    with \langle \text{High} \> - as' \mapsto \ast \text{ parent-node } n' \> \text{ have parent-node } n' = \langle \text{High} \>
    \text{ by (fastforce simp: intra-path-def})
    with \langle n' \in \text{HRB-slice } S \rangle \langle \text{High} \rangle \notin [\text{HRB-slice } S]_{CFG} \rangle
    \text{ have False}
    \text{ thus } \lnot \text{thesis by simp}
next
case False
  with \langle \text{High} \> - as' \mapsto \ast \text{ parent-node } n' \> \text{ have hd } (\text{sourcenodes as'}) = \langle \text{High} \>
  \text{ by (fastforce simp: intra-path-def})
  from False \text{ have hd } (\text{sourcenodes as'}) \in \text{set } (\text{sourcenodes as'})
  \text{ by (fastforce intro: hd-in-set simp: sourcenodes-def)}
  with \langle as = a' \# \text{as'} \rangle \text{ have hd } (\text{sourcenodes as'}) \in \text{set } (\text{sourcenodes as')}
  \text{ by (simp add: sourcenodes-def})
  from \langle \text{hd } (\text{sourcenodes as'}) = \langle \text{High} \rangle \rangle
  \text{ have valid-node } (\text{hd } (\text{sourcenodes as'})) \text{ by simp}
  \text{ have valid-SDG-node } (\text{CFG-node } \langle \text{High} \rangle) \text{ by simp}
  with \langle \text{hd } (\text{sourcenodes as'}) = \langle \text{High} \rangle \rangle
    \langle \text{hd } (\text{sourcenodes as'}) \in \text{set } (\text{sourcenodes as'}) \rangle
  \langle \forall n''. \text{ valid-SDG-node } n''. \text{ parent-node } n'' \in \text{set } (\text{sourcenodes as}) \rangle
  \rightarrow V \notin \text{Def}_{SDG} n''
  \text{ have } V \notin \text{Def } \langle \text{High} \rangle
  \text{ by (fastforce dest: CFG-Def-SDG-Def[OF \langle valid-node } (\text{hd } (\text{sourcenodes as'})))]
  \text{ hence } V \notin \text{ H by (simp add: DefHigh)}
  \text{ thus } \lnot \text{thesis by (rule not-High-Low)}
qed
qed

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

proof

fix $V$ assume $V \in \nu S \ (\CFG-node \ (-\Entry-))$

with $(\text{-High-}) \notin \ [\text{HRB-slice } S]_{\CFG}$ 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, $\CFG-node \ (\text{-High-}) \notin \text{HRB-slice } S$, where $\CFG-node \ (\text{-Low-}) \in S$, 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 $\CFG-node \ (\text{-Low-}) \in S$

shows \[ \text{same-level-path-aux } cs \text{ as ; upd-cs } cs \text{ as } [] ; \text{ same-level-path-aux } cs \text{ as } s' ; \]

$\forall \ c \in \text{set } cs, \text{ valid-edge } c; \ m \ -\text{as} \Rightarrow \text{-Low-}; \ m \ -\text{as}' \Rightarrow \text{-Low-};$

$\forall \ i < \text{length } cs, \forall \ V \in \nu S \ (\CFG-node \ (\text{ sourcenode } (cs!i))).$

$\text{fst } (s!\text{Suc } i) \ V = \text{fst } (s'!\text{Suc } i) \ V; \forall \ i < \text{Suc } (\text{length } cs), \text{snd } (s!i) = \text{snd } (s'!i);$

$\forall \ V \in \nu S \ (\CFG-node \ m). \ \text{state-val } s \ V = \text{state-val } s' \ V;$

$\text{preds } (\text{slice-kinds } S \text{ as } s); \text{preds } (\text{slice-kinds } S \text{ as } s') s';$

$\text{length } s = \text{Suc } (\text{length } cs); \text{length } s' = \text{Suc } (\text{length } cs)]$

$\Longrightarrow \forall \ V \in \text{ Use } (\text{-Low-}). \ \text{state-val } (\text{transfers(slice-kinds } S \text{ as } s) \ s) \ V =$

$\text{state-val } (\text{transfers(slice-kinds } S \text{ as } s') s') \ V$

proof(induct arbitrary;m as' s s' rule:slpa-induct)

case (slpa-empty cs)

from $\langle m - \rangle \Rightarrow \text{-Low-} \rangle$ have $m = \text{-Low-}$ by fastforce

from $\langle m - \rangle \Rightarrow \text{-Low-} \rangle$ have valid-node m

by(rule path-valid-node)+

{ fix $V$ assume $V \in \text{Use } (\text{-Low-})$

moreover

from (valid-node m) $\langle m = \text{-Low-} \rangle$ have $\text{-Low-} 

\Rightarrow (\text{-Low-})$ by(fastforce intro:empty-path simp:intra-path-def)

moreover

from (valid-node m) $\langle m = \text{-Low-} \rangle$ $\langle \CFG-node \ (\text{-Low-}) \in S \rangle$

have $\CFG-node \ (\text{-Low-}) \in \text{HRB-slice } S$

by(fastforce intro:HRB-slice-refl)

ultimately have $V \in \nu S \ (\CFG-node \ m)$

using $\langle m = \text{-Low-} \rangle$

by(auto intro!:rel CFG-Use-SDG-Use simp:sourcenodes-def)

} hence $\forall \ V \in \text{Use } (\text{-Low-}). \ V \in \nu S \ (\CFG-node \ m)$ by simp

show ?case

proof(cases L = { })

case True with UseLow show ?thesis by simp

next
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)

next
case (slpa-intra cs a as)

note IH = (¬m as' s s'. [ upd-cs cs as = []]; same-level-path-aux cs as as';
  ∀ 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)]
  ⇒ (∀ 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)
  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 \text{ have } \text{kind } a = (\lambda s. \text{True}) \text{ by simp}

with \langle \text{targetnode } a = (-\text{Exit}) \rangle \text{ have } \text{targetnode } a - as \rightarrow* (-\text{Low})

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

with \text{False have } \text{False by } -(\text{drule Low-neq-Exit,simp})

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

\text{next}

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

with \langle m - as' \rightarrow* (-\text{Low}) \rangle \text{ have } \text{sourcenode } ax = m \text{ and } \text{valid-edge } ax

\text{and } \text{targetnode } ax - asx \rightarrow* (-\text{Low}) \text{ by } (\text{auto elim: path-split-Cons})

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

\text{obtain cf cfs where } \langle \text{simp}: s = cf \# cfs \rangle \langle \text{cases } s \rangle (\text{auto simp: slice-kinds-def})

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

\text{obtain cf' cfs' where } \langle \text{simp}: s' = cf' \# cfs' \rangle

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

\text{have } \text{intra-kind } (\text{kind } ax)

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

\text{case } (\text{Call } Q \ r \ p \ fs)

\text{have } \text{False}

\text{proof}(\text{cases } \text{sourcenode } a \in \langle \text{HRB-slice } S \rangle_{\text{CFG}})

\text{case } \text{True}

with \langle \text{intra-kind } (\text{kind } a') \rangle \text{ have } \text{slice-kind } S a = \text{kind } a

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

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

\text{have } \text{unique: } \exists! a'. \text{valid-edge } a' \land \text{sourcenode } a' = \text{sourcenode } ax \land

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

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

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

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

\text{and } \text{sourcenode } a' = \text{sourcenode } ax \text{ and } \text{kind } a' = (\lambda cf. \text{False}).

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

\text{with } \langle \text{valid-edge } a \rangle \langle \text{sourcenode } a = m \rangle \langle \text{sourcenode } ax = m \rangle

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

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

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

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

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

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

\text{next}

\text{case } \text{False}

with \langle \text{kind } ax = Q: r \rightarrow p \ fs \rangle \langle \text{sourcenode } a = m \rangle \langle \text{sourcenode } ax = m \rangle

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

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

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

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

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

\text{qed}

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

\text{next}

\text{case } (\text{Return } Q \ p \ f)
from ‹valid-edge ax› ‹kind ax = Qe→pf› ‹valid-edge a› ‹intra-kind (kind a)›
  ‹source-node a = m› ‹source-node ax = m›
have False by -(drule return-edges-only,auto simp:intra-kind-def)
thus ‹?thesis› by simp
qed simp
with ‹same-level-path-aux cs as ax› ‹as' = ax#asx›
have same-level-path-aux cs asx by (fastforce simp:intra-kind-def)
show ‹?thesis›
proof(cases target-node a = target-node 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
  ‹preds (slice-kinds S (a # as')) s›
  ‹preds (slice-kinds S as') s'\ › ‹as' = ax # asx›
have rv ∀ V ∈ rv S (CFG-node (target-node 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) = []› ‹intra-kind (kind a)›
have upd-cs cs as = [] by (fastforce simp:intra-kind-def)
from ‹target-node ax -asx→* (-Low-). (a = ax)›
have target-node a -asx→* (-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 source-node a ∈ [HRB-slice S]CFG) apply auto
apply(fastforce dest:slack-intra-kind-in-slice simp:intra-kind-def)
apply(auto simp:intra-kind-def)
apply(drule slice-kind-Ord) 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 source-node a ∈ [HRB-slice S]CFG) apply auto
apply(fastforce dest:slack-intra-kind-in-slice simp:intra-kind-def)
apply(auto simp:intra-kind-def)
apply(drule slice-kind-Ord) apply auto
by (erule kind-Predicate-notin-slice-slice-kind-Predicate) auto
with cfx ∀ i < Suc (length cs). snd (s!i) = snd (s'!i)
have snds:∀ i<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 rvv cfx cfx' have rvv'∀ i<length cs.
\( \forall V \in \tau S \ (CFG\text{-}node \ (sourcenode \ (cs \ ! \ i))) \). 
\( fst \ (transfer \ (slice\text{-}kind \ S \ a) \ s \ ! \ Suc \ i) \ V = \ 
\)
\( fst \ (transfer \ (slice\text{-}kind \ S \ a) \ s' \ ! \ Suc \ i) \ V \ 
\) by fastforce

from \( \langle \text{preds} \ (slice\text{-}kinds} \ S \ (a \ # \ as) \rangle \ s \) 
have \( \text{preds} \ (slice\text{-}kinds} \ S \ as) \ 
(transfer \ (slice\text{-}kind} \ S \ a) \ s) \ by(simp \ add\text{-}slice\text{-}kinds\text{-}def) 
moreover 
from \( \langle \text{preds} \ (slice\text{-}kinds} \ S \ as' \rangle \ s', \langle as' = ax \ # \ ax \rangle \ (a = ax) \) 
have \( \text{preds} \ (slice\text{-}kinds} \ S \ axz) \ (transfer \ (slice\text{-}kind} \ S \ ax) \ s') 
by(simp add:slice-kinds-def) 
moreover 
from \( \langle \text{valid\text{-}edge} \ a \ \langle \text{ intra\text{-}kind} \ (\text{kind} a) \rangle \rangle \) 
have \( \text{length} \ (transfer \ (slice\text{-}kind} \ S \ a) \ s) = \text{length} \ s' 
by(cases sourcenode a \in \ [HRB\text{-}slice} \ S]_{CFG} \) 
(auto dest:slice\text{-}intra\text{-}kind\text{-}in\text{-}slice slice\text{-}kind\text{-}Upd 
elim:kind\text{-}Predicate\text{-}notin\text{-}slice\text{-}slice\text{-}kind\text{-}Predicate simp:intra\text{-}kind\text{-}def) 
with \( \langle \text{length} \ s = Suc \ (\text{length} \ cs) \rangle \) 
have \( \text{length} \ (transfer \ (slice\text{-}kind} \ S \ ax) \ s) = Suc \ (\text{length} \ cs) 
by simp 
moreover 
from \( \langle a = ax \rangle \ \langle \text{valid\text{-}edge} \ a \ \langle \text{ intra\text{-}kind} \ (\text{kind} a) \rangle \rangle \) 
have \( \text{length} \ (transfer \ (slice\text{-}kind} \ S \ ax) \ s) = \text{length} \ s' 
by(cases sourcenode ax \in \ [HRB\text{-}slice} \ S]_{CFG} \) 
(auto dest:slice\text{-}intra\text{-}kind\text{-}in\text{-}slice slice\text{-}kind\text{-}Upd 
elim:kind\text{-}Predicate\text{-}notin\text{-}slice\text{-}slice\text{-}kind\text{-}Predicate simp:intra\text{-}kind\text{-}def) 
with \( \langle \text{length} \ s' = Suc \ (\text{length} \ cs) \rangle \) 
have \( \text{length} \ (transfer \ (slice\text{-}kind} \ S \ ax) \ s) = Suc \ (\text{length} \ cs) 
by simp 
moreover 
from \( IH(OF \ \langle \text{upd\text{-}cs} \ cs \ as = [] \rangle \ \langle \text{same\text{-}level\text{-}path\text{-}aux} \ cs \ axz \rangle \) 
\(\forall cscs. \ \text{valid\text{-}edge} \ c \ \langle \text{targetnode} \ a \rightarrow ax \ (\text{Low} \rightarrow) \) 
\(\langle \text{targetnode} \ a \rightarrow ax \rightarrow (\text{Low} \rightarrow) \ \rangle \ \text{rvs} \ \text{snds} \ \text{rv} \ \text{calculation} \) 
\(\langle as' = ax \ # \ ax \rangle \ (a = ax) \) 
show \( \text{thesis} \ by(simp \ add\text{-}slice\text{-}kinds\text{-}def) \) 
next 
\( \text{case} \ False \)
from \( \forall i < Suc(\text{length} \ cs) \). \( \text{snd} \ (s! i) = \text{snd} \ (s'! i) \). 
have \( \text{snd} \ (\text{hd} \ s) = \text{snd} \ (\text{hd} \ s') \ by(\text{erule\text{-}tac} \ x=0 \ \text{in} \ allE) \) fastforce 
with \( \langle \text{valid\text{-}edge} \ a \ \langle \text{valid\text{-}edge} \ ax \rangle \ \langle \text{sourcenode} \ a = m \rangle \) 
\(\langle \text{sourcenode} \ ax = m \rangle \ \langle as' = ax \ # \ ax \rangle \ \text{False} \) 
\(\langle \text{intra\text{-}kind} \ (\text{kind} a) \rangle \ \langle \text{intra\text{-}kind} \ (\text{kind} ax) \rangle \) 
\(\langle \text{preds} \ (\text{slice\text{-}kinds} \ S \ (a \ # \ as)) \ s \rangle \) 
\(\langle \text{preds} \ (\text{slice\text{-}kinds} \ S \ as') \ s' \rangle \) 
\(\forall V \in \tau S \ (CFG\text{-}node} \ m) \). \( \text{state\text{-}val} \ s \ V = \text{state\text{-}val} \ s' \ V \) 
\(\text{length} \ s = Suc \ (\text{length} \ cs) \ \langle \text{length} \ s' = Suc \ (\text{length} \ cs) \rangle \) 
have \( \text{False} \ by(\text{fastforce} \ \text{intro}:\ \text{re\text{-}branching\text{-}edges\text{-}slice\text{-}kinds\text{-}False}[\text{of} \ a \ ax]) \) 
thus \( \text{thesis} \ by \text{simp} \) 
qed
qed
qed
next

\begin{case}
(slapa-Call \textit{cs} \textit{a} as \textit{Q} \textit{r} \textit{p} \textit{fs})
\end{case}

\begin{note}
\textit{IH} = \{ \forall \textit{m} \textit{as}' \textit{s} \textit{s}'.
\end{note}

\[ \texttt{[upd-cs (a \# cs) \textit{as} = []; same-level-path-aux (a \# cs) as'}; \]
\quad \forall \textit{c} \in \textit{set (a \# cs)} \quad \text{valid-edge} \textit{c}; \quad \textit{m} \quad \texttt{→* \# (-Low-)}; \quad \textit{m} \quad \texttt{→* \# (-Low-)}; \quad \\
\quad \forall \texttt{i < length (a \# cs)} . \quad \forall \texttt{V \in RV S (CFG-node (sourcecode ((a \# cs) \# i)))}. \\
\quad \texttt{fst (s ! Suc i) V = fst (s' ! Suc i) V}; \\
\quad \forall \texttt{i < Suc (length (a \# cs))}. \quad \texttt{snd (s ! i) = snd (s' ! i)}; \\
\quad \forall \texttt{V \in RV S (CFG-node m)}. \quad \texttt{state-val s V = state-val s' V}; \\
\quad \texttt{preds (slice-kinds S as) s; \quad \texttt{preds (slice-kinds S as'} \textit{s}'.} \\
\quad \texttt{length s = Suc (length (a \# cs)); \quad \texttt{length s' = Suc (length (a \# cs))}} \\
\Rightarrow \forall \texttt{V \in Use (-Low-)}. \quad \texttt{state-val (transfers(slice-kinds S as) s) \quad V =} \\
\texttt{state-val (transfers(slice-kinds S as') s') V}, \]
\end{note}

\begin{note}
\texttt{rus = \forall \textit{i < length cs}. \forall \texttt{V \in RV S (CFG-node (sourcecode (cs \# i)))}.} \\
\quad \texttt{fst (s ! Suc i) V = fst (s' ! Suc i) V}, \]
\end{note}

\begin{from}
\texttt{\forall \textit{c} \in \textit{set cs}. \quad \texttt{valid-edge c \quad \texttt{\textbf{by simp}}}} \quad \texttt{valid-edge a} \\
\quad \texttt{\texttt{\textbf{with}} \quad \texttt{\textbf{same-level-path-aux (a \# cs) as'}}} \quad \texttt{have \textit{sourcecode} \textit{a} \textit{=} \textit{m} \texttt{\quad \texttt{and valid-edge a}} \\
\end{from}

\begin{from}
\texttt{\forall \texttt{\textbf{with}} \quad \texttt{\textbf{same-level-path-aux (a \# cs) as'}}} \quad \texttt{have \textit{sourcecode} \textit{a} \textit{=} \textit{m} \texttt{\quad \texttt{and valid-edge a}} \\
\end{from}

\begin{show}
\texttt{\textbf{proof cases L = {}}} \texttt{\textbf{with UseLow show \textbf{?thesis by simp}}}
\end{show}

\begin{next}
\texttt{case False}
\end{next}

\begin{proof}
\texttt{\textbf{case Nil}} \quad \texttt{\textbf{with}} \quad \texttt{\textbf{\textbf{same-level-path-aux (a \# cs) as'}}} \quad \texttt{have \textit{m} \textit{=} \textit{(-Low-)} \texttt{\textbf{by fastforce}}}
\end{proof}

\begin{proof}
\texttt{\textbf{case Cons ax ax}} \quad \texttt{\textbf{with}} \quad \texttt{\textbf{\textbf{same-level-path-aux (a \# cs) as'}}} \quad \texttt{have \textit{m} \textit{=} \textit{ax} \texttt{\quad \texttt{and valid-edge ax}} \\
\end{proof}
obtain cf cfs where [simp]:s = cf # cfs by (cases s) (auto simp: slice-kinds-def)
from ‹preds (slice-kinds S as') s' (as' = ax # ax)›
obtain cf' cfs' where [simp]:s' = cf' # cfs'
  by (cases s') (auto simp: slice-kinds-def)
have ‹∃ Q r p fs. kind ax = Q:r→pfs›
proof (cases kind ax rule: edge-kind-cases)
case Intra
  have False
  proof (cases sourcenode ax ∈ ⌊HRB-slice S⌋ CFG)
    case True
    with ⟨intra-kind (kind ax)⟩
    have slice-kind S ax = kind ax
      by (rule slice-intra-kind-in-slice)
    have unique:∃! a'. valid-edge a' ∧ sourcenode a' = sourcenode a ∧
      intra-kind (kind a') by (rule call-only-one-intra-edge)
    from ⟨valid-edge a⟩ ⟨kind a = Q:r→pfs⟩ obtain x
      where x ∈ get-return-edges a
    proof (fastforce dest: get-return-edge-call)
      with ⟨valid-edge ax⟩ obtain a' where valid-edge a'
        and sourcenode a' = sourcenode a and kind a' = (λ cf. False)
        by (fastforce dest: call-return-node-edge)
    have False by (simp add: slice-kinds-def)
    thus ?thesis by simp
  next
    case False
    with ⟨kind a = Q:r→pfs⟩ ⟨sourcecenode ax = m⟩ ⟨sourcecenode a = m⟩
    have slice-kind S a = (λ cf. False):r→pfs
      by (fastforce intro: slice-kind-Call)
    with ⟨preds (slice-kinds S (a ≠ as)) s'⟩
    have False by (simp add: slice-kinds-def)
    thus ?thesis by simp
  qed
thus ?thesis by simp
next
  case (Return Q' p' f')
  from ⟨valid-edge ax⟩ ⟨kind ax = Q':p→p'⟩ ⟨valid-edge a⟩ ⟨kind a = Q:r→pfs⟩
  ⟨sourcecenode a = m⟩ ⟨sourcecenode ax = m⟩
  have False by ¬ (drule return-edges-only, auto)
  thus ?thesis by simp
qed simp
have sourcenode a ∈ [HRB-slice S] CFG
proof (rule contr)
  assume sourcenode a ∉ [HRB-slice S] CFG
next
from this \( \langle \text{kind } a = Q; r \mapsto p; fs \rangle \)
have \( \text{slic-kind } S a = (\text{cf. } \text{False}) ; r \mapsto p; fs \)
  by (rule slic-kind-Call)
  with \( \langle \text{preds } (\text{slic-kinds } S (a \# as)) s \rangle \)
show False by (simp add: slic-kinds-def)

qed

with \( \langle \text{preds } (\text{slic-kinds } S (a \# as)) s \rangle \langle \text{kind } a = Q; r \mapsto p; fs \rangle \)
have \( \text{pred } (\text{kind } a) s \)
  by (fastforce dest: slic-kind-Call-in-slice simp: slic-kinds-def)

from \( \langle \text{source-node } a \in [\text{HRB-slice } S]_{\text{CFG}} \rangle \)
  \( \langle \text{source-node } a = m \rangle \langle \text{source-node } ax = m \rangle \)
have \( \langle \text{source-node } ax \in [\text{HRB-slice } S]_{\text{CFG}} \rangle \) by simp
with \( \langle \text{as'} = ax \# asx \rangle \langle \text{preds } (\text{slic-kinds } S as') s' \rangle \)
\( \langle \exists Q \ r \ p \ fs. \ \text{kind } ax = Q; r \mapsto p; fs \rangle \)

have \( \langle \text{pred } (\text{kind } ax) s' \rangle \)
  by (fastforce dest: slic-kind-Call-in-slice simp: slic-kinds-def)

