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 fulfill 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/StaticInter/FundamentalProperty
begin
2.1 Assumptions of this Approach

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

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

```
locale NonInterferenceInterGraph =
  SDG sourcenode targetnode kind valid-edge Entry
  get-proc get-return-edges procs Main Exit Def Use ParamDefs ParamUses
for sourcenode :: 'edge ⇒ 'node and targetnode :: 'edge ⇒ 'node
and kind :: 'edge ⇒ (‘var,’val,’ret,’pname) edge-kind
and valid-edge :: 'edge ⇒ bool
and Entry :: (‘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 :: (‘High-‘)
fixes Low :: (‘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-Exit-or-Low: [valid-edge a; targetnode a = (‘Exit-)]
  ⇒ sourcenode a = (‘Exit-) ∨ 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-)
```

2
and DefHigh: Def (-High-) = H 
and UseHigh: Use (-High-) = H 
and UseLow: Use (-Low-) = L 
and HighLowDistinct: H ∩ L = {} 
and HighLowUNIV: H ∪ L = UNIV

begin

lemma Low-neq-Exit: assumes L ≠ {} shows (-Low-) ≠ (-Exit-)
  proof
  assume (-Low-) = (-Exit-)
  have Use (-Exit-) = {} by fastforce
  with UseLow ⟨L ≠ {}⟩ ⟨(-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 ⟨valid-edge a⟩ ⟨intra-kind (kind a)⟩
  have get-proc (sourcenode a) = get-proc (targetnode a) by (rule get-proc-intra)
  with ⟨sourcenode a = (-Low-)⟩ ⟨targetnode a = (-Exit-)⟩ 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 ⟨valid-edge a⟩ ⟨intra-kind (kind a)⟩
  have get-proc (sourcenode a) = get-proc (targetnode a) by (rule get-proc-intra)
  with ⟨sourcenode a = (-Entry-)⟩ ⟨targetnode a = (-High-)⟩ get-proc-Entry
  show ?thesis by simp
  qed
lemma Entry-path-High-path:
assumes (-Entry-) \( \rightarrow as \rightarrow n \) and inner-node \( n \)
obtains \( a' \) as' where \( as = a' \# as' \) and (-High-) \( \rightarrow as' \rightarrow n \) and kind \( a' = (\lambda s. \text{True}) \)
proof (atomize-elim)
from (-Entry-) \( \rightarrow as \rightarrow n \) (inner-node \( n \))
show \( \exists a' as'. as = a' \# as' \land (-\text{High}) \rightarrow as' \rightarrow n \land \text{kind} \ a' = (\lambda s. \text{True}) \)
proof (induct \( n' \equiv (-\text{Entry}) \) as \( n \) rule: \( \text{path-induct} \))
case (Cons-path \( n'' \) as \( n' \) \( a \))
  from \( n'' \rightarrow as \rightarrow n' \) (inner-node \( n' \) \( h \)) have \( n'' \neq (-\text{Exit}) \)
  by (fastforce simp: inner-node-def)
with valid-edge \( a \) (sourcenode \( a = (-\text{Entry}) \) (targetnode \( a = n'' \))
  have \( n'' = (-\text{High}) \) by -(drule Entry-edge-Exit-or-High, auto)
from High-target-Entry-edge
obtain \( a' \) where valid-edge \( a' \) and sourcenode \( a' = (-\text{Entry}) \)
  and targetnode \( a' = (-\text{High}) \) and kind \( a' = (\lambda s. \text{True}) \)
by blast
with valid-edge \( a \) (sourcenode \( a = (-\text{Entry}) \) (targetnode \( a = n'' \))
  \( n'' = (-\text{High}) \) \( \langle \text{kind} \ a' = (\lambda s. \text{True}) \rangle \) show \( \text{case} \ \text{by} \ \text{blast} \)
qed fastforce