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

  from \( \langle \text{valid-edge } a \rangle \langle \text{source-node } a \rangle \langle \text{source-node } a \rangle \)
    by (fastforce intro!: empty-path simp: intra-path-def)
  with \( \langle \text{source-node } a \in [\text{HRB-slice } S]_{\text{CFG}} \rangle \)
  \( \langle \text{valid-edge } a \rangle \langle V \in \text{Use } (\text{source-node } a) \rangle \)
  have \( V \in rv S \) (CFG-node (source-node a))
    by (auto intro!: rv CFG-Use-SDG-Use simp: SDG-to-CFG-set-def source-nodes-def)
\}

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

have Use: \( \forall V \in \text{Use } (\text{source-node } a) \). state-val s V = state-val s' V by simp
from \( \forall i < \text{Suc } (\text{length } cs) \). snd (s ! i) = snd (s' ! i) \)

have \( \text{snd } (hd s) = \text{snd } (hd s') \) by fastforce

with \( \langle \text{valid-edge } a \rangle \langle \text{kind } a = Q; r \mapsto p; fs \rangle \langle \text{valid-edge } ax \rangle \)
\( \langle \exists Q \ r \ p \ fs. \ \text{kind } ax = Q; r \mapsto p; fs \rangle \langle \text{source-node } a = m \rangle \langle \text{source-node } ax = m \rangle \)
\( \langle \text{pred } (\text{kind } a) s \rangle \langle \text{pred } (\text{kind } ax) s' \rangle \)
  Use \( \langle \text{length } s = \text{Suc } (\text{length } cs) \rangle \)
  \( \langle \text{length } s' = \text{Suc } (\text{length } cs) \rangle \)

have \( \langle \text{simp} : az = a \rangle \) by (fastforce intro!: CFG-equal-Use-equal-call)
from \( \langle \text{same-level-path-aux } cs as' = ax \# asx \rangle \langle \text{kind } a = Q; r \mapsto p; fs \rangle \)
\( \langle \exists Q \ r \ p \ fs. \ \text{kind } ax = Q; r \mapsto p; fs \rangle \)

have \( \langle \text{same-level-path-aux } (a \# cs) asx \rangle \) by simp

from \( \langle \text{target-node } ax \rangle \langle \text{asx} \mapsto* (-\text{Low}) \rangle \) have \( \langle \text{target-node } a \rangle \langle \text{asx} \mapsto* (-\text{Low}) \rangle \)

by simp

from \( \langle \text{kind } a = Q; r \mapsto p; fs \rangle \langle \text{upd-cs } cs (a \# as) = [] \rangle \)

have upd-cs (a \# cs) as = [] by simp

from \( \langle \text{source-node } a \in [\text{HRB-slice } S]_{\text{CFG}} \rangle \langle \text{kind } a = Q; r \mapsto p; fs \rangle \)
have slice-kind: slic-kind S a = \( Q; r \mapsto p (\text{target-node } a) \) (HRB-slice S) fs
  by (rule slice-kind-Call-in-slice)

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

have \( \text{snds } ! i < \text{Suc } (\text{length } (a \# cs)) \).
  \( \text{snd } \) (transfer (slice-kind S a) s ! i) = \( \text{snd } \) (transfer (slice-kind S a) s' ! i) 


by auto(case-tac i,auto)
from ⟨valid-edge a⟩ ⟨kind a = Q∶r→pfs⟩ obtain ins outs
  where (p,ins,outs) ∈ set procs by(fastforce dest!: callee-in-procs)
with ⟨valid-edge a⟩ ⟨kind a = Q∶r→pfs⟩
have length (ParamUses (sourcenode a)) = length ins
  by(fastforce intro:ParamUses-call-source-length)
with ⟨valid-edge a⟩
  have ∀ i < length ins. ∀ V ∈ (ParamUses (sourcenode a))!i. V ∈ Use
tARGET
        (sourcenode a)
        by(fastforce intro:ParamUses-in-Use)
with  ∀ V ∈ Use (sourcenode a). state-val s V = state-val s' V
have ∀ i < length ins. ∀ V ∈ (ParamUses (sourcenode a))!i.
    state-val s V = state-val s' V by fastforce
with ⟨valid-edge a⟩ ⟨kind a = Q∶r→pfs⟩ ⟨(p,ins,outs) ∈ set procs⟩
  ⟨pred (kind a) s⟩ ⟨pred (kind ax) s'⟩
have ∀ i < length ins. (params fs (fst (hd s)))!i = (params fs (fst (hd s')))!i
  by(fastforce intro!∶CFG-call-edge-params)
from ⟨valid-edge a⟩ ⟨kind a = Q∶r→pfs⟩ ⟨(p,ins,outs) ∈ set procs⟩
have length fs = length ins by(rule CFG-call-edge-length)
{ fix i assume i < length fs
  with ⟨length fs = length ins⟩ have i < length ins by simp
  from ⟨i < length fs⟩ have (params fs (fst cf))!i = (fs!i) (fst cf)
    by(rule params-nth)
  moreover
  from ⟨i < length fs⟩ have (params fs (fst cf'))!i = (fs!i) (fst cf')
    by(rule params-nth)
  ultimately have (fs!i) (fst (hd s)) = (fst (hd s'))
    using ⟨i < length ins⟩
      ⟨∀ i < length ins. (params fs (fst (hd s)))!i = (params fs (fst (hd s')))!i⟩
    by simp 
}{ hence ∀ i < length fs. (fs ! i) (fst cf) = (fs ! i) (fst cf') by simp
  { fix i assume i < length fs
    with ⟨∀ i < length fs. (fs ! i) (fst cf) = (fs ! i) (fst cf')⟩
    have (fs ! i) (fst cf) = (fs ! i) (fst cf') by simp
    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
        with ⟨i < length fs⟩
        have (csppa (targetnode a) (HRB-slice S) 0 fs)!i = fs!i
          by(rule csppa-Formal-in-in-slice)
        with ⟨(fs ! i) (fst cf) = (fs ! i) (fst cf')⟩ show ?thesis by simp
      next
      case False
        with ⟨i < length fs⟩
        have (csppa (targetnode a) (HRB-slice S) 0 fs)!i = Map.empty
          by(rule csppa-Formal-in-notin-slice)
      thus ?thesis by simp
\[\text{qed }\]

**hence** \(\forall i < \text{length } fs\).

\[((\text{cspp (targetnode } a) (\text{HRB-slice } S) \text{fs})!i)(\text{fst } cf) =\]

\[((\text{cspp (targetnode } a) (\text{HRB-slice } S) \text{fs})!i)(\text{fst } cf')\]

by(simp add:cspp-def)

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

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

\( (\text{fst } cf)!i =\)

\((\text{cspp (targetnode } a) (\text{HRB-slice } S) \text{fs})!i)(\text{fst } cf')\)

by(fastforce intro:params-nth)

**moreover**

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

\( (\text{fst } cf)!i =\)

\((\text{cspp (targetnode } a) (\text{HRB-slice } S) \text{fs})(\text{fst } cf')!i\)

by(fastforce intro:params-nth)

**ultimately**

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

\( (\text{fst } cf)!i =\)

(param (cspp (targetnode } a) (HRB-slice } S) \text{fs})(\text{fst } cf')!i

using \(eq \cdot i < \text{length } fs\) by simp }

**hence** (params (cspp (targetnode } a) (HRB-slice } S) \text{fs})(\text{fst } cf) =

params (cspp (targetnode } a) (HRB-slice } S) \text{fs})(\text{fst } cf')

by(simp add:list-eq-iff-nth-eq)

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

**obtain** cfx where \([simp];\]

transfer (slice-kind } S a) (cf#cfs) = cfx#cf#cfs

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

by auto

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

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

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

**from** res \(\forall V \in rv S (\text{CFG-node } m), \text{state-val } s V = \text{state-val } s' V,\)

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

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

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

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

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

by auto(case-tac i,auto)

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

**have** preds (slice-kinds } S as)

(transfer (slice-kind } S a) \text{ s} by(simp add:slice-kinds-def)

**moreover**

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

**have** preds (slice-kinds } S asx)

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

**moreover**

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

**have** length (transfer (slice-kind } S a) \text{ s} =
Succ \( \text{(length \ (a \ # \ cs))} \) \text{ by simp}

moreover
from \( \langle \text{length} \ s' = \text{Succ} \ (\text{length} \ cs) \rangle \)
have \( \text{length} \ (\text{transfer} \ (\text{slice-kind} \ S \ a) \ s') = \text{Succ} \ (\text{length} \ (a \ # \ cs)) \) \text{ by simp}

moreover
from IH[OF \( \langle \text{upd-cs} \ (a \ # \ cs) \ as = [] \rangle \); \( \langle \text{same-level-aux} \ (a \ # \ cs) \ asx \rangle \)
\( \forall c \in \text{set} \ (a \ # \ cs) \); \( \text{valid-edge} \ c \); \( \langle \text{targetnode} \ a \ \rightarrow^{\ast} \ \text{(Low-\ (\cdot))} \rangle \)
\( \langle \text{targetnode} \ a \ \rightarrow^{\ast} \ \text{(Low-\ (\cdot))} \rangle \)
\( \text{res'} \ \text{snds} \ \text{rv} \ \text{calculation} \) \( \langle \text{as'} = ax\#axs \rangle \)
show \( \forall \text{thesis} \ by(\text{simp add: slice-kinds-def}) \)

qed

qed

next

case \( \langle \text{slpa-Return} \ cs \ a \ as \ Q \ p \ f \ c' \ cs' \rangle \)

note IH = \( \langle \\forall m \ as' \ s \ s'. \ [\text{upd-cs} \ cs' \ as = [] \]; \langle \text{same-level-aux} \ cs' \ as' \rangle \); \( \forall c \in \text{set} \ cs' \); \( \text{valid-edge} \ c \); \( m \ \rightarrow^{\ast} \ \text{(Low-\ (\cdot))} \); \( m \ \rightarrow^{\ast} \ \text{(Low-\ (\cdot))} \)

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

\( \text{fst} \ (s ! \text{Suc} \ i) \ V = \text{fst} \ (s' ! \text{Suc} \ i) \ V \)

\( \forall i<\text{Suc} \ (\text{length} \ cs') \); \( \text{snd} \ (s ! i) = \text{snd} \ (s' ! i) \)

\( \forall V \in \text{rv} \ S \ (\text{CFG-node} \ m) \); \( \text{state-val} \ s \ V = \text{state-val} \ s' \ V \)

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

\( \text{length} \ s = \text{Succ} \ (\text{length} \ cs') \); \( \text{length} \ s' = \text{Succ} \ (\text{length} \ cs') \)

\( \forall V \in \text{Use} \ \text{(Low-\ (\cdot))} \); \( \text{state-val} \ (\text{transfers} \ (\text{slice-kinds} \ S \ as) \ s) \ V = \text{state-val} \ (\text{transfers} \ (\text{slice-kinds} \ S \ as') \ s') \ V \)

note res = \( \langle \\forall i<\text{length} \ cs \); \( \forall V \in \text{rv} \ S \ (\text{CFG-node} \ (\text{source-node} \ (cs' ! i))) \); \( \text{fst} \ (s ! \text{Suc} \ i) \ V = \text{fst} \ (s' ! \text{Suc} \ i) \ V \)

from \( \langle m \ \rightarrow^{\ast} \ \text{(Low-\ (\cdot))} \rangle \)
have \( \text{source-node} \ a = m \) \text{ and } \( \text{valid-edge} \ a \)

and \( \text{targetnode} \ a \ \rightarrow^{\ast} \ \text{(Low-\ (\cdot))} \) \text{ by (auto elim: path-split-cons)}

from \( \forall c \in \text{set} \ cs \); \( \text{valid-edge} \ c \); \( \langle cs = cs' \rangle \)
have \( \text{valid-edge} \ c' \) \text{ and } \( \forall c \in \text{set} \ cs' \); \( \text{valid-edge} \ c' \) \text{ by simp-all}

show \?case

proof(cases \( \text{L = \{\}} \))
case True with \text{UseLow}

show \?thesis \text{ by simp}

next

case False

show \?thesis

proof(cases \( \text{as'} \))
case Nil

with \( \langle m \ \rightarrow^{\ast} \ \text{(Low-\ (\cdot))} \rangle \) \text{ have } \( m = (\text{Low-\ (\cdot)}) \) \text{ by fastforce}

with \( \langle \text{valid-edge} \ a \rangle \); \( \langle \text{source-node} \ a = m \rangle \) \text{ have } \text{targetnode} \ a = (\text{Exit-\ (\cdot)})

by \( \text{(rule Exit-successor-of-Low,simp+)} \)

from \( \text{Low-source-Exit-edge} \) \text{ obtain } \text{a'} \text{ where } \text{valid-edge} \ a'

and \( \text{source-node} \ a' = (\text{Low-\ (\cdot)} \) \text{ and } \text{targetnode} \ a' = (\text{Exit-\ (\cdot)})

and \( \text{kind} \ a' = (\text{\lambda \ True}) \) \text{ by blast}

from \( \langle \text{valid-edge} \ a \rangle \); \( \langle \text{source-node} \ a = m \rangle \); \( \langle m = (\text{Low-\ (\cdot)} \rangle \)

\( \text{targetnode} \ a = (\text{Exit-\ (\cdot)}) \); \( \langle \text{valid-edge} \ a' \rangle \); \( \langle \text{source-node} \ a' = (\text{Low-\ (\cdot)} \rangle \)

\( \text{targetnode} \ a' = (\text{Exit-\ (\cdot)}) \)

have \( a = a' \) \text{ by (fastforce dest: edge-det)}

with \( \langle \text{kind} \ a' = (\text{\lambda \ True}) \rangle \) \text{ have } \text{kind} \ a = (\text{\lambda \ True}) \text{ by simp}
with \langle\text{targetnode } a = (\text{Exit})\rangle \oplus \langle\text{targetnode } a → as \rightarrow \ast (\text{Low})\rangle
have (\text{Low}) = (\text{Exit}) \Longleftarrow \langle\text{drule path-Exit-source,auto}\rangle
with False have False by \langle\text{drule Low-neq-Exit,simp}\rangle
thus \text{thesis} by simp

next
case \langle\text{Cons } ax \ asx\rangle
with \langle m → as' \rightarrow \ast \text{ (Low)}\rangle have sourcenode ax = m \text{ and valid-edge ax}
and targetnode ax → asx \rightarrow \ast \langle\text{Low}\rangle by (\text{auto elim:path-split-Cons})
from \langle\text{valid-edge } a\rangle \langle\text{valid-edge ax}\rangle \langle\text{kind } a = Q_{p}f\rangle
⟨\text{sourcenode } a = m\rangle \langle\text{sourcenode } ax = m\rangle
have \exists Q \ f. \text{ kind } ax = Q_{p}f \ by (\text{auto dest: return-edges-only})
with \langle\text{same-level-path-aux } cs \ as'\rangle \langle as' = ax\#ax\rangle \langle cs = c' \# cs'\rangle
have ax ∈ get-return-edges c' \text{ and same-level-path-aux } cs' \ asx by auto
from \langle\text{valid-edge } c'\rangle \langle ax ∈ get-return-edges c'\rangle \langle a ∈ get-return-edges c'\rangle
have [simp]; ax = a by (\text{rule get-return-edges-unique})
from \langle\text{targetnode } ax → asx \rightarrow \ast \text{ (Low)}\rangle have targetnode a → asx \rightarrow \ast \langle\text{Low}\rangle
by simp
from \langle\text{upd-cs } (a \# as) = []\rangle \langle\text{kind } a = Q_{p}f\rangle \langle cs = c' \# cs'\rangle
\langle a ∈ get-return-edges c'\rangle
have upd-cs cs' as = [] by simp
from \langle\text{length } s = \text{Suc} (\text{length } cs)\rangle \langle cs = c' \# cs'\rangle
obtain cf cfx cfs where s = cf#cfx#cfs
by (cases s, auto, case-lac list, fastforce+)
from \langle\text{length } s' = \text{Suc} (\text{length } cs)\rangle \langle cs = c' \# cs'\rangle
obtain cf’ cfx’ cfs’ where s’ = cf’#cfx’#cfs’
by (cases s’, auto, case-lac list, fastforce+)
from res (cs = c’ # cs’); (s = cf#cfx#cfs); (s’ = cf’#cfx’#cfs’)
have res1:∀ i<\text{length } cs.
\forall V∈rv S (\text{CFG-node (sourcenode (cs’ ! i)))}
\text{fst (cf#cfs) ! Suc } i \ V = \text{fst ((cfx’#cfs’) ! Suc } i \ V)
\text{and } \forall V∈rv S (\text{CFG-node (sourcenode c’}))
(fst cfx) \ V = (\text{fst cfx}) \ V
by auto
from \langle\text{valid-edge } c'\rangle \langle a ∈ get-return-edges c'\rangle
obtain Qr px pxs where kind c' = Q_{p}f_{px → pafpxs}
by (fastforce dest!:only-call-get-return-edges)
have \forall V ∈ rv S (\text{CFG-node (targetnode a)}).
V ∈ rv S (\text{CFG-node (sourcenode c’)})

proof
fix V assume V ∈ rv S (\text{CFG-node (targetnode a)})
from \langle\text{valid-edge } c'\rangle \langle a ∈ get-return-edges c'\rangle
obtain a’ where edge:valid-edge a' source-node a' = source-node c'
targetnode a' = targetnode a intra-kind (kind a')
by -(\text{drule call-return-node-edge,auto simp:intra-kind-def})
from \langle V ∈ rv S (\text{CFG-node (targetnode a)})\rangle
obtain as n’ where targetnode a → as → as* parent-node n’
and n’ ∈ HRB-slice S and V ∈ UseSDG n'
and all:∀ n", valid-SDG-node n" ∧ parent-node n" ∈ set (source-nodes as)
→ V \notin DefSDG n" by (fastforce elim:reE)
proof
tshow thesis
qed
have \(\forall\) with rvs1
have sx with \(\langle\)
have \(\langle\)
ultimately have \([\]
have \(\langle\)
moreover
have \(\langle\)
moreover
have \(\langle\)
from \(\langle\)
have \(\langle\)
by \((\)
qed
show thesis
proof \((\) cases \(\langle\)
case True
from \(\langle\)
have \(\langle\)
by \(\)
moreover
from \(\langle\)
have \(\langle\)
ultimately have \(\)
from \(\langle\)
obtain ins outs where \(\) \(\) \(\)
by \((\)
with \(\) \(\)
have slice-kind: slice-kind \(\)
have \(\)
and \(\)
by simp-all
with rvs1 have rvs1 \(\) \(\) \(\)
\(\forall\) \(\)
\(\)
\(\)
\begin{align*}
\text{fst } ((\text{transfer } (\text{slice-kind } S \ a) \ s) \ ! \ Suc \ i) \ V &= \\
\text{fst } ((\text{transfer } (\text{slice-kind } S \ a) \ s') \ ! \ Suc \ i) \ V
\end{align*}

by fastforce

from slice-kind \ \forall \ i < Suc (\text{length } cs). \ \text{snd } (s ! i) = \text{snd } (s' ! i) \ \langle cs = c' \#
\text{cs'} \rangle

\begin{align*}
\langle s = cf \# cf' \# cf' \# cfs \rangle \ \langle s' = cf' \# cf' \# cfs' \rangle \\
\text{have snds: } \forall \ i < Suc (\text{length } cs'). \\
\text{snd } (\text{transfer } (\text{slice-kind } S \ a) \ s ! i) &= \\
\text{snd } (\text{transfer } (\text{slice-kind } S \ a) \ s' ! i)
\end{align*}

apply auto apply\(\text{case-tac } i\) apply auto

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

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

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

proof

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

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

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

case True

then obtain \(i\) where \(i < \text{length } \text{(ParamDefs } (\text{targetnode } a))\)

and \(\text{(ParamDefs } (\text{targetnode } a))!i = V\)

by(fastforce simp\(\text{ in }\text{-set-conv-nth}\))

from \(\text{valid-edge } a \ \langle \text{kind } a = Q \langle p,f \rangle \ \langle p,\text{ins},\text{outs}\rangle \in \text{set procs}\)\)

have \(\text{length(ParamDefs } (\text{targetnode } a)) = \text{length } \text{outs}\)

by(fastforce intro:ParamDefs-return-target-length)

show \(?\text{thesis}\)

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

case True

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

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

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

from \(\text{valid-edge } a \langle \text{kind } a = Q \langle p,f \rangle \ \langle p,\text{ins},\text{outs}\rangle \in \text{set procs}\rangle\)

have \(\forall \ V \in \text{set } \text{outs}, \ V \in \text{Use } \text{(source} \text{node } a)\) by(fastforce dest:outs-in-Use)

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

proof

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

from \(\text{valid-edge } a \langle \text{source} \text{node } a = m \rangle\)
have parent-node (CFG-node m) −[−,∗ parent-node (CFG-node m)
by (fastforce intro:empty-path simp:intro-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 refl,
  auto intro!:CFG-Use-SDG-Use simp:SDG-to_CFG-set-def
sourcenes-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 cfx) (fst cf)) V =
(fst cfx)(⟨ParamDefs (targetnode a))!i⟩
(rspp (targetnode a)
⟨HRB-slice S⟩ outs (fst cfx′) (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 cfx) V = (fst cfx′) V⟩
⟨∀ V ∈ rv S (CFG-node (sourcenode a))⟩,
⟨∀ V ∈ rv S (CFG-node (targetnode a))⟩,
⟨∀ V ∈ rv S (CFG-node (sourcenode c′))⟩,
⟨(ParamDefs (targetnode a))!i = V;[THEN sym]
have (fst cfx) (ParamDefs (targetnode a) ! i) =
(fst cfx′) (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 cfx) V = (fst cfx′) 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
qed

with \texttt{sz sz'}

have \(rv' \forall V \in rv S\) (CFG-node (targetnode \(a\))).

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

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

by fastforce

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

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

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

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

moreover

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

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

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

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

moreover

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

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

by(\(\text{simp simp add: s = cf#cfx#cfs}\))

moreover

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

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

by(\(\text{simp simp add: s' = cf'#cfx'#cfs'}\))

moreover

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

\(\langle \forall \text{cset cs'} \ \text{valid-edge c} \ \langle \text{targetnode a \ as}\rightarrow* \ (-\text{Low}-)\rangle\)

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

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

next

\text{case False}

from \text{this} \langle \text{kind a} = Q\#p f\rangle

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

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

with \(\langle s = cf#cfx#cfs\rangle \langle s' = cf'#cfx'#cfs'\rangle\)

have \(\text{simp: transfer} \ (\text{slice-kind S a}) \ s = cfx#cfs\)

\(\text{transfer} \ (\text{slice-kind S a}) \ s' = cfx'#cfs'\) by simp-all

from \text{slice-kind} \(\forall i<\text{Suc} \ (\text{length cs})\).

\(\text{snd} \ (s \ i) = \text{snd} \ (s' \ i)\).

\(\langle cs = c' \# cs' \rangle \langle s = cf#cfx#cfs \rangle \langle s' = cf'#cfx'#cfs'\rangle\)

have \(\text{snds: } \forall i<\text{Suc} \ (\text{length cs}')\).

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

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

\(\forall V \in rv S\) (CFG-node (sourcenode (cs' \ i))).

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

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

by fastforce

from \(\forall V \in rv S\) (CFG-node (targetnode a))\)

\(V \in rv S\) (CFG-node (sourcenode c'))\)

\(\forall V \in rv S\) (CFG-node (sourcenode c')).
proof

lemma rv-Low-Use-Low:
  assumes m \rightarrow ω (-Low) and m \rightarrow ω* (-Low) and get-proc m = Main
  and \forall V \in rv S (CFG-node m), cf V = cf' V
  and preds (slice-kinds S as) [(cf,undefined)]
  and preds (slice-kinds S as') [(cf',undefined)]
  and CFG-node (-Low) \in S
  shows \forall V \in Use (-Low),
  state-val (transfers(slice-kinds S as) [(cf,undefined)]) V =
  state-val (transfers(slice-kinds S as') [(cf',undefined)]) V

proof(cases as)
  case Nil
  with \langle m \rightarrow ω (-Low) \rangle have valid-node m and m = (-Low)
  by(auto intro:path-valid-node simp:ev-def)
  { fix V assume V \in Use (-Low)
    moreover
    from valid-node m \langle m = (-Low) \rangle have (-Low) \rightarrow ω* (-Low)
    by(justforce intro:empty-path simp:intra-path-def)
  qed
moreover
from valid-node m \( m = \langle \text{-Low}\rangle \) have \( \langle \text{CFG-node} \ \text{-Low}\rangle \) in HRB-slice S 
by (fastforce intro:HRB-slice-refl)
ultimately have \( V \in rv S \) (CFG-node m) using \( m = \langle \text{-Low}\rangle \) 
by (auto intro:rel CFG-Use-SDG-Use simp:sourcenodes-def) 

hence \( \forall \ V \in Usec \ \text{-Low} \). \( V \in rv S \) (CFG-node m) by simp 

show ?thesis 
proof (cases \( L = \{ \} \) )
case True with UseLow show ?thesis by simp 
next 
case False 
from \( \langle m - as' \rightarrow \rightarrow \rightarrow \text{-Low} \rangle \) have \( m - as' \rightarrow \rightarrow \rightarrow \text{-Low} \) by (simp add:vp-def) 
from \( \langle m - as' \rightarrow \rightarrow \rightarrow \text{-Low} \rangle \) \( m = \langle \text{-Low}\rangle \) have \( as' = [] \) 

proof (induct \( m \) as' \( m' \equiv \langle \text{-Low} \rangle \) rule:path.induct) 
case (Cons-path \( m'' \) as \( m \) ) 
from \( \langle \text{valid-edge} \ \text{a} \rangle \) (sourcenode \( a = m \) ) \( m = \langle \text{-Low}\rangle \) 

have targetnode \( a = \langle \text{-Exit}\rangle \) by (rule Exit-successor-of-Low,simp+)
with \( \langle \text{targetnode} \ \text{a} = m'' \rangle \) \( m'' = \rightarrow \rightarrow \rightarrow \text{-Low} \)

have \( \langle \text{-Exit}\rangle = \langle \text{-Exit}\rangle \) by (drule path-Exit-source,auto) 
with False have False by (drule Low-neq-Exit,simp) 
thus ?case by simp 

qed simp 
with \( \forall \ V \in rv S \) (CFG-node m). cf \( V = cf' \ V \) 
\( \forall \ V \in Usec \ \text{-Low} \). \( V \in rv S \) (CFG-node m) 

show ?thesis by (fastforce simp:slice-kinds-def) 

qed 
next 
case \( \langle \text{Cons} \ \text{ax} \ \text{asx} \rangle \) 
with \( \langle m - ax' \rightarrow \rightarrow \rightarrow \text{-Low} \rangle \) have sourcenode \( ax = m \) and valid-edge \( ax \) 

and targetnode \( ax = \rightarrow \rightarrow \rightarrow \text{-Low} \) 

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 \( \text{as}' \) ) 
case Nil 
with \( \langle m - as' \rightarrow \rightarrow \rightarrow \text{-Low} \rangle \) have \( m = \langle \text{-Low}\rangle \) by (fastforce simp:vp-def) 
with \( \langle \text{valid-edge} \ \text{ax} \rangle \) (sourcenode \( ax = m \) ) have targetnode \( ax = \langle \text{-Exit}\rangle \) 

by (rule Exit-successor-of-Low,simp+) 
from Low-source-Exit-edge obtain \( \text{a}' \) where valid-edge \( \text{a}' \) 

and sourcenode \( \text{a}' = \langle \text{-Low}\rangle \) and targetnode \( \text{a}' = \langle \text{-Exit}\rangle \) 

and kind \( \text{a}' = (\text{ax} \ \text{asx}) \) by blast 
from \( \langle \text{valid-edge} \ \text{ax} \rangle \) (sourcenode \( ax = m \) ) \( m = \langle \text{-Low}\rangle \) 

(targetnode \( ax = \langle \text{-Exit}\rangle \) ) valid-edge \( \text{a}' \) (sourcenode \( \text{a}' = \langle \text{-Low}\rangle \) ) 

(targetnode \( \text{a}' = \langle \text{-Exit}\rangle \) )
have \( ax = a' \) by (fastforce dest:edge-det)

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

with \( \langle \text{targetnode } ax = (\text{-Exit}) \rangle \) \( \langle \text{targetnode } \text{asx} \rightarrow \text{-Low} \rangle \)

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

with \( \text{False have False} \) by \( -(\text{drule Low-neq-Exit},\text{simp})\)

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

next

\begin{align*}
\text{case } & (\text{Cons } ax' \text{ asx}') \\
\text{from } & \langle m \rightarrow as \rightarrow , \rightarrow \rangle \langle \text{-Low} \rangle \text{ have } \text{valid-path-aux} \left[ \right] \text{ as and } m \leftarrow as \rightarrow \rightarrow \langle \text{-Low} \rangle \\
& \ (\text{by (simp-all add:vp-def valid-path-def)}) \\
\text{from this } & \langle as = ax'\# asx' \rangle \langle \text{get-proc } m = \text{Main} \rangle \\
\text{have } & \text{same-level-path-aux} \left[ \right] \text{ as and upd-cs} \left[ \right] \text{ as } \ = \ \left[ \right] \\
& \ (\text{by -(rule vpa-Main-slpa[of - - m (\text{-Low})], (fastforce intro!:get-proc-Low simp:valid-call-list-def)}) + ) \\
\text{hence } & \text{same-level-path-aux} \left[ \right] \text{ as and upd-cs} \left[ \right] \text{ as } \ = \ \left[ \right] \text{ by simp-all} \\
\text{from } & \langle m \rightarrow as' \rightarrow , \rightarrow \rangle \langle \text{-Low} \rangle \text{ have } \text{valid-path-aux} \left[ \right] \text{ as'} and m \rightarrow as' \rightarrow \rightarrow \langle \text{-Low} \rangle \\
& \ (\text{by (simp-all add:vp-def valid-path-def)}) \\
\text{from this } & \langle as' = ax'\# asx' \rangle \langle \text{get-proc } m = \text{Main} \rangle \\
\text{have } & \text{same-level-path-aux} \left[ \right] \text{ as'} and upd-cs \left[ \right] \text{ as'} = \left[ \right] \\
& \ (\text{by -(rule vpa-Main-slpa[of - - m (\text{-Low})], (fastforce intro!:get-proc-Low simp:valid-call-list-def)}) + ) \\
\text{hence } & \text{same-level-path-aux} \left[ \right] \text{ as'} \text{ by simp} \\
\text{from } & \langle \text{same-level-path-aux} \left[ \right] \text{ as} \rangle \langle \text{upd-cs} \left[ \right] \text{ as } \ = \ \left[ \right] \\
& \langle \text{same-level-path-aux} \left[ \right] \text{ as'} \rangle \langle m \rightarrow as \rightarrow \rightarrow \langle \text{-Low} \rangle \rangle \langle m \rightarrow as' \rightarrow \rightarrow \langle \text{-Low} \rangle \rangle \\
& \forall V \in \text{rv} S \langle \text{CFG-node } m \rangle, \text{ cf } V = \text{ cf' } V \langle \text{CFG-node } \langle \text{-Low} \rangle \rangle \in S, \\
& \langle \text{preds } \langle \text{slice-kinds } S \text{ as} \rangle \langle [\text{cf,undefined}] \rangle \rangle \\
& \langle \text{preds } \langle \text{slice-kinds } S \text{ as'} \rangle \langle [\text{cf',undefined}] \rangle \rangle \\
\text{show } & \text{thesis by -(erule slpa-rv-Low-Use-Low,auto)}
\end{align*}

\begin{align*}
\text{qed} & \\
\text{qed} & \\
\text{qed} &
\end{align*}

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

\begin{align*}
\text{assumes } & \langle \text{cf} \rangle \approx_L \langle \text{cf'} \rangle \text{ and } \langle \text{-High} \rangle \notin \langle \text{HRB-slice } S \rangle_{\text{CFG}} \\
\text{and } & \langle \text{CFG-node } \langle \text{-Low} \rangle \rangle \in S \\
& \langle \text{-Entry-} \rangle \rightarrow as \rightarrow \rightarrow \langle \text{-Low} \rangle \text{ and } \text{preds } \langle \text{ kinds as } \rangle \langle [\text{cf,undefined}] \rangle \\
& \langle \text{CFG-node } \langle \text{-Low} \rangle \rangle \in S \\
\text{shows } & \map \text{ fst } \langle \text{transfers } \langle \text{ kinds as } \rangle \langle [\text{cf,undefined}] \rangle \rangle \approx_L \\
& \map \text{ fst } \langle \text{transfers } \langle \text{ kinds as' } \rangle \langle [\text{cf',undefined}] \rangle \rangle \\
\text{proof} - \\
\text{from } & \langle \text{-Entry-} \rangle \rightarrow as \rightarrow \rightarrow \langle \text{-Low} \rangle \rangle \langle \text{preds } \langle \text{ kinds as } \rangle \langle [\text{cf,undefined}] \rangle \rangle \\
& \langle \text{CFG-node } \langle \text{-Low} \rangle \rangle \in S \\
\text{obtain } & \text{asx where } \text{preds } \langle \text{slice-kinds } S \text{ asx} \rangle \langle [\text{cf,undefined}] \rangle \\
& \text{and } \forall V \in \text{ Use } \langle \text{-Low} \rangle . \\
& \text{state-val } \langle \text{transfers } \langle \text{slice-kinds } S \text{ asx} \rangle \langle [\text{cf,undefined}] \rangle \rangle V = \\
& \text{state-val } \langle \text{transfers } \langle \text{ kinds as } \rangle \langle [\text{cf,undefined}] \rangle \rangle V \\
& \text{and } \text{slice-edges } S \left[ \right] \text{ as } = \text{ slice-edges } S \left[ \right] \text{ asx}
\end{align*}

\[25\]
\begin{verbatim}
and transfers (kinds as) [(cf, undefined)] ≠ []
and (-Entry) → as→j* (-Low-
by(erule fundamental-property-of-static-slicing)
from ⟨(-Entry) → as→j* (-Low-), \preds (kinds as') [(cf', undefined)]⟩
⟨CFG-node (-Low-), S⟩ ∈ S
obtain as' where \preds (slice-kinds S as') [(cf', undefined)]
and \(∀ V ∈ Use (-Low-).
state-val (transfers (slice-kinds S as') [(cf', undefined))] V =
state-val (transfers (kinds as') [(cf, undefined)]) V
and slice-edges S [] as' =
slice-edges S [] as'
and transfers (kinds as') [(cf', undefined)] ≠ []
and (-Entry) → as'→j* (-Low-)
by(erule fundamental-property-of-static-slicing)
from ⟨cf⟩ \(≈ L \langle cf' \rangle \cdot \langle \text{High-} \rangle \notin \text{HRB-slice S} \rangle CFG\)
have \(∀ V ∈ \text{rv S} (CFG-node (-Entry-)). cf V = cf' V
by(fastforce dest:lowEquivalence-relevant-nodes-Entry)
with ⟨(-Entry) → as→j* (-Low-), (-Entry) → as'→j* (-Low-),
⟨CFG-node (-Low-), S⟩ ∈ 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 asx) [(cf, undefined)]) V,
\(∀ V ∈ Use (-Low-).
state-val (transfers (slice-kinds S asx') [(cf', undefined)]) V =
state-val (transfers (kinds asx') [(cf', undefined)]) V,
\transfers (kinds as) [(cf, undefined)] ≠ []
\transfers (kinds as') [(cf', undefined)] ≠ []
show \(?\text{thesis by}(fastforce simp:lowEquivalence-def UseLow neq-Nil-conv)
qed

\end{verbatim}
\[\text{by (fastforce elim: path-split-Cons simp: vp-def)} \]

from \((-\text{Entry}) - as \to *) (-\text{Exit})\) have valid-path as by (simp add: vp-def)

from \(\text{valid-edge } x\) have valid-node (targetnode \(x\)) by simp

hence inner-node (targetnode \(x\))

proof (cases rule: valid-node-cases)
  case Entry
  with \(\text{valid-edge } x\) have False by (rule Entry-target)
  thus \(\text{thesis}\) by simp

next

  case Exit
  with \(\text{targetnode } x \to xs\) (-\text{-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 \(z\) = \((\lambda s. \text{False}), \_\) by blast

  from valid-edge \(x\) \(\text{valid-edge } z\) \((-\text{-Entry}) = \text{sourcenode } x\)
  \(\text{source-node } z\) = (-Entry-) Exit \(\text{target-node } z\) = (-Exit-)
  have \(x = z\) by (fastforce intro: edge-det)

  with \(\text{preds (kinds as)}\) \(((\text{cf. undefined}))\) \(\text{as} = x\#xs\) \(\text{xs} = []\)
  \(\langle\text{kind } z = (\lambda s. \text{False}), \_\rangle\)
  have False by (simp add: kinds-def)
  thus \(\text{thesis}\) by simp

qed simp

with \(\text{targetnode } x \to xs\) (-\text{-Exit}) obtain \(x'\) \(xs'\) where \(xs = xs'@[x']\)
  and targetnode \(x \to xs'\) (-\text{-Low}) and kind \(x'\) = \((\lambda s. \text{True}), \_\)
  by (fastforce elim: exit-path-Low-path)

with \((-\text{-Entry}) = \text{sourcenode } x\) \(\text{valid-edge } x\)
have \((-\text{-Entry}) - x\#xs'\to +\) (-\text{-Low}) by (fastforce intro: Cons-path)

from \(\text{valid-path as}\) \(\langle as = x\#xs\rangle\) \(\langle xs = xs'@[x']\rangle\)

have valid-path \((x\#xs')\)
  by (simp add: valid-path-def del: valid-path-aux.simps)

  (rule valid-path-aux-split, simp)

with \((-\text{-Entry}) - x\#xs'\to +\) (-\text{-Low}) have \((-\text{-Entry}) - x\#xs'\to +\) (-\text{-Low})
  by (simp add: vp-def)

from \(\langle as = x\#xs\rangle\) \(\langle xs = xs'@[x']\rangle\) have \(as = (x\#xs')@[x']\) by simp

with \(\text{preds (kinds as)}\) \(((\text{cf. undefined}))\)
  have \(\text{preds (kinds } (x\#xs'))\) \(((\text{cf. undefined}))\)
  by (simp add: kinds-def preds-split)

from \((-\text{-Entry}) - as'\to +\) (-\text{-Exit}) obtain \(y\) \(ys\) where \(as' = y\#ys\)
  and \((-\text{-Entry}) = \text{sourcenode } y\) and valid-edge \(y\)
  and targetnode \(y \to ys\) (-\text{-Exit-})

apply (cases \(\text{as'} = []\)

apply (clarsimp simp: vp-def, drule empty-path-nodes, drule Entry-noteq-Exit, simp)

by (fastforce elim: path-split-Cons simp: vp-def)

from \((-\text{-Entry}) - as'\to +\) (-\text{-Exit}) have valid-path \(as'\) by (simp add: vp-def)

from \(\text{valid-edge } y\) 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

\text{case Exit}

with \( \text{targetnode } y \rightarrow ys \rightarrow^* (\text{Exit}) \) have \( ys = [] \)

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

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

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

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

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

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

\( \langle as \rangle = y \# ys \) \( \langle ys \rangle = [] \)

\( \langle \text{kind } z = (\lambda s. \text{False}) \rangle \)

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

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

qed simp

with \( \langle \text{targetnode } y \rightarrow ys \rightarrow^* (\text{Exit}) \rangle \) obtain \( y' \) \( y's \) where \( ys = ys'@[y'] \)

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

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

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

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

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

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

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

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

with \( (\text{Entry}) = y \# y's' \rightarrow^* (\text{Low}) \) \( \text{have } (\text{Entry}) = y \# y's' \rightarrow^* \) \( (\text{Low}) \)

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

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

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

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

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

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

\((\text{Entry}) = x \# x's' \rightarrow^* \) \( (\text{Low}) \) \( \langle \text{preds } (\text{kinds } (x \# x's)) \rangle (\{ \text{cf}, \text{undefined} \}) \)

\( (\text{Entry}) = y \# y's' \rightarrow^* \) \( (\text{Low}) \) \( \langle \text{preds } (\text{kinds } (y \# y's)) \rangle (\{ \text{cf}, \text{undefined} \}) \)

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

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

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

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

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

show \( \text{thesis} \)