lemma Exit-path-Low-path:
assumes \( n \rightarrow as \rightarrow n \) (\( \text{-Exit} \)) and inner-node \( n \)
obtains \( a' \) as' where \( as = as' \# [a'] \) and \( n \rightarrow as' \rightarrow n \) (\( \text{-Low} \))
and kind \( a' = (\lambda s. \text{True}) \)
proof (atomize-elim)
from \( n \rightarrow as \rightarrow n \) (\( \text{-Exit} \))
show \( \exists a' as'. as = as' \# [a'] \land n \rightarrow as' \rightarrow n \land \text{kind} \ a' = (\lambda s. \text{True}) \)
proof (induct \( n \) as \( rule: rev-induct \))
case Nil
  with inner-node \( n \) show \( \text{case} \ \text{by} \ \text{fastforce} \)
next
case \( \text{snoc} \ a' \ as' \)
  from \( n \rightarrow as' \# [a'] \rightarrow n \) (\( \text{-Exit} \))
have \( n \rightarrow as' \rightarrow n \) sourcenode \( a' \) and valid-edge \( a' \) and targetnode \( a' = (-\text{Exit}) \)
by (auto elim: path-split-snoc)
  \{ assume sourcenode \( a' = (-\text{Entry}) \)
    with \( n \rightarrow as' \rightarrow n \) sourcenode \( a' \) have \( n = (-\text{Entry}) \)
    by (blast intro!: path-Entry-target)
    with inner-node \( n \) have \( \text{False} \) by (simp add: inner-node-def) \} 
with valid-edge \( a' \) (targetnode \( a' = (-\text{Exit}) \) have sourcenode \( a' = (-\text{Low}) \)
by (blast dest!: Edge-edge Exiting-or-Low)
from Low-source-Exit-edge
obtain \( ax \) where valid-edge \( ax \) and sourcenode \( ax = (-\text{Low}) \)
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

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

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

2.2 Low Equivalence

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

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

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

lemma relevant-vars-Entry:
assumes V ∈ rv S (CFG-node (-Entry-)) and (-High-) /∈ ⌊HRB-slice S⌋
CFG
shows V ∈ L
proof
− from ⟨V ∈ rv S (CFG-node (-Entry-)): obtain as n' where (-Entry-) −as→∗ parent-node n'
and n' ∈ HRB-slice S and V ∈ UseSDG n'
and ∀ n'', valid-SDG-node n'' ∧ parent-node n'' ∈ set (sourcenodes as)
−→ V /∈ DefSDG n'' by (fastforce elim:rvE)
from (-Entry-) −as→∗ parent-node n' have valid-node (parent-node n')
by (fastforce intro:path-valid-node simp:intra-path-def)
thus ?thesis
proof (cases parent-node n' rule:valid-node-cases)
case Entry
with ⟨V ∈ UseSDG n'⟩ have False
by -(drule SDG-Use-parent-Use,simp add:Entry-empty)
thus ?thesis by simp
next
case Exit
  with \( V \in \text{Use}_{SDG} n \) have False
  by \((\text{drule SDG-Use-parent-Use,simp add:Exit-empty})\)
thus ?thesis by simp

next
case inner
with \( \langle \text{-Entry-} \rangle \rightarrow \ast \) parent-node \( n \) \( \langle \text{-Entry-} \rangle \rightarrow \ast \) parent-node \( n \)
by (fastforce elim:Entry-path-High-path simp:intra-path-def)
from \( \langle \text{-Entry-} \rangle \rightarrow \ast \) parent-node \( n \) \( \langle \text{-Entry-} \rangle \rightarrow \ast \) parent-node \( n \)
have sourcenode \( a \) \( \langle \text{-Entry-} \rangle \rightarrow \ast \) parent-node \( n \)
  by (fastforce elim: path-cases simp: intra-path-def)
show ?thesis
proof (cases as \( \notin \) [])
case True
with \( \langle \text{-High-} \rangle \rightarrow \ast \) parent-node \( n \) \( \langle \text{-High-} \rangle \rightarrow \ast \) parent-node \( n \)
by (fastforce simp: intra-path-def)
with \( as \in \text{HRB-slice} \ S \) \( \text{CFG} \)
have False
thus ?thesis by simp