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

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

by \((\text{cases transfers } (\text{map kind } y's)) \) \( (\text{transfer } (\text{kind } y)) \rangle (\{ \text{cf}, \text{undefined} \}) \)

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

qed
The second theorem assumes that we have an operational semantics, whose evaluations are written \( \langle c, s \rangle \Rightarrow \langle c', s' \rangle \) 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 \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 list and ParamUses :: 'node \Rightarrow 'var set list
and sem :: 'com \Rightarrow ('var \times 'val) list \Rightarrow 'com \Rightarrow ('var \times 'val) list \Rightarrow bool
\(((1 <\cdot\cdot\cdot 1)) = (1 <\cdot\cdot\cdot 1)) [0,0,0,0] 81\)
and identifies :: 'node \Rightarrow 'com \Rightarrow bool (- \triangleq [51,0] 80)
and H :: 'var set and L :: 'var set
and High :: 'node ('('High'')) and Low :: 'node ('('Low'')) +
fixes final :: 'com \Rightarrow bool
assumes final-edge-Low: [final c; n \triangleq c]

\( \Rightarrow \exists a. \text{valid-edge } a \land \text{sourcenode } a = n \land \text{targetnode } a = (-\text{Low}) \land \text{kind } a = \uparrow id \)

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 (\( \lambda s. \text{True} \)) edge, does not work as the same statement could be identified by several nodes, some initial, some not. E.g., in the program while (\( \text{True} \)) Skip;;Skip two nodes identify this initial statement: the initial node and the node within the loop (because of loop unrolling).

theorem nonInterference:
assumes \([c_f] \approx_L [c_f_2] \land (-\text{High-}) \notin [\text{HRB-slice } S] \text{CFG} \)
and CFG-node (-\text{Low-}) \in S
and valid-edge a and sourcenode a = (-\text{High-}) and targetnode a = n
and kind a = (As. True) \_ and n \triangleq c and final c'
and \( (c,[c_f]) \Rightarrow (c',s_1) \) and \( (c, [c_f_2]) \Rightarrow (c', s_2) \)
shows \( s_1 \approx_L s_2 \)
proof -
from High-target-Entry-edge obtain ax where valid-edge ax
and sourcenode ax = (-Entry-) and targetnode ax = (-High-)
and kind ax = (As. True) \_ by blast
from $n \triangleq c \cdot \langle e, [cf_1] \rangle \Rightarrow \langle e', s_1 \rangle$

obtain $n_1 \ ensure cfs_1 \ where \ n - as_1 \rightarrow \_ \ n_1 \ and \ n_1 \triangleq c'$

and $\text{preds} \ (\text{kinds} \ as_1) \ [(\text{cf}_1\_\text{undefined})]$

and $\text{transfers} \ (\text{kinds} \ as_1) \ [(\text{cf}_1\_\text{undefined})] = cfs_1 \ and \ map \ \text{fst} \ cfs_1 = s_1$

by (fastforce dest: fundamental-property)

from $\langle n - as_1 \rightarrow \_ \ n_1 \rangle \ (\text{valid-edge} \ a) \ \langle \text{sourcecnode} \ a = (-\text{High}) \rangle \ (\text{targetnode} \ a = n)$

$\langle \text{kind} \ a = (\lambda s. \ \text{True}) \rangle$

have $(-\text{High}) - a \# as_1 \rightarrow \_ n_1 \ by (\text{fastforce intro: Cons-path simp: vp-def valid-path-def})$

from $\langle \text{final} \ c' \ (n_1 \triangleq c')$ $\langle \text{valid-edge} \ a_1 \ and \ \text{sourcecnode} \ a_1 = n_1$

and $\text{targetnode} \ a_1 = (-\text{Low}) \ and \ \text{kind} \ a_1 = \uparrow \text{id}$ $by (\text{fastforce dest: final-edge-Low})$

hence $n_1 - [a_1] \rightarrow (\text{-Low})$ $by (\text{fastforce intro: path-edge})$

with $\langle (-\text{High}) - a \# as_1 \rightarrow \_ n_1 \rangle$ $\text{have} \ (-\text{High}) - (a \# as_1)@[a_1] \rightarrow (\text{-Low})$

$by (\text{fastforce intro: path-Append simp: vp-def})$

with $\langle \text{valid-edge} \ ax \rangle \ \langle \text{sourcecnode} \ ax = (\text{-Entry}) \rangle \ \langle \text{targetnode} \ ax = (-\text{High}) \rangle$

have $(-\text{Entry}) - ax \#((a \# as_1)@[a_1]) \rightarrow (\text{-Low})$ $by (\text{rule Cons-path})$

moreover

from $\langle (-\text{High}) - a \# as_1 \rightarrow \_ n_1 \rangle$ $\text{have} \ \langle \text{valid-path-aux} \ [] \ (a \# as_1)$

$by (\text{simp add: vp-def valid-path-def})$

with $\langle \text{kind} \ a_1 = \uparrow \text{id} \rangle$ $\text{have} \ \langle \text{valid-path-aux} \ [] \ (a \# as_1)@[a_1] \rangle$

$by (\text{fastforce intro: valid-path-aux Append})$

with $\langle \text{kind} \ ax = (\lambda s. \ \text{True}) \rangle$

$\text{have} \ \langle \text{valid-path-aux} \ [] \ (ax \#((a \# as_1)@[a_1])) \rangle$

$by \text{simp}$

ultimately have $(-\text{Entry}) - ax \#((a \# as_1)@[a_1]) \rightarrow \_ \ (\text{-Low})$

$by (\text{simp add: vp-def valid-path-def})$

from $\langle \text{valid-edge} \ a \rangle \ \langle \text{kind} \ a = (\lambda s. \ \text{True}) \rangle \ \langle \text{sourcecnode} \ a = (-\text{High}) \rangle$

$\langle \text{targetnode} \ a = n \rangle$

have $\text{get-proc} \ n = \text{get-proc} \ (-\text{High})$

$by (\text{fastforce dest: get-proc-intra simp: intra-kind-def})$

with $\text{get-proc-Low}$ $\text{have} \ \text{get-proc} \ n = \text{Main}$ $by \text{simp}$

from $\langle \text{valid-edge} \ a_1 \rangle \ \langle \text{sourcecnode} \ a_1 = n_1 \rangle \ \langle \text{targetnode} \ a_1 = (-\text{Low}) \rangle \ \langle \text{kind} \ a_1 = \uparrow \text{id} \rangle$

have $\text{get-proc} \ n_1 = \text{get-proc} \ (-\text{Low})$

$by (\text{fastforce dest: get-proc-intra simp: intra-kind-def})$

with $\text{get-proc-Low}$ $\text{have} \ \text{get-proc} \ n_1 = \text{Main}$ $by \text{simp}$

from $\langle n - as_1 \rightarrow \_ n_1 \rangle$ $\text{have} \ n - as_1 \rightarrow \_ n_1$

$by (\text{cases as}_1)$

$\langle \text{auto dest: vpa-Main-slp intro: get-proc n_1 = Main} \ \langle \text{get-proc} \ n = \text{Main} \rangle$

$\text{simp: vp-def valid-path-def valid-call-list-def slp-def}$

$\text{same-level-path-def simp del: valid-path-aux-simps}$

then obtain $\text{cfx r}$ $where \ \text{cfx: transfers} \ (\text{map kind} \ as_1) \ [(\text{cf}_1\_\text{undefined})] = [(\text{cf}_x,r)]$

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

from $\langle \text{kind} \ ax = (\lambda s. \ \text{True}) \rangle \ \langle \text{kind} \ a = (\lambda s. \ \text{True}) \rangle$

$\langle \text{preds} \ (\text{kinds} \ as_1) \ [(\text{cf}_1\_\text{undefined})] \ (\text{kind} \ a_1 = \uparrow \text{id}) \ \text{cfx}$

have $\text{preds} \ (\text{kinds} \ ((a \# (a \# as_1)@[a_1]))) \ [(\text{cf}_1\_\text{undefined})]$

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

from $n \triangleq c \cdot \langle e, [cf_2] \rangle \Rightarrow \langle e', s_2 \rangle$
obtain \( n_2 \) as \( c f s_2 \) where \( n - as \rightarrow \ast \) \( n_2 \) and \( n_2 \triangleq c' \)
and \( \text{preds} = (\text{kinds as2}) \left[ \left( c f s_2, \text{undefined} \right) \right] \)
and \( \text{transfers} = (\text{kinds as2}) \left[ \left( c f s_2, \text{undefined} \right) \right] = c f s_2 \text{ and } \text{map fst cfs} = s_2 \)
by (fastforce dest fundamental-property)
from \( (n - as \rightarrow \ast \) \( n_2 \), valid-edge \( a \), source-node \( a = \) (High), target-node \( a = n \),
\( \langle \text{kind a = (\lambda s. True)\rangle} \)
have \( (\text{Low}) - as \# \rightarrow \ast \) \( n_2 \) by (fastforce intro: Cons-path simp: vp-def valid-path-def)
from \( \langle \text{final c'} \rangle \text{ (n2 \triangleq c')} \)
obtain \( a_2 \) where valid-edge \( a_2 \) and source-node \( a_2 = n_2 \)
and \( \text{target-node} a_2 = (\text{Low}) \) and \( \text{kind} a_2 = \uparrow\text{id} \) by (fastforce dest final-edge-Low)
hence \( n_2 - [a_2] \rightarrow \ast \) (Low) by (fastforce intro: path-edge)
with \( (\text{Low}) - as \# a_2 \rightarrow \ast \) \( n_2 \) have \( (\text{Low}) - (as \# a_2) \circ [a_2] \rightarrow \ast \) (Low)
by (fastforce intro: path-Append simp: vp-def)
with \( \langle \text{valid-edge ax} \rangle \text{ source-node ax = (Entry)}, \langle \text{target-node ax = (High)} \rangle \)
have \( (\text{Entry}) - ax\# ((as \# a_2) \circ [a_2]) \rightarrow \ast \) (Low) by (rule Cons-path)
moreover
from \( (\text{Low}) - as \# a_2 \rightarrow \ast \) \( n_2 \) have valid-path-aux [] (a#as2)
\( \text{by simp add: vp-def valid-path-def} \)
with \( \langle \text{kind a_2 = \uparrow\text{id}} \rangle \text{ have valid-path-aux [] ((a\# as2) \circ [a_2])} \)
\( \text{by (fastforce intro: valid-path-aux-Append)} \)
with \( \langle \text{kind ax = (\lambda s. True)\rangle} \text{ have valid-path-aux [] (a\# ((a\# as2) \circ [n_2]))} \)
\( \text{by simp} \)
ultimately have \( (\text{Entry}) - ax\# ((a\# as2) \circ [a_2]) \rightarrow \ast \) (Low)
\( \text{by (simp add: vp-def valid-path-def)} \)
from \( \langle \text{valid-edge a} \rangle \langle \text{kind a = (\lambda s. True)\rangle} \text{ source-node a = (High)} \)
\( \langle \text{target-node a = n} \rangle \)
have \( \text{get-proc n = get-proc (High)} \)
\( \text{by (fastforce dest: get-proc intra simp: intra-kind-def)} \)
with \( \text{get-proc-High} \text{ have get-proc n = Main by simp} \)
from \( \langle \text{valid-edge a_2} \rangle \langle \text{source-node a_2 = n_2} \rangle \langle \text{target-node a_2 = (Low)} \rangle \langle \text{kind a_2 = \uparrow\text{id}} \rangle \)
have \( \text{get-proc n_2 = get-proc (Low)} \)
\( \text{by (fastforce dest: get-proc intra simp: intra-kind-def)} \)
with \( \text{get-proc-Low} \text{ have get-proc n_2 = Main by simp} \)
from \( \langle n - as \rightarrow \ast \) \( n_2 \rangle \langle \text{have n - as \#} \rightarrow \text{* n_2} \rangle \)
\( \langle \text{cases as2} \rangle \text{ (auto dest! Main-slp intro: get-proc n_2 = Main) (get-proc n = Main)} \text{ simp: vp-def valid-path-def valid-call-list-def slp-def}
\( \text{same-level-path-def simp del: valid-path-aux.simps)} \)
then obtain \( cfz, r' \)
\( \langle \text{where cfz': transfers (map kind as2) \left[ \left( cfz, \text{undefined} \right) \right] = \left[ \left( cfz', r' \right) \right]} \)
\( \text{by (fastforce elim: slp-callstack-length-equal simp: kinds-def)} \)
from \( \langle \text{kind ax = (\lambda s. True)\rangle} \langle \text{kind a = (\lambda s. True)\rangle} \langle \text{preds (kinds as2) \left[ \left( cfz, \text{undefined} \right) \right]} \langle \text{kind a_2 = \uparrow\text{id}} \rangle \langle \text{cfz'} \rangle \langle \text{have preds (kinds (ax\# ((a\# as2) \circ [a_2]))) \left[ \left( cfz, \text{undefined} \right) \right]} \langle \text{by (auto simp: kinds-def preds-split)} \)
from \( \langle \text{cfz} \rangle \approx L \langle \text{cfz} \rangle \langle \text{(High) \notin \left[ \text{HRB-slice S} \right] cfz} \rangle \langle \text{CFG-node (Low) \in S} \rangle \langle (-\text{Entry}) - ax\# ((a\# as1) \circ [a_1]) \rightarrow \ast \text{ (Low)} \rangle \)

31
have \( \text{map \, fst \, (transfers \, (kinds \, (ax \# ((a \# as_1) @ [a_1]))) \, cf_1, \text{undefined})} \approx L \)
by (rule nonInterference-path-to-Low)
with \( \langle \text{kind \, ax} = (\lambda s. \text{True}) \rangle, \langle \text{kind \, a} = (\lambda s. \text{True}) \rangle, \langle \text{kind \, a}_1 = \uparrow \text{id} \rangle, \langle \text{kind \, a}_2 = \uparrow \text{id} \rangle \)
\( \langle \text{transfers \, (kinds \, as_1)} \, cf_1, \text{undefined} \rangle \, \langle \text{map \, fst \, cf_1} = s_1 \rangle \)
\( \langle \text{transfers \, (kinds \, as_2)} \, cf_2, \text{undefined} \rangle \, \langle \text{map \, fst \, cf_2} = s_2 \rangle \)
show \( \text{thesis \, by (cases \, s_1)} (\text{cases \, s_2} (\text{fastforce \, simp:kinds-def \, transfers-split})\,)\,)+\)

end

end

3 Framework Graph Lifting for Noninterference

theory LiftingInter
imports NonInterferenceInter
begin

In this section, we show how a valid CFG from the slicing framework in [8] can be lifted to fulfil all properties of the \( \text{NonInterferenceIntraGraph} \) locale. Basically, we redefine the hitherto existing \( \text{Entry} \) and \( \text{Exit} \) nodes as new \( \text{High} \) and \( \text{Low} \) nodes, and introduce two new nodes \( \text{NewEntry} \) and \( \text{NewExit} \). Then, we have to lift all functions to operate on this new graph.

3.1 Liftings

3.1.1 The datatypes

datatype \('\, \text{node} \, LDCFG\text{-node} = \text{Node} \, '\, \text{node} \\
| \text{NewEntry} \\
| \text{NewExit} \\

type-synonym \('\, \text{edge} \, LDCFG\text{-edge} = \text{LDCFG\text{-edge} = '\, \text{node} \, LDCFG\text{-node} \times (('\, \text{var}, '\, \text{val}, '\, \text{ret}, '\, \text{pname}) \, \text{edge-kind}) \times '\, \text{node} \, LDCFG\text{-node} \\

3.1.2 Lifting basic definitions using \('\, \text{edge} \) and \('\, \text{node} \\
inductive lift-valid-edge :: ('\, \text{edge} \Rightarrow \text{bool}) \Rightarrow ('\, \text{edge} \Rightarrow '\, \text{node}) \Rightarrow ('\, \text{edge} \Rightarrow '\, \text{node}) \\
\Rightarrow ('\, \text{edge} \Rightarrow ('\, \text{var}, '\, \text{val}, '\, \text{ret}, '\, \text{pname}) \, \text{edge-kind}) \Rightarrow '\, \text{node} \Rightarrow '\, \text{node} \Rightarrow ('\, \text{edge}, '\, \text{node}, '\, \text{var}, '\, \text{val}, '\, \text{ret}, '\, \text{pname}) \, \text{LDCFG\text{-edge} \Rightarrow bool} \)
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:
\[ \text{valid-edge} \ a; \ src \ a \neq E \lor \ trg \ a \neq X; \]
\[ e = \text{Node} \ (\text{src} \ a),\ knd \ a,\ \text{Node} \ (\text{trg} \ a)) \]
⇒ lift-valid-edge valid-edge src trg knd E X e

| lve-Entry-edge:
\[ e = \text{NewEntry},(\lambda s. \text{True}) \text{ √},\ \text{Node} \ E \]
⇒ lift-valid-edge valid-edge src trg knd E X e

| lve-Exit-edge:
\[ e = \text{Node} \ X,(\lambda s. \text{True}) \text{ √},\ \text{NewExit} \]
⇒ lift-valid-edge valid-edge src trg knd E X e

| lve-Entry-Exit-edge:
\[ e = \text{NewEntry},(\lambda s. \text{False}) \text{ √},\ \text{NewExit} \]
⇒ 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 ⇒ ('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-edges\[1\]:
\[ e = (\text{Node} \ (\text{src} \ a),\ knd \ a,\ \text{Node} \ (\text{trg} \ a)); \ \text{valid-edge} \ a; \ a' \in \text{get-return-edges} \ a; \]
\[ e' = (\text{Node} \ (\text{src} \ a'),\ knd \ a',\ \text{Node} \ (\text{trg} \ a')) \]
⇒ 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 ⇒

33
'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 LD-CFG-node
  where src a ≡ fst a

abbreviation trg :: (′edge,′node,′var,′val,′ret,′pname) LDCFG-edge ⇒ ′node LD-CFG-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 a)
    (lift-get-proc get-proc Main NewEntry)
    (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(rule wf)
    interpret Postdomination sourcenode targetnode kind valid-edge Entry get-proc get-return-edges procs Main Exit
      by(rule 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'

35
\[
\text{and } \text{src } a = \text{src } a' \text{ and } \text{try } a = \text{try } a' \\
\text{thus } a = a'
\]

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

- **case** lve-edge **thus** case by \(-\text{(erule lift-valid-edge.cases,auto dest:edge-det)}\)
- **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 = \(Q : r \rightarrow \rightarrow \text{Main}\)
- **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 = \(Q' : \leftarrow \rightarrow \text{Main}\)
- **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 = \(Q : r \rightarrow \rightarrow p : fs\)
- **thus** \(\exists \text{ins outs. } (p, \text{ins, outs}) \in \text{set procs}\)
- **by** (fastforce elim: lift-valid-edge.cases intro: callee-in-procs)

**next**

- **fix** \(a Q r p f s\)
- **assume** lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
- **and** knd a = \(Q : r \rightarrow \rightarrow p : fs\)
- **thus** lift-get-proc get-proc Main (src a) = lift-get-proc get-proc Main (try a)
- **by** (fastforce elim: lift-valid-edge.cases intro: get-proc-intra simp: get-proc-Entry get-proc-Exit)

**next**

- **fix** \(a Q' r p f s\)
- **assume** lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
- **and** knd a = \(Q' : \leftarrow \rightarrow p : fs\)
- **thus** lift-get-proc get-proc Main (try a) = p
- **by** (fastforce elim: lift-valid-edge.cases intro: get-proc-call)

**next**

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

**next**

- **fix** \(a Q r p f s\)
- **assume** lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
- **and** knd a = \(Q : r \rightarrow \rightarrow p : fs\)
- **then obtain** \(ax \text{ where valid-edge } ax \text{ and kind } ax = Q : r \rightarrow \rightarrow p : fs\)
- **and** sourcenode ax \(\neq\) Entry \(\lor\) targetnode ax \(\neq\) Exit
- **and** src a = Node (sourcenode ax) and try a = Node (targetnode ax)
- **by** (fastforce elim: lift-valid-edge.cases)
- **from** valid-edge ax \(\langle\text{kind } ax = Q : r \rightarrow \rightarrow p : fs\rangle\)
- **have** all: \(\forall a', \text{valid-edge } a' \land\text{targetnode } a' = \text{targetnode } ax\implies\)
\[(\exists \ Q \ x \ \text{fsx. kind } a' = Q\Sigma x\rightarrow_p \text{fsx})\]

by(auto dest:call-edges-only)

\{ fix \ a' \}
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a'
and \(\text{try } a' = \text{try } a\)

hence \(\exists \ Q \ x \ \text{fsx. kind } a' = Q\Sigma x\rightarrow_p \text{fsx}\)

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

\{fix \(a'\) \}

\text{note} \ ([\text{simp}]) = (e = (\text{Node (sourcenode } ax'), \text{kind } ax', \text{Node (targetnode } ax')))\

from \(\text{try } e = \text{try } a\) \(\text{a = Node (targetnode } ax)\)

\text{have targetnode } ax = \text{Entry by simp}

\text{with (valid-edge } ax'\text{) have False by (rule Entry-target)}

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

next

\text{case (lve-Entry-edge } e)\
from \(\text{e = (NewEntry, (\lambda s. True), Node Entry)\)} \(\text{try } e = \text{try } a\)
\(\text{a = Node (targetnode } ax)\)

\text{have targetnode } ax = \text{Entry by simp}

\text{thus (case by simp)}

next

\text{case (lve-Entry-Exit-edge } e)\
from \(\text{e = (NewEntry, (\lambda s. False), NewExit)\)} \(\text{try } e = \text{try } a\)
\(\text{a = Node (targetnode } ax)\)

\text{have False by simp}

\text{thus (case by simp)}

qed \}

\text{thus} \(\forall a'. \text{ lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit } a' \land \text{try } a' = \text{try } a \rightarrow (\exists \ Q \ x \ \text{fsx. kind } a' = Q\Sigma x\rightarrow_p \text{fsx})\) by simp

next

\text{fix} \ a \ Q' \ p \ f'\
\text{assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit } a\text{ and knd } a = Q\Sigma' \rightarrow_p \text{psx'} \text{ then obtain ax where valid-edge ax and kind } ax = Q\Sigma' \rightarrow_p \text{psx'} \text{ and sourcenode } ax \neq \text{Entry} \lor \text{targetnode } ax \neq \text{Exit} \text{ and src } a = \text{Node (sourcenode } ax)\text{ and try } a = \text{Node (targetnode } ax)\text{ by (fastforce elim:lift-valid-edge.cases)}\text{ from (valid-edge ax) \(\text{kind } ax = Q\Sigma' \rightarrow_p \text{psx'}\),

\text{have all:}\forall a'. \text{ valid-edge } a' \land \text{sourcenode } a' = \text{sourcenode } ax \rightarrow (\exists \ Q \ x \ \text{fsx. kind } a' = Q\Sigma x\rightarrow_p \text{fsx})\text{ by (auto dest:return-edges-only)}\}

\{ fix \ a' \}
\text{assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit } a' \text{ and src } a' = \text{src } a\text{ hence } \(\exists \ Q \ x \ \text{fsx. kind } a' = Q\Sigma x\rightarrow_p \text{fsx}\)
proof (induct rule: lift-valid-edge.induct)
  case (lve-edge ax' e)
  note [simp] = \langle e = (Node (sourcenode ax'), kind ax', Node (targetnode ax'))\rangle
  from \langle src e = src a, \langle src a = Node (sourcenode ax)\rangle\rangle
  have sourcenode ax' = sourcenode ax by simp
  with \langle valid-edge ax'\rangle all have \exists Qx fx. kind ax' = Qx\rightarrow_p fx by blast
  thus \langle case by simp \rangle
  next
  case (lve-Entry-edge e)
  from \langle e = (NewEntry, \langle\lambda s. True\rangle, Node Entry)\rangle \langle src e = src a \rangle
  \langle src a = Node (sourcenode ax)\rangle have False by simp
  thus \langle case by simp \rangle
  next
  case (lve-Exit-edge e)
  from \langle e = (Node Exit, \langle\lambda s. True\rangle, Node Exit)\rangle \langle src e = src a \rangle
  \langle src a = Node (sourcenode ax)\rangle have sourcenode ax = Exit by simp
  with \langle valid-edge ax\rangle have False by (rule Exit-source)
  thus \langle case by simp \rangle
  next
  case (lve-Entry-Exit-edge e)
  from \langle e = (NewEntry, \langle\lambda s. False\rangle, NewExit)\rangle \langle src e = src a \rangle
  \langle src a = Node (sourcenode ax)\rangle have False by simp
  thus \langle case by simp \rangle
  qed 
thus \forall a'. lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a' \land
src a' = src a \rightarrow (\exists Qx fx. kind a' = Qx\rightarrow_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\rightarrow_p fs
thus lift-get-return-edges get-return-edges valid-edge
  sourcenode targetnode kind a \neq \{\}
proof (induct rule: lift-valid-edge.induct)
  case (lve-edge ax e)
  from \langle e = (Node (sourcenode ax), kind ax, Node (targetnode ax))\rangle
  \langle knd e = Q: r\rightarrow_p fs\rangle
  have kind ax = Q: r\rightarrow_p fx by simp
  with \langle valid-edge ax\rangle have get-return-edges ax \neq \{\}
  by (rule get-return-edge-call)
then obtain ax' where ax' \in get-return-edges ax by blast
with \langle e = (Node (sourcenode ax), kind ax, Node (targetnode ax))\rangle \langle valid-edge ax\rangle
  have \langle Node (sourcenode ax'), kind ax', Node (targetnode ax')\rangle \in
  lift-get-return-edges get-return-edges valid-edge
  sourcenode targetnode kind e
  by (fastforce intro: lift-get-return-edgesI)
thus \langle case by fastforce \rangle
qed simp-all
next
fix a a'
assume a' ∈ lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind a
and lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
thus lift-valid-edge valid-edge sourcenode 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' by (fastforce dest!: call-return-edges)
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 simp
  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 sourcenode targetnode kind a
and lift-valid-edge valid-edge sourcenode 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 ⟨valid-edge ax⟩ ⟨a' ∈ get-return-edges ax⟩ have ∃ Q r p fs, kind ax = Q; r→p; fs
  by (rule only-call-get-return-edges)
with ⟨a = (Node (sourcenode ax), kind ax, Node (targetnode ax))⟩
  ⟨valid-edge a'⟩
  show ?case by simp
qed
next
fix a Q r p fs a'
assume a' ∈ lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind a
and kind a = Q; r→p; fs
and lift-valid-edge valid-edge sourcenode 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))⟩...
∃a. lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a ∧ (∃Q r fs. kind a' = Q'→ p fs) ∧ a ∈ lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind a'  

proof (induct rule: lift-valid-edge.induct)  
  case (lve-edge e)  
    from (valid-edge e) (valid-edge a')  
    show False by simp  
  qed  
next  
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a  
  and kind a = Q'→ p fs  
  thus ∃a'. lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a' ∧ (∃Q r fs. kind a' = Q'→ p fs) ∧ a ∈ lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind a'  

proof (rule: lift-valid-edge.induct)  
  case (lve-edge e)  
    from (valid-edge e) (valid-edge a')  
    show False by simp  
  qed
source node target node kind \( x \)

from ⟨lift-valid-edge valid-edge source node target node kind Entry Exit \( x \)⟩
\( \exists Q r fs. \text{kind } x = Q;r\to fs \) obtain \( y \) where valid-edge \( y \)
and \( x = (\text{Node } (\text{source node } y), \text{kind } y, \text{Node } (\text{target node } y)) \)
by (fastforce elim:lift-valid-edge.cases)
with \( e \in\) lift-get-return-edges get-return-edges valid-edge
source node target node kind \( x \) ⟨valid-edge \( a \)⟩
\( e = (\text{Node } (\text{source node } a), \text{kind } a, \text{Node } (\text{target node } a)) \)

have \( x = ?e' \)
proof (induct rule: lift-get-return-edges.induct)
  case ⟨lift-get-return-edges\( _I \) ax ax' \( e \)⟩
from ⟨valid-edge ax⟩ ⟨ax' ∈ get-return-edges ax⟩ have valid-edge ax'
by (rule get-return-edges-valid)
from \( e = (\text{Node } (\text{source node } ax'), \text{kind } ax', \text{Node } (\text{target node } ax')) \)
\( e = (\text{Node } (\text{source node } a), \text{kind } a, \text{Node } (\text{target node } a)) \)
have source node \( a = \) source node \( ax' \) and target node \( a = \) target node \( ax' \)
by simp-all
with ⟨valid-edge a⟩ ⟨valid-edge ax'⟩ have [simp]: \( a = ax' \) by (rule edge-det)
from ax = (Node (source node ax), kind ax, Node (target node ax))
\( \exists Q r fs. \text{kind } x = Q;r\to fs \) have \( \exists Q r fs. \text{kind } x = Q;r\to fs \) by simp
with ⟨valid-edge ax⟩ ⟨ax' ∈ get-return-edges ax⟩ imp
have ax = ax' by fastforce
with \( x = ax' \) show \( ?\text{thesis} \) by simp

qed }
ultimately show \( ?\text{case by}(\text{blast intro:exI}) \)
qed simp-all

next
fix \( a \) \( a' \)
assume \( a' ∈\) lift-get-return-edges get-return-edges valid-edge source node
target node kind \( a \)
and lift-valid-edge valid-edge source node target node kind Entry Exit \( a \)
thus \( \exists a'', \text{lift-valid-edge valid-edge source node target node kind Entry Exit } a'' \land \)
src \( a'' = \text{try } a \land \text{try } a'' = \text{src } a' \land \text{kind } a'' = (\lambda cf. \text{False}) √ \)
proof (induct rule: lift-get-return-edges.induct)
  case ⟨lift-get-return-edges\( _I \) ax ax' \( e \)⟩
from ⟨valid-edge ax⟩ ⟨ax' ∈ get-return-edges ax⟩
obtain ax' where valid-edge ax' and source node ax' = target node ax
and target node ax' = source node a' and kind ax' = (\lambda cf. \text{False}) √
by (fastforce dest: intra-proc-additional-edge)
let \( ?\text{ex} = (\text{Node } (\text{source node } ax'), \text{kind } ax', \text{Node } (\text{target node } ax')) \)
have target node ax ≠ Entry
proof
  assume target node ax = Entry
with ⟨valid-edge ax⟩ show False by (rule Entry-target)
qed
with ⟨source node ax' = target node ax⟩ have source node ax' ≠ Entry by simp
with ⟨valid-edge ax'⟩
have lift-valid-edge valid-edge source node target node kind Entry Exit ?ex

41
\[ \begin{align*}
&\text{by (fastforce intro:live-edge)} \\
&\text{with } \langle e' = (Node (sourcenode a'), \text{kind } a', \text{Node (targetnode } a')) \rangle \\
&\langle a = (Node (sourcenode ax), \text{kind } ax, \text{Node (targetnode } ax)) \rangle \\
&\langle e' = (Node (sourcenode a'), \text{kind } a', \text{Node (targetnode } a')) \rangle \\
&\langle \text{sourcenode } ax' = \text{targetnode } ax \rangle \langle \text{targetnode } ax' = \text{sourcenode } a' \rangle \\
&\langle \text{kind } ax' = (\lambda cf. \ False) \rangle, \\
&\text{show } ?\text{case by simp} \\
&\text{qed} \\
&\text{next} \\
&\text{fix } a a' \\
&\text{assume } a' \in \text{l}ift\text{-get-return-edges get-return-edges valid-edge sourcenode} \\
&\text{targetnode kind } a \\
&\text{and } \text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit } a \\
&\text{thus } ?a''. \text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit } a'' \land \\
&\text{src } a'' = \text{src } a \land \text{try } a'' = \text{try } a' \land \text{knd } a'' = (\lambda cf. \ False) \checkmark \\
&\text{proof (induct rule: lift-get-return-edges.induct)} \\
&\text{case } (\text{l}ift\text{-get-return-edgesI } ax a' e') \\
&\text{from } \langle \text{valid-edge } ax \rangle \langle a' \in \text{get-return-edges } ax \rangle \\
&\text{obtain } ax' \text{ where valid-edge } ax' \text{ and sourcenode } ax' = \text{sourcenode } ax \\
&\text{and targetnode } ax' = \text{targetnode } a' \text{ and kind } ax' = (\lambda cf. \ False) \checkmark \\
&\text{by (fastforce dest: call-return-node-edge)} \\
&\text{let } ?ex = (\text{Node (sourcenode } ax'), \text{kind } ax', \text{Node (targetnode } ax')) \\
&\text{from } \langle \text{valid-edge } ax \rangle \langle a' \in \text{get-return-edges } ax \rangle \\
&\text{obtain } Q r p fs \text{ where kind } ax = Q; r \rightarrow p; fs \\
&\text{by (fastforce dest: ! only-call-get-return-edges)} \\
&\text{have } \text{sourcenode } ax \neq \text{Entry} \\
&\text{proof} \\
&\text{assume } \text{sourcenode } ax = \text{Entry} \\
&\text{with } \langle \text{valid-edge } ax \rangle \langle \text{kind } ax = Q; r \rightarrow p; fs \rangle \text{ show False} \\
&\text{by (rule Entry-no-call-source)} \\
&\text{qed} \\
&\text{with } \langle \text{sourcenode } ax' = \text{sourcenode } ax \rangle \text{ have } \text{sourcenode } ax' \neq \text{Entry} \text{ by simp} \\
&\text{with } \langle \text{valid-edge } ax' \rangle \\
&\text{have } \text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit } ?ex \\
&\text{by (fastforce intro: live-edge)} \\
&\text{with } \langle e' = (\text{Node (sourcenode } a'), \text{kind } a', \text{Node (targetnode } a')) \rangle \\
&\langle a = (\text{Node (sourcenode } ax), \text{kind } ax, \text{Node (targetnode } ax)) \rangle \\
&\langle e' = (\text{Node (sourcenode } a'), \text{kind } a', \text{Node (targetnode } a')) \rangle \\
&\langle \text{sourcenode } ax' = \text{sourcenode } ax \rangle \langle \text{targetnode } ax' = \text{targetnode } a' \rangle \\
&\langle \text{kind } ax' = (\lambda cf. \ False) \rangle \checkmark \\
&\text{show } ?\text{case by simp} \\
&\text{qed} \\
&\text{next} \\
&\text{fix } a Q r p fs \\
&\text{assume } \text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit } a \\
&\text{and } \text{knd } a = Q; r \rightarrow p; fs \\
&\text{thus } ?a''. \text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit } a'' \land \\
&\text{src } a'' = \text{src } a \land \text{intra-kind } (\text{knd } a') \\
&\text{proof (induct rule: lift-valid-edge.induct)} \\
\end{align*}\]
case (lve-edge a e)
  from \( e = (\text{Node} (\text{sourcenode } a), \text{kind } a, \text{Node} (\text{targetnode } a)) \) \( \langle \text{knd } e = Q:r\rightarrow pfs \rangle \)
  have \( \text{knd } a = Q:r\rightarrow pfs \) by simp
  with \( \langle \text{valid-edge } a \rangle \) have \( \exists ! a'. \text{valid-edge } a' \land \text{sourcenode } a' = \text{sourcenode } a \)
  \( \land \text{intra-kind}(\text{knd } a') \) by (rule call-only-one-intra-edge)
  then obtain \( a' \) where \( \text{valid-edge } a' \land \text{sourcenode } a' = \text{sourcenode } a \)
  and \( \text{intra-kind}(\text{knd } a') \)
  and \( \text{imp} : \forall x. \text{valid-edge } x \land \text{sourcenode } x = \text{sourcenode } a \land \text{intra-kind}(\text{knd } x) \)
  \( \rightarrow x = a' \) by (fastforce elim:ex1E)
  let \( ?e' = (\text{Node} (\text{sourcenode } a'), \text{kind } a', \text{Node} (\text{targetnode } a')) \)
  have \( \text{sourcenode } a \neq \text{Entry} \) by simp
  proof
    assume \( \text{sourcenode } a = \text{Entry} \)
    with \( \langle \text{valid-edge } a \rangle \) \( \langle \text{kind } a = Q:r\rightarrow pfs \rangle \) show False
    by (rule Entry-no-call-source)
  qed
  with \( \langle \text{sourcenode } a' = \text{sourcenode } a \rangle \) have \( \text{sourcenode } a' \neq \text{Entry} \) by simp
  with \( \langle \text{valid-edge } a' \rangle \)
  have \( \text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit } ?e' \)
  by (fastforce intro:lift-valid-edge.lve-edge)
  moreover
  from \( \langle e = (\text{Node} (\text{sourcenode } a), \text{kind } a, \text{Node} (\text{targetnode } a)) \rangle \)
  \( \langle \text{sourcenode } a' = \text{sourcenode } a \rangle \)
  have \( \text{src } ?e' = \text{src } e \) by simp
  moreover
  from \( \langle \text{intra-kind}(\text{knd } a') \rangle \) have \( \text{intra-kind} (\text{knd } ?e') \) by simp
  moreover
  \{ fix x
    assume \( \text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit } x \)
    and \( \text{src } x = \text{src } e \) and \( \text{intra-kind } (\text{knd } x) \)
    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 (lve-edge ax ex)
      from \( \langle \text{intra-kind } (\text{knd } x) \rangle \) \( \langle x = ex \rangle \) \( \langle \text{src } x = \text{src } e \rangle \)
      \( \langle ex = (\text{Node} (\text{sourcenode } ax), \text{kind } ax, \text{Node} (\text{targetnode } ax)) \rangle \)
      \( \langle e = (\text{Node} (\text{sourcenode } a), \text{kind } a, \text{Node} (\text{targetnode } a)) \rangle \)
      have \( \text{intra-kind } (\text{kind } ax) \) and \( \text{sourcenode } ax = \text{sourcenode } a \) by simp-all
      with \( \langle \text{valid-edge } ax \rangle \) \( \text{imp } \langle \text{ax = a' } \rangle \) by fastforce
      with \( \langle x = ex \rangle \) \( \langle ex = (\text{Node} (\text{sourcenode } ax), \text{kind } ax, \text{Node} (\text{targetnode } ax)) \rangle \)
      show ?case by simp
    next
      case (lve-Entry-edge ex)
      with \( \langle \text{src } x = \text{src } e \rangle \)
      \( \langle e = (\text{Node} (\text{sourcenode } a), \text{kind } a, \text{Node} (\text{targetnode } a)) \rangle \)
  \}
  43
have False by simp
thus ?case by simp
next
case (lve-Exit-edge ex)
  with \( \langle \text{src } x = \text{src } e \rangle \)
  \( \langle e = (\text{Node (sourcenode } a), \text{kind } a, \text{Node (targetnode } a)) \rangle \)
  have sourcenode \( a = \text{Exit} \) by simp
  with \( \langle \text{valid-edge } a \rangle \) have False by (rule Exit-source)
  thus ?case by simp
next
case (lve-Entry-Exit-edge ex)
  with \( \langle \text{src } x = \text{src } e \rangle \)
  \( \langle e = (\text{Node (sourcenode } a), \text{kind } a, \text{Node (targetnode } a)) \rangle \)
  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 \( a = Q'\leftarrow_p f' \)
thus \( \exists !a'. \) lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit \( a' \) \( \land \)
try \( a' = \) try \( a \) \( \land \) intra-kind (knd \( a' \))
proof (induct rule:lift-valid-edge.induct)
case (lve-edge \( a \) \( e \))
from \( \langle e = (\text{Node (sourcenode } a), \text{kind } a, \text{Node (targetnode } a)) \rangle \) \( \langle \text{kind } e = Q'\leftarrow_p f' \rangle \)
  have kind \( a = Q'\leftarrow_p f' \) by simp
  with \( \langle \text{valid-edge } a \rangle \) have \( \exists !a'. \) valid-edge \( a' \) \( \land \) targetnode \( a' = \) targetnode \( a \) \( \land \) intra-kind (knd \( a' \))
  then obtain \( a' \) where valid-edge \( a' \) and targetnode \( a' = \) targetnode \( a \)
  and intra-kind (knd \( a' \))
  and simp:\( \forall x. \) valid-edge \( x \) \( \land \) targetnode \( x = \) targetnode \( a \) \( \land \) intra-kind (kind \( x \))
  \( \rightarrow x = a' \) by (fastforce elim:ex1E)
let \( ?e' = (\text{Node (sourcenode } a'), \text{kind } a', \text{Node (targetnode } a')) \)
have targetnode \( a \neq \text{Exit} \)
proof
  assume targetnode \( a = \text{Exit} \)
  with \( \langle \text{valid-edge } a' \rangle \) \( \langle \text{kind } a = Q'\leftarrow_p f' \rangle \) show False
  by (rule Exit-no-return-target)
qed
with \( \langle \text{targetnode } a' = \text{targetnode } a \rangle \) have targetnode \( a' \neq \text{Exit} \) by simp
with \( \langle \text{valid-edge } a' \rangle \)
have lift-valid-edge valid-edge sourcenode targetnode kind Entry \( \text{Exit } ?e' \)
  by (fastforce intro:lift-valid-edge.lve-edge)
moreover
from \( \langle e = (\text{Node (sourcenode } a), \text{kind } a, \text{Node (targetnode } a)) \rangle \)
\( \langle \text{targetnode } a' = \text{targetnode } a \rangle \)

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

moreover

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

moreover

\[
\{ \text{fix } x \\
\text{assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit } x \\
\text{and } \text{trg } x = \text{trg } e \text{ and intra-kind } (\text{knd } x) \\
\text{from } \langle \text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit } x \rangle \\
\text{have } x = ?e' \}
\]

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

\[
\text{case (lve-edge ax ex)} \\
\text{from } \langle \text{intra-kind } (\text{knd } x) \rangle \langle x = ex \rangle \langle \text{trg } x = \text{trg } e \rangle \\
\langle e = (\text{Node } \text{(sourcenode } ax), \text{kind } ax, \text{Node } \text{(targetnode } ax)) \rangle \\
\text{have intra-kind } (\text{kind } ax) \text{ and targetnode } ax = \text{targetnode } a \text{ by simp-all} \\
\text{with } \langle \text{valid-edge ax} \rangle \text{ imp have } ax = a' \text{ by fastforce} \\
\text{with } (x = ex) \langle ex = (\text{Node } \text{(sourcenode } ax), \text{kind } ax, \text{Node } \text{(targetnode } ax)) \rangle \\
\text{show } ?\text{case by simp} \\
\text{next} \\
\text{case (lve-Entry-edge ex)} \\
\text{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 \\
\text{have targetnode } a = \text{Entry by simp} \\
\text{with } \langle \text{valid-edge ax} \rangle \text{ have False by(rule Entry-target)} \\
\text{thus } ?\text{case by simp} \\
\text{next} \\
\text{case (lve-Exit-edge ex)} \\
\text{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 \\
\text{have False by simp} \\
\text{thus } ?\text{case by simp} \\
\text{next} \\
\text{case (lve-Entry-Exit-edge ex)} \\
\text{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 \\
\text{have False by simp} \\
\text{thus } ?\text{case by simp} \\
\text{next} \\
\text{ultimately show } ?\text{case by(blast intro:ex1I)} \\
\text{qed simp-all} \\
\text{next} \\
\text{fix } a a' Q_1 r_1 p f s_1 Q_2 r_2 f s_2 \\
\text{assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit } a \\
\text{and lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit } a' \\
\text{and knd } a = Q_1;r_1 \mapsto p f s_1 \text{ and knd } a' = Q_2;r_2 \mapsto p f s_2 \\
\text{then obtain } x x' \text{ where valid-edge } x \\
\text{and } a: a = (\text{Node } \text{(sourcenode } x), \text{kind } x, \text{Node } \text{(targetnode } x)) \text{ and valid-edge}
\( x' \)

and \( a' = (\text{Node (sourcenode } x'), \text{kind } x', \text{Node (targetnode } x')) \)
by (auto elim!: lift-valid-edge. cases)
with \( \text{knd } a = Q_1: r_1 \mapsto p_{fs_1} \text{ and } knd a' = Q_2: r_2 \mapsto p_{fs_2} \)
have \( \text{kind } x = Q_1: r_1 \mapsto p_{fs_1} \text{ and } \text{kind } x' = Q_2: r_2 \mapsto p_{fs_2} \)
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 \( (p, \text{ins}, \text{outs}) \in \text{set procs} \)
from distinct-formal-ins[OF this] show distinct ins.
next
fix \( p \) ins outs
assume \( (p, \text{ins}, \text{outs}) \in \text{set procs} \)
from distinct-formal-outs[OF this] show distinct outs.
qed
qed

lemma lift-CFG-wf:
assumes \( \text{wf:CFGExit-wf sourcenode targetnode kind valid-edge Entry get-proc get-return-edges procs Main Exit Def Use ParamDefs ParamUses} \)
and \( \text{pd:Postdomination sourcenode targetnode kind valid-edge Entry get-proc get-return-edges procs Main Exit} \)
shows \( \text{CFG-wf src trg knd} \)
(lift-valid-edge \( \text{sourcenode targetnode kind Entry Exit} \) \( \text{NewEntry} \)
(lift-get-proc \( \text{get-proc Main} \)
(lift-get-return-edges \( \text{get-return-edges valid-edge sourcenode targetnode kind} \)
procs Main (lift-Def Def Entry Exit H L) (lift-Use Use Entry Exit H L)
(lift-ParamDefs ParamDefs) (lift-ParamUses ParamUses)
proof
interpret \( \text{CFGExit-wf sourcenode targetnode kind valid-edge Entry get-proc get-return-edges procs Main Exit Def Use ParamDefs ParamUses} \)
by (rule \( \text{wf} \))
interpret \( \text{Postdomination sourcenode targetnode kind valid-edge Entry get-proc get-return-edges procs Main Exit} \)
by (rule \( \text{pd} \))
interpret \( \text{CFG:CFG src trg knd} \)
lift-valid-edge \( \text{valid-edge sourcenode targetnode kind Entry Exit NewEntry} \)
lift-get-proc \( \text{get-proc Main} \)
lift-get-return-edges \( \text{get-return-edges valid-edge sourcenode targetnode kind} \)
procs Main
by (fastforce intro: lift-CFG \( \text{wf pd} \))
show \( \)thesis
proof
show \( \text{lift-Def Def Entry Exit H L NewEntry} = \{ \} \) ∧
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;r→_pfs and (p, ins, outs) ∈ set proc
thus length (lift-ParamUses ParamUses (src a)) = length ins
proof (induct rule: lift-valid-edge.induct)
case (lev-edge a e)

from e = (Node (sourcenode a), kind a, Node (targetnode a))

have knd a = Q;r→_pfs and src e = Node (sourcenode a) by simp-all
with ⟨valid-edge a, ⟨p, ins, outs⟩ ∈ set proc⟩

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 (trg a))
proof (induct rule: lift-valid-edge.induct)
case (lev-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

next

case (lev-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'fs and (p, ins, outs) ∈ set proc
thus length (lift-ParamDefs ParamDefs (trg a)) = length outs
proof (induct rule: lift-valid-edge.induct)
case (lev-edge a e)
from ⟨e = (Node (sourcenode a), kind a, Node (targetnode a))⟩

⟨knd e = Q'_;p'fs⟩

have knd a = Q'_;p'fs and trg e = Node (targetnode a) by simp-all

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

have \( \text{length}(\text{ParamDefs } (\text{targetnode } a)) = \text{length } \text{outs} \)
by \(-\text{(rule ParamDefs-return-target-length)} \)

with \( \langle \text{try } e = \text{Node } (\text{targetnode } a) \rangle \) show ?case by simp

qed simp-all

next

fix \( n \) \( V \)

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

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

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

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

proof\(\text{(erule disjE)}\)+

assume \( n = \text{NewExit} \)

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

next

assume \( n = \text{NewExit} \)

with \( \langle V \in \text{set } (\text{lift-ParamDefs } \text{ParamDefs } n) \rangle \) show ?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 \( \langle n = \text{Node } m \rangle \langle V \in \text{set } (\text{lift-ParamDefs } \text{ParamDefs } n) \rangle \)

have \( V \in \text{set } (\text{ParamDefs } m) \) 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 ?thesis by\(\text{(fastforce intro:lift-Def-node)}\)

qed

next

fix \( a \) \( Q \) \( r \) \( p \) \( f s \) \( \text{ins} \) \( \text{outs} \) \( V \)

assume \( \text{lift-valid-edge } \text{valid-edge } \text{sourcenode } \text{targetnode } \text{kind } \text{Entry Exit } a \)
and \( \text{knd } a = Q: r \rightarrow p f s \) \( (p, \text{ins}, \text{outs}) \in \text{set procs} \) \( n \) \( V \in \text{set } \text{ins} \)

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

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

case \( (\text{lev-edge } a \ e) \)

from \( \langle e = (\text{Node } (\text{sourcenode } a), \text{knd } a, \text{Node } (\text{targetnode } a)) \rangle \) \( \langle \text{knd } e = Q: r \rightarrow p f s \rangle \)

have \( \text{knd } a = Q: r \rightarrow p f s \) by simp

from \( \langle \text{valid-edge } a \rangle \langle \text{knd } a = Q: r \rightarrow p f s \rangle \langle (p, \text{ins}, \text{outs}) \in \text{set procs} \rangle \langle V \in \text{set } \text{ins} \rangle \)

have \( V \in \text{Def } (\text{targetnode } a) \) by\(\text{(rule ins-in-Def)}\)

from \( \langle e = (\text{Node } (\text{sourcenode } a), \text{knd } a, \text{Node } (\text{targetnode } a)) \rangle \)

have \( \langle \text{try } e = \text{Node } (\text{targetnode } a) \rangle \) by simp

with \( \langle V \in \text{Def } (\text{targetnode } a) \rangle \) show ?case by\(\text{(fastforce intro:lift-Def-node)}\)

qed simp-all

next

fix \( a \) \( Q \) \( r \) \( p \) \( f s \)

assume \( \text{lift-valid-edge } \text{valid-edge } \text{sourcenode } \text{targetnode } \text{kind } \text{Entry Exit } a \)
and \( \text{knd } a = Q: r \rightarrow p f s \)

48
thus \( \text{lift-Def} \text{ Def} \text{ Entry} \text{ Exit} H L (\text{src} a) = \{\} \)

\[
\text{proof}(\text{induct rule:lift-valid-edge.induct})
\]
\[
\text{case} (\text{lee-edge} a e)
\]
\[
\text{show} \ ?\text{case}
\]
\[
\text{proof}(\text{rule ccontr})
\]
\[
\text{assume} \ \text{lift-Def} \text{ Def} \text{ Entry} \text{ Exit} H L \ (\text{src} e) \neq \{\}
\]
\[
\text{then obtain} \ x \ \text{where} \ x \in \text{lift-Def} \text{ Def} \text{ Entry} \text{ Exit} H L \ (\text{src} e) \ \text{by blast}
\]
\[
\text{from} \ \langle e = (\text{Node} (\text{source} a), \text{kind} a, \text{Node} (\text{target} a))\rangle \ \langle \text{knd} e = Q : r \hookrightarrow \rightarrow p \text{fs} \rangle
\]
\[
\text{have} \ \text{kind} a = Q : r \hookrightarrow \rightarrow p \text{fs} \ \text{by simp}
\]
\[
\text{with} \ \langle \text{valid-edge} a \rangle \ \text{have} \ \text{Def} (\text{source} a) = \{\}
\]
\[
\text{by}(\text{rule call-source-Def-empty})
\]
\[
\text{have} \ \text{source} a \neq \text{Entry}
\]
\[
\text{proof}
\]
\[
\text{assume} \ \text{source} a = \text{Entry}
\]
\[
\text{with} \ \langle \text{valid-edge} a \rangle \ \text{have} \ \langle \text{knd} a = Q : r \hookrightarrow \rightarrow p \text{fs} \rangle
\]
\[
\text{show} \ \text{False} \ \text{by}(\text{rule Entry-no-call-source})
\]
\[
\text{qed}
\]
\[
\text{from} \ \langle e = (\text{Node} (\text{source} a), \text{kind} a, \text{Node} (\text{target} a))\rangle
\]
\[
\text{have} \ \text{src} e = \text{Node} (\text{source} a)
\]
\[
\text{with} \ \langle \text{Def} (\text{source} a) = \{\}, \ x \in \text{lift-Def} \text{ Def} \text{ Entry} \text{ Exit} H L \ (\text{src} e)\rangle
\]
\[
\langle \text{source} a \neq \text{Entry} \rangle
\]
\[
\text{show} \ \text{False} \ \text{by}(\text{fastforce elim:lift-Def-set.cases})
\]
\[
\text{qed}
\]
\[
\text{qed} \ \text{simp-all}
\]
\[
\text{next}
\]
\[
\text{fix} \ n \ V
\]
\[
\text{assume} \ \text{CFG}, \text{CFG.valid-node} \ \text{src} \ \text{try}
\]
\[
\text{(lift-valid-edge valid-edge source target kind Entry Exit) n}
\]
\[
\text{and} \ V \in \bigcup \text{(set (lift-ParamUses ParamUses n))}
\]
\[
\text{hence} \ ((n = \text{NewEntry}) \vee n = \text{NewExit}) \vee (\exists m. n = \text{Node} m \land \text{valid-node} m)
\]
\[
\text{by}(\text{auto elim:lift-valid-edge.cases simp:CFG.valid-node-def})
\]
\[
\text{thus} \ V \in \text{lift-Use Use Entry Exit H L n apply –}
\]
\[
\text{proof}(\text{erule disjE})+
\]
\[
\text{assume} \ n = \text{NewEntry}
\]
\[
\text{with} \langle V \in \bigcup \text{(set (lift-ParamUses ParamUses n))}\rangle \ \text{show} \ ?\text{thesis} \ \text{by simp}
\]
\[
\text{next}
\]
\[
\text{assume} \ n = \text{NewExit}
\]
\[
\text{with} \langle V \in \bigcup \text{(set (lift-ParamUses ParamUses n))}\rangle \ \text{show} \ ?\text{thesis} \ \text{by simp}
\]
\[
\text{next}
\]
\[
\text{assume} \ \exists m. n = \text{Node} m \land \text{valid-node} m
\]
\[
\text{then obtain} \ m \ \text{where} \ n = \text{Node} m \ \text{and} \ \text{valid-node} m \ \text{by blast}
\]
\[
\text{from} \langle V \in \bigcup \text{(set (lift-ParamUses ParamUses n))}\rangle \ \langle n = \text{Node} m\rangle
\]
\[
\text{have} \ V \in \bigcup \text{(set (ParamUses m))} \ \text{by simp}
\]
\[
\text{with} \ \langle \text{valid-node} m \rangle \ \text{have} \ V \in \text{Use} m \ \text{by}(\text{rule ParamUses-in-Use})
\]
\[
\text{with} \langle n = \text{Node} m\rangle \ \text{show} \ ?\text{thesis} \ \text{by}(\text{fastforce intro:lift-Use-node})
\]
\[
\text{qed}
\]
\[
\text{next}
\]
fix a Q p f ins outs V
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and \( knd a = Q \leftrightarrow p f \) and \((p, ins, outs) \in set procs\) and \( V \in set outs\)
thus \( V \in lift-Use Use Entry Exit H L (src a)\)
proof (induct rule:lift-valid-edge.induct)
  case (lee-edge a e)
  from \( e = (Node (sourcenode a), kind a, Node (targetnode a))\)
  \( \langle knd e = Q \leftrightarrow p f \rangle\)
  have \( knd a = Q \leftrightarrow p f \) by simp
  from \( \langle valid-edge a \rangle \langle knd a = Q \leftrightarrow p f \rangle \langle (p, ins, outs) \in set procs\rangle \langle V \in set outs\rangle\)
  have \( V \in Use (sourcenode a) by (rule outs-in-Use)\)
  from \( e = (Node (sourcenode a), kind a, Node (targetnode a))\)
  have \( src e = Node (sourcenode a) by simp\)
  with \( \langle V \in Use (sourcenode a)\rangle show \ cases by (fastforce intra:lift-Use-node)\)
  qed simp-all
next
fix a V s
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and \( V \notin lift-Def Def Entry Exit H L (src a)\) and intra-kind \( (knd a)\)
and \( pred (knd a) s\)
thus state-val (transfer \( (knd a) s\) \( s = state-val s V\)
proof (induct rule:lift-valid-edge.induct)
  case (lee-edge a e)
  from \( e = (Node (sourcenode a), kind a, Node (targetnode a))\)
  \( \langle intra-kind (knd e) \rangle \langle pred (knd e) s\rangle\)
  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 \( V \notin lift-Def Def Entry Exit H L (src e) \langle src e = Node (sourcenode a)\rangle\)
  have \( V \notin Def (sourcenode a) by (auto dest: lift-Def-node)\)
  from \( \langle valid-edge a \rangle \langle V \notin Def (sourcenode a) \rangle \langle intra-kind (knd a)\rangle \langle pred (knd a) s\rangle\)
  have state-val (transfer \( (knd a) s\) \( s = state-val s V\)
  by (rule CFG-intra-edge-no-Def-equal)
  with \( \langle knd e = kind a\rangle show \ cases by simp\)
next
  case (lee-Entry-edge e)
  from \( e = (NewEntry, (\lambda s. True) \sqrt{}/, Node Entry)\)
  \( \langle pred (knd e) s\rangle\)
  show \ cases by (cases s) auto
next
  case (lee-Exit-edge e)
  from \( e = (Node Exit, (\lambda s. True) \sqrt{}/, NewExit)\)
  \( \langle pred (knd e) s\rangle\)
  show \ cases by (cases s) auto
next
  case (lee-Entry-Exit-edge e)
  from \( e = (NewEntry, (\lambda s. False) \sqrt{}/, NewExit)\)
  \( \langle pred (knd e) s\rangle\)
  have False by (cases s) auto
  thus \ cases by simp
qed
next
fix \(a \ s \ s'\)
assume \(\text{assms:lift-valid-edge \ valid-edge \ sourcenode \ targetnode \ kind \ Entry \ Exit} \ a\)
\[
\forall V \in \text{lift-Use} \ \text{Use Entry Exit} \ H L (\text{src} \ a). \ \text{state-val} \ s \ V = \text{state-val} \ s' \ V
\]
intra-kind (\(\text{kind} a\)) \ pred (\(\text{kind} a\)) \ s \ pred (\(\text{kind} a\)) \ s'
show \(\forall V \in \text{lift-Def} \ \text{Def Entry Exit} \ H L (\text{src} \ a). \ \text{state-val} \ (\text{transfer} \ (\text{kind} a) \ s) \ V = \text{state-val} \ (\text{transfer} \ (\text{kind} a) \ s') \ V\)
proof
fix \(V\) assume \(V \in \text{lift-Def} \ \text{Def Entry Exit} \ H L (\text{src} \ a)\)
with \(\text{assms}\)
show \text{state-val} \ (\text{transfer} \ (\text{kind} a) \ s) \ V = \text{state-val} \ (\text{transfer} \ (\text{kind} a) \ s') \ V\)
proof (induct rule:lift-valid-edge.induct)
case (\(\text{live-edge} \ a \ e\))
from \(e = (\text{Node} \ (\text{sourcenode} \ a), \ \text{kind} a, \ \text{Node} \ (\text{targetnode} \ a))\)
\(\langle\text{intra-kind} \ (\text{kind} e)\rangle\) \ have \(\text{intra-kind} \ (\text{kind} a)\) by simp
show ?thesis by simp
next
case True
hence \(\text{sourcenode} \ a = \text{Entry}\) by simp
from \(\text{Entry-Exit-edge}\) obtain \(a'\) where \(\text{valid-edge} \ a'\)
and \(\text{sourcenode} \ a' = \text{Entry}\) and \(\text{targetnode} \ a' = \text{Exit}\)
and \(\text{kind} a' = (\lambda s. \text{False})\)
by blast
have \(\exists \ Q. \ \text{kind} a = (Q)\)
proof (cases \(\text{targetnode} \ a = \text{Exit}\))
case True
with \(\langle\text{valid-edge} \ a\rangle \ \langle\text{valid-edge} \ a'\rangle \ \langle\text{sourcenode} \ a = \text{Entry}\rangle\)
\(\langle\text{sourcenode} \ a' = \text{Entry}\rangle \ \langle\text{targetnode} \ a' = \text{Exit}\rangle\)
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 \(\langle\text{valid-edge} \ a\rangle \ \langle\text{valid-edge} \ a'\rangle \ \langle\text{sourcenode} \ a = \text{Entry}\rangle\)
\(\langle\text{sourcenode} \ a' = \text{Entry}\rangle \ \langle\text{targetnode} \ a' = \text{Exit}\rangle\)
\(\langle\text{intra-kind} \ (\text{kind} a)\rangle \ \langle\text{kind} a' = (\lambda s. \text{False})\rangle\)
show ?thesis by (auto dest:deterministic simp:intra-kind-def)
qed
from True \(\langle\ V \in \text{lift-Def} \ \text{Def Entry Exit} \ H L \ \text{src} \ e\\rangle \ \text{Entry-empty}\)
\(\langle e = (\text{Node} \ (\text{sourcenode} \ a), \ \text{kind} a, \ \text{Node} \ (\text{targetnode} \ a))\rangle\)
have \(V \in H\) by (fastforce elim:lift-Def-set.cases)
from True \(\langle e = (\text{Node} \ (\text{sourcenode} \ a), \ \text{kind} a, \ \text{Node} \ (\text{targetnode} \ a))\rangle\)
\(\langle\text{sourcenode} \ a \neq \text{Entry} \land \text{targetnode} \ a \neq \text{Exit}\rangle\)
have \(\forall V \in H. \ \forall V \in H. \ \forall V \ \text{in lift-Use Use Entry Exit} \ H L (\text{src} \ e)\)
by (fastforce intro:lift-Use-High)
with \(\forall V \in \text{lif-Use Use Entry Exit} \ H L (\text{src} \ e)\).
\(\text{state-val} \ s \ V = \text{state-val} \ s' \ V\)
\(\langle V \in H \rangle\)
have \(\text{state-val} \ s \ V = \text{state-val} \ s' \ V\)
by simp
with \(\langle e = (\text{Node} \ (\text{sourcenode} \ a), \ \text{kind} a, \ \text{Node} \ (\text{targetnode} \ a))\rangle\)
\[ Q \land \text{kind } a = (Q) \land \text{pred } (\text{kind } e) s \land \text{pred } (\text{kind } e) s' \]

\textbf{next case} False

\{ fix \ V' \ \textbf{assume} \ V' \in \text{Use } \text{(source node } a) \\
\quad \textbf{with} \ \langle e = (\text{Node } \text{(source node } a), \text{kind } a, \text{Node } \text{(target node } a)) \rangle \\
\quad \textbf{have} \ V' \in \text{lift-Use Use Entry Exit } H \ L \ (\text{src } e) \\
\quad \quad \textbf{by} (\text{fastforce intro:lift-Use-node}) \}

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

\text{state-val } s \ V = \text{state-val } s' \ V \\
\textbf{by fastforce}

\textbf{from} \ \langle \text{valid-edge } a \rangle \ \textbf{this} \ \langle \text{pred } (\text{kind } e) s \land \text{pred } (\text{kind } e) s' \rangle \\
\langle e = (\text{Node } \text{(source node } a), \text{kind } a, \text{Node } \text{(target node } a)) \rangle \\
\langle \text{intra-kind } (\text{kind } e) \rangle \\
\textbf{have} \ \forall V \in \text{Def } \text{(source node } a). \ \text{state-val } (\text{transfer } (\text{kind } a) s) V = \\
\text{state-val } (\text{transfer } (\text{kind } a) s') V \\
\textbf{by} \ -(\text{erule CFG-intra-edge-transfer-uses-only-Use,auto)}

\textbf{from} \ \langle V \in \text{lift-Def Def Entry Exit } H \ L \ (\text{src } e) \rangle \ \textbf{False} \\
\langle e = (\text{Node } \text{(source node } a), \text{kind } a, \text{Node } \text{(target node } a)) \rangle \\
\textbf{have} \ V \in \text{Def } \text{(source node } a) \textbf{ by} (\text{fastforce elim:lift-Def-set.cases}) \\
\textbf{with} \ \forall V \in \text{Def } \text{(source node } a). \ \text{state-val } (\text{transfer } (\text{kind } a) s) V = \\
\text{state-val } (\text{transfer } (\text{kind } a) s') V \\
\langle e = (\text{Node } \text{(source node } a), \text{kind } a, \text{Node } \text{(target node } a)) \rangle \\
\textbf{show} \ \text{thesis by simp} \quad \text{qed}

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

\textbf{from} \ \langle V \in \text{lift-Def Def Entry Exit } H \ L \ (\text{src } e) \rangle \\
\langle e = (\text{NewEntry}, (\lambda s. \text{True}) \land, \text{Node Entry}) \rangle \\
\textbf{have} \ False \ \textbf{by} (\text{fastforce elim:lift-Def-set.cases}) \\
\textbf{thus} \ \text{?case by simp} \quad \text{next}

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

\textbf{from} \ \langle V \in \text{lift-Def Def Entry Exit } H \ L \ (\text{src } e) \rangle \\
\langle e = (\text{Node Exit}, (\lambda s. \text{True}) \land, \text{NewExit}) \rangle \\
\textbf{have} \ False \\
\textbf{by} (\text{fastforce elim:lift-Def-set.cases intro!:Entry-not-eq-Exit simp:Exit-empty}) \\
\textbf{thus} \ \text{?case by simp} \quad \text{next}

\textbf{next case} (\text{lve-Entry-Exit-edge } e) \\
\textbf{thus} \ \text{?case by}(\text{cases } s) \text{ auto} \quad \text{qed}

\textbf{next}

\textbf{fix} \ a \ s \ s' \ \textbf{assume} \ \text{lift-valid-edge valid-edge source node } \text{target node } \text{kind } \text{Entry } \text{Exit } a \\
\text{and} \ \text{pred } (\text{kind } a) s \ \text{and} \ \text{snd } (\text{hd } s) = \text{snd } (\text{hd } s')
and \( \forall V \in \text{lift-Use} \) use Entry Exit H L (src a). state-val s V = state-val s' V
and length s = length s'
thus \( \text{pred} (\text{knd a}) s' \)
proof (induct rule: lift-valid-edge.induct)
case (lee-edge a e)
  from \( \langle e = (\text{Node} (\text{sourcenode a}), \text{kind a}, \text{Node} (\text{targetnode a})) \rangle \langle \text{pred} (\text{knd e}) s \rangle \)
    have \( \text{pred} (\text{knd a}) s \) and \( \text{src e} = \text{Node} (\text{sourcenode a}) \) by simp
  from \( \langle \text{src e} = \text{Node} (\text{sourcenode a}) \rangle \langle \forall V \in \text{lift-Use} \) use Entry Exit H L (src e). state-val s V = state-val s' V \rangle \)
    have \( \forall V \in \text{Use} (\text{sourcenode a}) \). state-val s V = state-val s' V \)
      by (auto dest: lift-Use-node)
    from \( \langle \text{valid-edge a} \rangle \langle \text{pred} (\text{knd a}) s \rangle \langle \text{snd} (\text{hd s}) = \text{snd} (\text{hd s'}) \rangle \)
      this \langle \text{length s} = \text{length s'} \rangle
    have \( \text{pred} (\text{knd a}) s' \) by (rule CFG-edge-Uses-pred-equal)
  with \( \langle e = (\text{Node} (\text{sourcenode a}), \text{kind a}, \text{Node} (\text{targetnode a})) \rangle \)
    show \( \langle \text{case by simp} \rangle \)
next
  case (lee-Entry-edge e)
  thus \( \langle \text{case by (cases s')} \rangle \) auto
next
  case (lee-Exit-edge e)
  thus \( \langle \text{case by (cases s')} \rangle \) auto
next
  case (lee-Entry-Exit-edge e)
  thus \( \langle \text{case by (cases s')} \rangle \) auto
  qed
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 fs and \( (p, \text{ins}, \text{outs}) \in \text{set procs} \)
thus \( \text{length fs} = \text{length ins} \)
proof (induct rule: lift-valid-edge.induct)
case (lee-edge a e)
  from \( \langle e = (\text{Node} (\text{sourcenode a}), \text{kind a}, \text{Node} (\text{targetnode a})) \rangle \langle \text{knd e} = Q := p fs' \rangle \)
    have \( \text{knd a} = Q := p fs \) by simp
  from \( \langle \text{valid-edge a} \rangle \langle \text{knd a} = Q := p fs' \rangle \langle (p, \text{ins}, \text{outs}) \in \text{set procs} \rangle \)
    show \( \langle \text{case by (rule CFG-call-edge-length) \rangle \) simp-all
  qed
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 := p fs and \( \text{knd a'} = Q := p' fs' \)
and lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a'
and src a = src a' and \( \text{pred} (\text{knd a}) s \) and \( \text{pred} (\text{knd a'}) s \)
from lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
\( \langle \text{knd a} = Q := p fs \rangle \langle \text{pred} (\text{knd a}) s \rangle \)
obtain x where \( a : a = (\text{Node} (\text{sourcenode x}), \text{kind x}, \text{Node} (\text{targetnode x})) \)
and valid-edge \( x \) and \( src \ a = Node \ (sourcenode \ x) \)
and kind \( x = Q: r \rightarrow pfs \) and \( pred \ (kind \ x) \ s \)
by (fastforce elim:lift-valid-edge.cases)
from \( \langle \langle \text{kind} \ a' = Q: r \rightarrow pfs', \ pred \ (\text{kind} \ a') \ s \rangle \)
obtain \( x' \) where \( a':a' = (\text{Node} \ (sourcenode \ x'), \text{kind} \ x', \text{Node} \ (targetnode \ x')) \)
and valid-edge \( x' \) and \( src \ a' = Node \ (sourcenode \ x') \)
and kind \( x' = Q: r \rightarrow pfs' \) and \( pred \ (\text{kind} \ x') \ s \)
by (fastforce elim:lift-valid-edge.cases)
from \( \langle \langle \text{src} \ a = Node \ (sourcenode \ x) \rangle, \langle \text{src} \ a' = Node \ (sourcenode \ x') \rangle \)
have sourcenode \( x = sourcenode \ x' \) by simp
from \( \langle \langle \text{kind} \ x = Q: r \rightarrow pfs \rangle, \text{valid-edge} \ (\langle \text{kind} \ x' = Q: r \rightarrow pfs' \rangle \)
\( \text{source} \ (x = \text{source} \ (x') \langle \text{pred} \ (\text{kind} \ x) \ s \rangle, \langle \text{pred} \ (\text{kind} \ x') \ s \rangle \)
have \( e = x' \) by (rule CFG-call-determ)
with \( a' \) show \( a = a' \) by simp
next
fix \( a \) Q \( r \) p fs i ins outs s s'
assume \( \langle \langle \text{lift}-\text{valid}-\text{edge} \ (\text{valid}-\text{edge} \ (\text{source} \ (\text{target} \ (a))) \\langle \text{kind} \ e = Q: r \rightarrow pfs) \\langle \text{pred} \ (\text{kind} \ e) \ s \rangle, \langle \text{pred} \ (\text{kind} \ e) \ s' \rangle \rangle \)
\( \langle \langle \text{kind} \ a = Q: r \rightarrow pfs \rangle, \langle \text{kind} \ a \rangle \rangle \langle \text{pred} \ (\text{kind} \ a) \ s \rangle \langle \text{pred} \ (\text{kind} \ a) \ s' \rangle \langle \text{source} \ e = \text{source} \ (\text{source} \ (a)) \rangle \)
by simp-all
from \( \forall \ V \in \text{lift}-\text{ParamUses} \ (\text{ParamUses} \ (\text{src} \ a)) \) ! i. state-val s V = state-val s' V
have \( e = \text{source} \ (\text{source} \ (a)) \)
\( \forall \ V \in (\text{ParamUses} \ (\text{source} \ (a))) \) ! i. state-val s V = state-val s' V by simp
with \( \langle \langle \text{valid}-\text{edge} \ a \rangle, \langle \text{kind} \ a = Q: r \rightarrow pfs \rangle, \langle \text{i < length ins} \rangle \rangle \langle \langle \text{pred} \ (\text{kind} \ a) \ s \rangle, \langle \text{pred} \ (\text{kind} \ a) \ s' \rangle \rangle \)
show \( \text{case by (rule CFG-call-edge-params)} \)
qed simp-all
next
fix a \( Q' \) p f' ins outs cf'f'
assume \( \langle \langle \text{lift}-\text{valid}-\text{edge} \ (\text{valid}-\text{edge} \ (\text{source} \ (\text{target} \ (a))) \\langle \text{kind} \ e = Q: r \rightarrow pfs) \\langle \text{pred} \ (\text{kind} \ e) \ s \rangle, \langle \text{pred} \ (\text{kind} \ e) \ s' \rangle \rangle \)
\( \langle \langle \text{kind} \ a = Q: r \rightarrow pfs' \rangle, \langle \text{kind} \ a \rangle \rangle \langle \text{pred} \ (\text{kind} \ a) \ s \rangle \langle \text{pred} \ (\text{kind} \ a) \ s' \rangle \)
\text{proof} (induct rule:lift-valid-edge.induct)
case (lev-edge a e)
from \( \langle e = (\text{Node} \ (\text{source} \ (a)), \text{kind} \ a, \text{Node} \ (\text{target} \ (a))) \rangle \), \( \langle \text{kind} \ e = Q: r \rightarrow pfs \rangle \)
\( \langle \text{pred} \ (\text{kind} \ e) \ s \rangle, \langle \text{pred} \ (\text{kind} \ e) \ s' \rangle \)
\( \langle \text{kind} \ a = Q: r \rightarrow pfs \rangle, \langle \text{kind} \ a \rangle \langle \text{pred} \ (\text{kind} \ a) \ s \rangle \langle \text{pred} \ (\text{kind} \ a) \ s' \rangle \langle \text{source} \ e = \text{source} \ (\text{source} \ (a)) \rangle \)
by simp-all
from \( \forall \ V \in \text{lift}-\text{ParamUses} \ (\text{ParamUses} \ (\text{src} \ a)) \) ! i. state-val s V = state-val s' V
have \( \langle \text{source} \ e = \text{source} \ (\text{source} \ (a)) \rangle \)
\( \forall \ V \in (\text{ParamUses} \ (\text{source} \ (a))) \) ! i. state-val s V = state-val s' V by simp
with \( \langle \langle \text{valid}-\text{edge} \ a \rangle, \langle \text{kind} \ a = Q: r \rightarrow pfs \rangle, \langle \text{i < length ins} \rangle \rangle \langle \langle \text{pred} \ (\text{kind} \ a) \ s \rangle, \langle \text{pred} \ (\text{kind} \ a) \ s' \rangle \rangle \)
\( \langle \text{case by (rule CFG-call-edge-params)} \rangle \)
qed simp-all
have kind a = Q′←pf′ and trg e = Node (targetnode a) by simp-all
from ⟨valid-edge a⟩ ⟨kind a = Q′←pf′⟩ ⟨(p, ins, outs) ∈ set procs⟩
have f′ cf cf′ = cf′(ParamDefs (targetnode a) \[=\] map cf outs)
  by (rule CFG-return-edge-fun)
with ⟨trg e = Node (targetnode a)⟩ show ?case by simp
qed simp-all
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′
  and intra-kind (knd a) and intra-kind (knd a′)
thus ∃ Q Q′. knd a = (Q) ∨ knd a′ = (Q′) ∨
  (∀ s. (Q s →¬ Q′ s) ∧ (Q′ s →¬ Q s))
proof (induct rule: lift-valid-edge.induct)
  case (lve-edge a e)
  from ⟨lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a′,
    ⟨valid-edge a⟩ ⟨(s, ins, outs) ∈ set procs⟩⟩
  ⟨src e = src a′⟩ ⟨trg e ≠ trg a′⟩ ⟨intra-kind (knd e)⟩ ⟨intra-kind (knd a′)⟩
  show ?case
  proof (induct rule: lift-valid-edge.induct)
    case lve-edge thus ?case by (auto dest: deterministic)
  qed auto
  qed (fastforce elim: lift-valid-edge.cases)+
qed

lemma lift-CFGExit:
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 src trg knd
  ⟨(lifvalid-edge valid-edge sourcenode targetnode kind Entry Exit) NewEntry
    (lif-get-proc get-proc Main)⟩
  ⟨(lif-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 CFG:CFG src trg knd
  lift-valid-edge valid-edge source node target node kind Entry Exit NewEntry
  lift-get-proc get-proc Main
  lift-get-return-edges get-return-edges valid-edge source node target node kind
  procs Main
  by (fastforce intro: lift-CFG wf pd)
show ?thesis
proof
  fix a assume lift-valid-edge valid-edge source node target node 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 source node target node kind Entry Exit a
  and knd a = Q←↩ p f and try a = NewExit
  thus False by (fastforce elim: lift-valid-edge. cases)
next
  show ∃ a. lift-valid-edge valid-edge source node target node kind Entry Exit a ∧
  src a = NewEntry ∧ try a = NewExit ∧ knd a = (λs. False)✓
  by (fastforce intro: lift-E Entry-Exit-edge)
qed
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 source node target node kind Entry Exit) NewEntry
  (lift-get-proc get-proc Main)
  (lift-get-return-edges get-return-edges valid-edge source node target node 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 source node target node 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
  interpret CFGExit:CFGExit src trg knd
  interpret CFGExit src trg knd
  interpret Postdomination src trg knd

  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 (m = sourcenode a ∨ m = targetnode a) }
have \(\text{CFG}, \text{CFG} . \text{valid-node} \ src \ trg\)
\(\text{(lift-valid-edge valid-edge} \ \text{sourcenode} \ \text{targetnode} \ \text{kind Entry Exit)} \ (\text{Node} \ m)\)

**proof**
- **assume** \(m = \text{sourcenode} a\)
- **show** \(?thesis\)

**proof**(cases \(m = \text{Entry}\))
- **case** True
  - **have** \(\text{lift-valid-edge valid-edge} \ \text{sourcenode} \ \text{targetnode} \ \text{kind Entry Exit}\)
    
    \(\text{(NewEntry,} (\lambda s. \ \text{True}) \ \sqrt, \text{Node Entry)} \ \text{by}(\text{fastforce intro:} \ \text{lev-Entry-edge})\)

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

- **next**
- **case** False

**next**
- **assume** \(m = \text{targetnode} a\)
- **show** \(?thesis\)

**proof**(cases \(m = \text{Exit}\))
- **case** True
  - **have** \(\text{lift-valid-edge valid-edge} \ \text{sourcenode} \ \text{targetnode} \ \text{kind Entry Exit}\)
    
    \(\text{(Node Exit,} (\lambda s. \ \text{True}) \ \sqrt, \text{NewExit)} \ \text{by}(\text{fastforce intro:} \ \text{lev-Edge})\)

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

- **next**
- **case** False

**next**
- **assume** \(m = \text{source} \ \text{node} a\)
- **show** \(?thesis\)

**proof**

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

\{ \text{fix} \ n \ \text{as} \ n' \ \text{cs} \ \text{m} \ \text{m'} \}
- **assume** \(\text{valid-path-aux} \ cs \ \text{as} \ \text{and} \ m - \text{as-}\rightarrow \ \text{m'} \ \text{and} \ \forall c \in \text{set} \ cs. \ \text{valid-edge} \ c\)

**hence** \(\exists \ cs' \ \text{cs} . \ \text{CFG} . \ \text{CFG} . \ \text{valid-path-aux} \ \text{kn}d\)

\(\text{(lift-get-return-edges get-return-edges} \ \text{sourcenode} \ \text{targetnode} \ \text{kind})\)
\(\cs' \ \text{cs} \wedge\)

**proof**(induct arbitrary:\(m\) rule:vpa-induct)
- **case** \(\text{vpa-empty} \ cs\)

**from** \(\text{m} - []^{\rightarrow*} \text{m'}\) **have** \[\text{simp}:m = \text{m'}\] **by** fastforce

58
from \( m \rightarrow \ast m' \) have valid-node \( m \) by (rule path-valid-node)
obtain \( cs' \) where \( cs' = \)
map (λc. (Node (sourcenode c), kind c, Node (targetnode c))) \( cs \) by simp

hence list-all2
(λc c'. c' = (Node (sourcenode c), kind c, Node (targetnode c))) \( cs \) \( cs' \)
by (simp add: list-all2-cone-all-nth)

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

next
case \( \text{vpa-intra } cs \ a \ as \)

note \( \text{IH} = \langle \bigwedge m. [ m \rightarrow \ast m'; \forall c \in \text{set } cs. \text{valid-edge } c; m \neq \text{Entry} \lor m' \neq \text{Exit} \] \( \Rightarrow \) \exists \( cs' \) es. CFG.valid-path-aux \( \text{kind} \)
(lift-get-return-edges get-return-edges valid-edge sourcenode
targetnode kind) \( cs' \) \( es \wedge \)
list-all2 \( (\lambda c c'. c' = (\text{Node (sourcenode } c), \text{kind } c, \text{Node (targetnode } c))) \) \( cs \)
\( cs' \wedge \text{CFG.path src trg (Node } m) \) \( es \) (Node \( m' \)).

from \( \langle m \rightarrow \ast a \rangle \) have \( m = \text{sourcenode } a \) and valid-edge \( a \)
and targetnode \( a \rightarrow \ast m' \) by (auto elim:path-split-Cons)
show ?case

proof (cases sourcenode \( a = \text{Entry} \wedge \text{targetnode } a = \text{Exit} \))
case True
with \( \langle m = \text{sourcenode } a \rangle \) \( (m \neq \text{Entry} \lor m' \neq \text{Exit}) \)
have \( m' \neq \text{Exit} \) by simp
from True have targetnode \( a = \text{Exit} \) by simp
with \( \langle \text{targetnode } a \rightarrow \ast m' \rangle \) have \( m' = \text{Exit} \)
by -(drule path-Exit-source, 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 \( \langle \text{valid-edge } a \rangle \)
have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit \( ?e \)
by (fastforce intro:lve-edge)
have targetnode \( a \neq \text{Entry} \)

proof
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 \( \text{IH} \langle \text{OF } \langle \text{targetnode } a \rightarrow \ast m' \rangle \rangle \langle \forall c \in \text{set } cs. \text{valid-edge } c \rangle \) this]
obtain \( cs' \) \( es \)
where valid-path: CFG.valid-path-aux \( \text{kind} \)
(lift-get-return-edges get-return-edges valid-edge sourcenode

59
targetnode kind) cs' es
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)) es (Node m') 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 − 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 ⟨valid-edge a⟩
have ∀c∈set (a # cs). valid-edge c by simp
let ᵇc = (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→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 ᵇc
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[OP ⟨targetnode a − as→* m'} ⟨c∈set (a # cs). valid-edge c⟩ 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
  ($\lambda c'. c' = (\text{Node} (\text{sourcenode} c), \text{kind} c, \text{Node} (\text{targetnode} c)))$ (a#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 cx csx where $cs' = cx#csx$
  and cx:cx = (\text{Node} (\text{sourcenode} a), \text{kind} a, \text{Node} (\text{targetnode} a))
  and list':list-all2
  ($\lambda c'. c' = (\text{Node} (\text{sourcenode} c), \text{kind} c, \text{Node} (\text{targetnode} c)))$ cs csx
by (fastforce simp:list-all2-Cons1)
from valid-path cx (cs' = cx#csx) (kind a = Q:$r\leftarrow p$) 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 = ($\exists m. \lbrack m - as\rightarrow m'; \forall c \in set \rbrack$. valid-edge c; m $\neq$ Entry $\lor$ m' $\neq$ Exit) 
  \Rightarrow \exists es. CFG.valid-path-aux knd
  (lift-get-return-edges get-return-edges valid-edge sourcenode
targetnode kind) cs' es \land
  list-all2 ($\lambda c'. c' = (\text{Node} (\text{sourcenode} c), \text{kind} c, \text{Node} (\text{targetnode} c)))$
  \lbrack cs' \land CFG.path src try \rbrack
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
  (Node m) es (Node m')
from m - a # as\rightarrow m' have m = sourcenode a and valid-edge a
  and targetnode a - as\rightarrow m' by (auto elim:path-split-Cons)
let ?e = (\text{Node} (\text{sourcenode} a), \text{kind} a, \text{Node} (\text{targetnode} a)) have targetnode a $\neq$ Exit
proof assume targetnode a = Exit
with (valid-edge a) (kind a = Q:$r\leftarrow p$) 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:lgv-edge)
have targetnode a $\neq$ 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 (targetnode a →∗ 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 trg
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(Node (targetnode a)) es (Node m’) by auto

from valid-path (kind a = Q←p)

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 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 ?case using (cs = []) by blast

next

case (vpa-ReturnCons cs a as Q p f c’ cs’)

note IH = (∀ m. [m →∗ m’; ∀ c∈set cs’. valid-edge c; m ≠ Entry ∨ m’ ≠ Exit] ⇒

∃ csx es. CFG.valid-path-aux knd
(lift-get-return-edges get-return-edges valid-edge sourcenode
(targetnode kind) csx es ∧
list-all2 (λ c c’. c’ = (Node (sourcenode c), kind c, Node (targetnode c)))
cs’ csx ∧ CFG.path src trg
(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

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←p)

have False by (rule Exit-no-return-target)

proof

assume targetnode a = Exit

with (valid-edge a) have False by (rule Exit-no-return-target)

next

case (vpa-ReturnCons cs a as Q p f c’ cs’)

note IH = (∀ m. [m →∗ m’; ∀ c∈set cs’. valid-edge c; m ≠ Entry ∨ m’ ≠ Exit] ⇒

∃ csx es. CFG.valid-path-aux knd
(lift-get-return-edges get-return-edges valid-edge sourcenode
(targetnode kind) csx es ∧
list-all2 (λ c c’. c’ = (Node (sourcenode c), kind c, Node (targetnode c)))
cs’ csx ∧ CFG.path src trg
(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

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) have False by (rule Exit-no-return-target)

proof

assume targetnode a = Entry

62
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 − as→* m' \rangle ∀ c∈set cs'. valid-edge c⟩ this]

obtain csx es

where valid-path:CFG.valid-path-aux kn
(lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind) csx es

and list:list-all2
(λ c. c'. c' = (Node (sourcenode c), kind c, Node (targetnode c))) cs' csx

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 (valid-edge c') (a ∈ get-return-edges c')

have ?e ∈ lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind
(Node (sourcenode c'), kind c', Node (targetnode c'))
by (fastforce intro:lift-get-return-edgesI)

with valid-path \langle kind a = Q←p'f⟩

have CFG.valid-path-aux kn
(lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind)
((Node (sourcenode c'), kind c', Node (targetnode c')) # csx) (?e # es)
by simp

moreover
from list \langle cs = c' # cs'⟩

have list-all2
(λ c. c'. c' = (Node (sourcenode c), kind c, Node (targetnode c))) cs
((Node (sourcenode c'), kind c', Node (targetnode c')) # csx)
by simp

moreover
from path \langle m = sourcenode a⟩

\langle 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 \langle kind a = Q←p'f⟩ by blast

qed

hence lift-valid-path:∀ m as m'. [m − as→* m'; m ≠ Entry ∨ m' ≠ Exit] 
⇒ ∃ es. CFG.CFG.valid-path' src try kn
(lift-valid-edge valid-valid-edge sourcenode targetnode kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind)
(Node m) es (Node m')
by (fastforce simp:vp-def valid-path-def CFGExit.vp-def CFGExit.valid-path-def)

show ?thesis

proof

fix n assume CFG.CFG.valid-node src try
(lift-valid-edge valid-node 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 \(\exists as. \ CFG.CFG.valid-path' \ src \ trg \ knd\)
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind)
NewEntry as n apply –

proof (erule disjE)+
assume n = NewEntry
hence 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 \[n\]
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 sourcenode targetnode kind Entry Exit
\(\langle\text{NewEntry}, (\lambda s. \text{False}), \text{NewExit}\rangle\)
by (fastforce intro: lve-Entry-Exit-edge)

hence CFG.CFG.path src trg
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
NewEntry \(\langle\text{NewEntry}, (\lambda s. \text{False}), \text{NewExit}\rangle\) NewExit
by (fastforce dest: CFGExit.path-edge)
with \(\langle n = \text{NewExit} \rangle\)

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 \(\langle\text{NewEntry}, (\lambda s. \text{False}), \text{NewExit}\rangle\) \(\text{n}\)
by (fastforce simp: CFGExit.vp-def CFGExit.valid-path-def)
thus \(?thesis\) by blast

next
assume \(\exists m. n = \text{Node} \ m \land \text{valid-node} \ m\)
then obtain \(\text{m}\) where \(n = \text{Node} \ m\) and valid-node \(m\) by blast
from \(\text{valid-node} \ m\)
show \(?thesis\)

proof (cases m rule: valid-node-cases)

case Entry
have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
\(\langle\text{NewEntry}, (\lambda s. \text{True}), \text{Node Entry}\rangle\)
by (fastforce intro: lve-Entry-Exit-edge)

with \(\langle m = \text{Entry} \rangle\)

have CFG.CFG.path src trg
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
NewEntry \(\langle\text{NewEntry}, (\lambda s. \text{True}), \text{Node Entry}\rangle\) \(\text{n}\)
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 \(\text{ax}\) where valid-edge \(\text{ax}\) and intra-kind \((\text{kind} \ \text{ax})\)
and inner-node \((\text{source} \ \text{ax})\)
and targetnode \(\text{ax} = \text{Exit}\) by (erule inner-node-Exit-edge)

hence lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
\(\langle\text{Node} \ (\text{source} \ \text{ax}), \text{kind} \ \text{ax}, \text{Node Exit}\rangle\)
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 knad
(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 lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
  (NewEntry,(λs. True) √ ,Node Entry) by(fastforce intro:lve-Entry-edge)

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

hence slp-edge':CFG.CFG.same-level-path' src try knad
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind)
(Node Entry) [(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→ ∗ sourcenode ax
by(fastforce dest:Entry-path)
with ‹sourcenode ax ≠ Exit›

have ∃ es, CFG.CFG.valid-path' src try knad
(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 knad
(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
with slp-edge have CFG.CFG.valid-path' src try knad
(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.vp-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-up-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:lev-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 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 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 −→ ∗ m
by(fastforce dest: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: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-up-Append)

with (n = Node m) show ?thesis by simp blast
qed

qed
next

fix n assume CFG, CFG.valid-node src trg
(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 trg knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind)

n as NewExit apply —

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 trg
(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 trg knd
(lift-valid-edge 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 vp-def CFGExit.valid-path-def)

thus ?thesis by blast

next

assume n = NewExit

hence 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)

n [] NewExit

by (fastforce intro: CFGExit.empty-path
simp: CFGExit vp-def 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 trg
(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
henceCFG.path src try
(lift-valid-edge valid-edge source node target node kind Entry Exit)
(Node Exit) ⟨(Node Exit, (λs. True) ∨, New Exit)⟩ by (fastforce intro: λ-see-Exit-edge)
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) ∨, New Exit)⟩ New Exit
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) ⟨([Node Entry, kind ax, Node (targetnode ax)])@es⟩ (Node
Exit)  
by (rule CFGExit.slp-vp-Append)
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)
  ((([(Node Entry, kind ax, Node (target node ax)]) @ es] @
  [(Node Exit, (λ s. True), NewExit)]) NewExit
by \text{‐}(\text{rule} CFGExit.vp-slp-Append)
with \langle m = \text{Entry}\rangle \langle n = \text{Node} m \rangle \text{show} ?\text{thesis} by simp blast
next
  case Exit
  have lift-valid-edge valid-edge source node target node kind Entry Exit
    (Node Exit, (λ s. True)) √ NewExit by (fastforce intro: lift-valid-edge)
with \langle m = \text{Exit}\rangle \langle n = \text{Node} m \rangle have CFG.CFG.path src trg
  (lift-valid-edge valid-edge source node target node kind Entry Exit)
  n [(Node Exit, (λ s. True)) √ NewExit] New Exit
by (fastforce intro: CFGExit.Cons-path CFGExit.empty-path
  simp:CFGExit.valid-node-def)
thus ?\text{thesis} by (fastforce simp: CFGExit.vp-def CFGExit.valid-path-def)
next
  case inner
  have lift-valid-edge valid-edge source node target node kind Entry Exit
    (Node Exit, (λ s. True)) √ New Exit by (fastforce intro: lift-valid-edge)
hence CFG.path src trg
  (lift-valid-edge valid-edge source node target node kind Entry Exit)
  (Node Exit) [(Node Exit, (λ s. True)) √ New Exit] New Exit
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, (λ s. True)) √ New Exit] New Exit
by (fastforce simp: CFGExit.slp-def CFGExit.same-level-path-def)
from \langle valid-node m \rangle obtain as where m \text{−}as→ √* Exit
by (fastforce dest: Exit-path)
with \langle inner-node m \rangle
  have \exists 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 m) es (Node Exit)
  by (fastforce intro: lift-valid-path simp: inner-node-def)
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 m) 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 sourcenode
targetnode kind) (Node m) \{ es@[\{ Node Exit, \lambda s. True \} \cup, NewExit] \} NewExit
by \text{-(rule CFGExit.vp-slp-Append)}
with \langle n = Node m \rangle \text{ show } ?\text{thesis by simp blast}
qed
qed
next
fix n n' assume method-exit1:CFGExit.CFGExit.method-exit src knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewExit n
and method-exit2:CFGExit.CFGExit.method-exit src knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewExit n'
and lift-eq:lift-get-proc get-proc Main n = lift-get-proc get-proc Main n'
from method-exit1 show n = n'
proof\text{(rule CFGExit.method-exit-cases)}
assume n = NewExit
from method-exit2 show ?thesis
proof\text{(rule CFGExit.method-exit-cases)}
assume n' = NewExit
with \langle n = NewExit \rangle \text{ show } ?\text{thesis by simp}
next
fix a Q f p assume n' = src a
and lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and knd a = Q \leftarrow p f
hence lift-get-proc get-proc Main (src a) = p
by \text{-(rule CFGExit.get-proc-return)}
with CFGExit.get-proc-Exit lift-eq \langle n' = src a \rangle \langle n = NewExit \rangle
have p = Main by simp
with \langle knd a = Q \leftarrow p f \rangle \text{ have } knd a = Q \leftarrow Main f \text{ by simp}
with \langle lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a \rangle
have False by\text{(rule CFGExit.Main-no-return-source)}
thus ?\text{thesis by simp}
qed
next
fix a Q f p assume n = src a
and lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and knd a = Q \leftarrow p f
then obtain x where valid-edge x and src a = Node (sourcenode x)
and kind x = Q \leftarrow p f
by\text{(fastforce elim:lift-valid-edge.cases)}
hence method-exit (sourcenode x) by\text{(fastforce simp:method-exit-def)}
from method-exit2 show ?thesis
proof\text{(rule CFGExit.method-exit-cases)}
assume n' = NewExit
from \langle lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a \rangle
\langle knd a = Q \leftarrow p f \rangle 
have lift-get-proc get-proc Main (src a) = p
by \textbf{\texttt{-(\textup{rule CFGExit.get-proc-return})}} \\
\textbf{\texttt{with CFGExit.get-proc-Exit lift-eq \langle n = \text{src} a \rangle \langle n' = \text{NewExit} \rangle}} \\
\textbf{\texttt{have \textit{p = Main} by simp}} \\
\textbf{\texttt{with \langle \text{kind} a = \text{Q'} \rightarrow p \prime \rangle \textbf{\texttt{have \textit{kind} a = \text{Q'} \rightarrow \text{Main}f by simp}}}} \\
\textbf{\texttt{with \langle \text{lift-valid-edge valid-edge source-node target-node kind Entry Exit} a \rangle \textbf{\texttt{have \textit{False by(\textup{rule CFGExit.Main-no-return-source})}}}}}} \\
\textbf{\texttt{thus \textbf{\texttt{?thesis by simp}}}} \\
\textbf{\texttt{next}} \\
\textbf{\texttt{fix \textit{a'} Q' f' p'}} \\
\textbf{\texttt{assume \textit{n'} = \text{src} a'}} \\
\and \textbf{\texttt{\textit{kind} a' = \text{Q'} \rightarrow p'f'}} \\
\textbf{\texttt{then obtain \textit{x'} where valid-edge \textit{x'} and \textit{src} a' = \text{Node (source-node \textit{x'})}}}} \\
\and \textbf{\texttt{\textit{kind} x' = \text{Q'} \rightarrow p'f' \textit{by(\textup{fastforce elim:lift-valid-edge.cases})}}} \\
\textbf{\texttt{hence method-exit (source-node \textit{x'}) by(\textup{fastforce simp:method-exit-def})}} \\
\textbf{\texttt{with \langle \text{method-exit (source-node \textit{x'})} \rangle \textbf{\texttt{lift-eq \langle \text{src} a \rangle \langle \text{src} a' \rangle}}}} \\
\langle \text{src a = Node (source-node \textit{x'})} \rangle \langle \text{src a' = Node (source-node \textit{x'})} \rangle \\
\textbf{\texttt{have \textit{source-node \textit{x} = source-node \textit{x'} by(\textup{fastforce intro:method-exit-unique})}}}} \\
\textbf{\texttt{with \langle \text{src a = Node (source-node \textit{x})} \rangle \langle \text{src a' = Node (source-node \textit{x'})} \rangle}} \\
\langle \text{\textit{n} = \text{src} a} \rangle \langle \text{n' = \text{src} a'} \rangle \\
\textbf{\texttt{show \textbf{\texttt{?thesis by simp}}}} \\
\textbf{\texttt{qed}}} \\
\textbf{\texttt{qed}}} \\
\textbf{\texttt{qed}} \\
\textbf{\texttt{lemma lift-\textit{SDG}:}} \\
\textbf{\texttt{assumes SDG:SDG source-node target-node kind valid-edge Entry get-proc get-return-edges procs Main Exit Def Use ParamDefs ParamUses}} \\
\and \textbf{\texttt{inner:CFGExit.inner-node source-node target-node valid-edge Entry Exit nx}} \\
\textbf{\texttt{shows SDG src try \textit{knf}}} \\
\langle \text{lift-valid-edge valid-edge source-node target-node kind Entry Exit} \rangle \textbf{\texttt{NewExit}} \\
\langle \text{lift-get-proc get-proc Main} \rangle \\
\langle \text{lift-get-return-edges get-return-edges valid-edge source-node target-node kind} \rangle \\
\textbf{\texttt{procs Main NewExit (lift-Def Def Entry Exit H L) (lift-Use Entry Exit H L) (lift-ParamDefs ParamDefs) (lift-ParamUses ParamUses)}}}} \\
\textbf{\texttt{proof}} \\
\textbf{\texttt{interpret SDG source-node target-node kind valid-edge Entry get-proc get-return-edges procs Main Exit Def Use ParamDefs ParamUses}}  \\
\textbf{\texttt{by(\textup{rule SDG})}} \\
\textbf{\texttt{have \textbf{\texttt{wf:CFGExit-wf source-node target-node kind valid-edge Entry get-proc get-return-edges procs Main Exit Def Use ParamDefs ParamUses}}}} \\
\textbf{\texttt{by(\textup{unfold-locales})}} \\
\textbf{\texttt{have \textbf{\texttt{pd:Postdomination source-node target-node kind valid-edge Entry get-proc get-return-edges procs Main Exit}}}} \\
\textbf{\texttt{by(\textup{unfold-locales})}} \\
\textbf{\texttt{interpret \textbf{\texttt{wf':CFGExit-wf src try \textit{knf}}}}}}
3.2.3 Low-deterministic security via the lifted graph


lemma Lift-NonInterferenceGraph:
fixes valid-edge and sourcenode and targetnode and kind and Entry and Exit
and get-proc and get-return-edges and procs and Main
and Def and Use and ParamDefs and ParamUses and H and L
defines lve:⊥e ≡ lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
and lget-proc:lget-proc ≡ lift-get-proc get-proc Main
and lget-return-edges:lget-return-edges ≡ lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind
and lDef:lDef ≡ lift-Def Def Entry Exit H L
and lUse:lUse ≡ lift-Use Use Entry Exit H L
and lParamDefs:lParamDefs ≡ lift-ParamDefs ParamDefs
and lParamUses:lParamUses ≡ lift-ParamUses ParamUses
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 valid-edge Entry Exit nx
and H ∩ L = {} and H ∪ L = UNIV
shows NonInterferenceInterGraph src trg knd lve NewEntry lget-proc
lget-return-edges procs Main NewExit lDef lUse lParamDefs lParamUses H L
(Node Entry) (Node Exit)

proof –
interpret SDG sourcenode targetnode kind valid-edge Entry get-proc
get-return-edges procs Main Exit Def Use ParamDefs ParamUses
by(rule SDG)
interpret SDG’: SDG src trg knd lve NewEntry lget-proc lget-return-edges
procs Main NewExit lDef lUse lParamDefs lParamUses
by(fastforce intro:lift-SDG SDG inner:simp:lve
lget-proc lget-return-edges lDef
lUse lParamDefs lParamUses)

show ?thesis

proof

fix a assume lve a and src a = NewEntry
thus try a = NewExit ∨ try a = Node Entry
by(fastforce elim:lift-valid-edge.cases simp:lve)
\[ \exists a. \: \text{lve} \ a \land \text{src} \ a = \text{NewEntry} \land \text{trg} \ a = \text{Node Entry} \land \text{knd} \ a = (\lambda s. \text{True}) \]

by (fastforce intro: lve-Entry-edge simp: lve)

next

show \[ \exists a. \: \text{lve} \ a \land \text{src} \ a = \text{NewEntry} \land \text{trg} \ a = \text{Node Entry} \land \text{knd} \ a = (\lambda s. \text{True}) \]

by (fastforce intro: lve-Entry-edge simp: lve)

next

fix \( a \) assume \( \text{lve} \ a \) and \( \text{trg} \ a = \text{Node Entry} \)
from \( \langle \text{lve} \ a \rangle \)
have \( \text{lift-valid-edge} \ \text{valid-edge} \ \text{source-node} \ \text{target-node} \ \text{kind} \ \text{Entry Exit} \ a \)
by (simp add: lve)
from this \( \langle \text{trg} \ a = \text{Node Entry} \rangle \)
show \( \text{src} \ a = \text{NewEntry} \)
proof (induct rule: lift-valid-edge.induct)
\[
\begin{cases}
\text{case } (\text{lve-edge } a \ e) \\
\text{from } \langle e = (\text{Node (source-node } a), \text{kind} \ a, \text{Node (target-node } a)) \rangle \\
\quad \langle \text{trg} \ e = \text{Node Entry} \rangle \\
\quad \text{have target-node } a = \text{Entry by simp} \\
\quad \text{with } \langle \text{valid-edge } a \rangle \ \text{have False by (rule Entry-target) } \\
\quad \text{thus } \text{case by simp} \\
\end{cases}
\]
qed simp-all

next

fix \( a \) assume \( \text{lve} \ a \) and \( \text{trg} \ a = \text{NewExit} \)

thus \( \text{src} \ a = \text{NewExit} \lor \text{src} \ a = \text{Node Exit} \)
by (fastforce elim: lift-valid-edge.cases simp: lve)

next

show \( \exists a. \: \text{lve} \ a \land \text{src} \ a = \text{Node Exit} \land \text{trg} \ a = \text{NewExit} \land \text{knd} \ a = (\lambda s. \text{True}) \)

by (fastforce intro: lve-Exit-edge simp: lve)

next

fix \( a \) assume \( \text{lve} \ a \) and \( \text{src} \ a = \text{Node Exit} \)
from \( \langle \text{lve} \ a \rangle \)

have \( \text{lift-valid-edge} \ \text{valid-edge} \ \text{source-node} \ \text{target-node} \ \text{kind} \ \text{Entry Exit} \ a \)
by (simp add: lve)
from this \( \langle \text{src} \ a = \text{Node Exit} \rangle \)

show \( \text{trg} \ a = \text{NewExit} \)
proof (induct rule: lift-valid-edge.induct)
\[
\begin{cases}
\text{case } (\text{lve-edge } a \ e) \\
\text{from } \langle e = (\text{Node (source-node } a), \text{kind} \ a, \text{Node (target-node } a)) \rangle \\
\quad \langle \text{src} \ e = \text{Node Exit} \rangle \\
\quad \text{have source-node } a = \text{Exit by simp} \\
\quad \text{with } \langle \text{valid-edge } a \rangle \ \text{have False by (rule Exit-source) } \\
\quad \text{thus } \text{case by simp} \\
\end{cases}
\]
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 \( \{ \} \cap L = \{ \} \) show \( H \cap L = \{ \} \).
next
from \( \{ \} \cup L = UNIV \) show \( H \cup L = UNIV \).
qed
qed

end

References