next
case False
with \( \langle \text{-High-} \rangle \rightarrow \ast \) parent-node \( n \) \( \langle \text{-High-} \rangle \rightarrow \ast \) parent-node \( n \)
by (fastforce intro:path-source-node simp: intra-path-def)
from False have \( \langle \text{-High-} \rangle \rightarrow \ast \) parent-node \( n \) \( \langle \text{-High-} \rangle \rightarrow \ast \) parent-node \( n \)
  by (fastforce intro:hd-in-set simp: sourcenodes-def)
with \( as \in \text{set} \text{ (sourcenodes as) } \)
  by (simp add: sourcenodes-def)
from \( (\text{hd} \text{ (sourcenodes as) } = \langle \text{-High-} \rangle) \)
have valid-node \( (\text{hd} \text{ (sourcenodes as) }) \) \( \text{by simp} \)
have valid-SDG-node \( \langle \text{CFG-node} \langle \text{-High-} \rangle \rangle \) \( \text{by simp} \)
with \( \langle \text{hd} \text{ (sourcenodes as) } = \langle \text{-High-} \rangle \rangle \)
  \( \langle \text{hd} \text{ (sourcenodes as) } = \langle \text{-High-} \rangle \rangle \)
  \( \langle \forall \text{ as'. valid-SDG-node as'} \text{ parent-node as'} \text{ set} \text{ (sourcenodes as) } \)
  \( \rightarrow V \notin \text{Def}_{SDG} n' \)
  \( \text{have V \notin \text{Def} (\langle -High- \rangle) } \)
  \( \text{by (fastforce dest: CFG-Def-SDG-Def [OF \ (valid-node \ (\text{hd} \text{ (sourcenodes as) }) \ )]) } \)
  \( \text{hence V \notin H \ by (simp add: DefHigh) } \)
  \( \text{thus ?thesis by (rule not-High-Low) } \)
qed

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

proof

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

with \((-\text{High}) \notin [\text{HRB-slice } S]_{\text{CFG}} \) have \( V \in L \) by -(rule relevant-vars-Entry)

with \((s \approx_{L} s')\) show \( \text{hd} \ s \ V = \text{hd} \ s' \ V \) by(simp add:lowEquivalence-def)

qed

2.3 The Correctness Proofs

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

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

lemma \( \text{slpa-rv-Low-Use-Low} \):

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

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

proof(induct arbitrary:\( m \) \( \text{ as } s' \) \( s' \) \( \text{ rule:slpa-induct} \))

case (\( \text{slpa-empty } cs \))

from \((m \text{ \(-\text{as}\gets\text{*-}(\text{-Low})\)})) have \( m = (-\text{Low}) \) by fastforce

from \((m \text{ \(-\text{as}\gets\text{*-}(\text{-Low})\)})) have \( \text{valid-node } m \)
by(rule path-valid-node)+

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

moreover

from \((\text{valid-node } m) \ (m = (-\text{Low})): \text{ have } (-\text{Low}) \ \text{-}\to\!\to_1 \ (-\text{Low}) \)
by(fastforce intro:empty-path simp:intra-path-def)

moreover

from \((\text{valid-node } m) \ (m = (-\text{Low})) \ (\text{CFG-node } (-\text{Low}) \in S) \)
have \( \text{CFG-node } (-\text{Low}) \in \text{HRB-slice } S \)
by(fastforce intro:HRB-slice-refl)

ultimately have \( V \in \text{rv } S \ (\text{CFG-node } m) \)
using \((m = (-\text{Low}))\)
by(auto intro!:rdl CFG-Use-SDG-Use simp:sourcenodes-def) \}

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

show \( ?\text{thesis} \)

proof(cases \( L = \{\} \))

case True with \text{UseLow} show \( ?\text{thesis} \) 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)
qed

next
case (slpa-intra cs a as)
note IH = ⟨∀ m as′ s′. [upd-cs cs as as′]; same-level-path-aux cs as′; ∀ a ∈ set cs. valid-edge a; m − as′→∗ (Low-); m − as′→∗ (Low-); ∀ i < length cs. ∀ V ∈ rv S (CFG-node (sourcenode (cs ! i)))
fst (s ! Suc i) V = fst (s′ ! Suc i) V;
∀ i < Suc (length cs), snd (s ! i) = snd (s′ ! i);
∀ V ∈ rv S (CFG-node m). state-val s V = state-val s′ V;
preds (slice-kinds S as) s; preds (slice-kinds S as′) s′;
length s = Suc (length cs); length s′ = Suc (length cs)]
⇒ ∀ 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′ = (as, True) by blast
from ⟨valid-edge a (sourcenode a = m)⟩ ⟨m = (Low-)⟩
⟨targetnode a = (Exit-)⟩ ⟨valid-edge a′ (sourcenode a′ = (Low-))
⟨targetnode a′ = (Exit-)⟩
have a = a′ by (fastforce dest:edge-det)
with \( \langle \text{kind } a' = (\lambda s. \text{True}) \rangle \) have kind \( a = (\lambda s. \text{True}) \) by simp

with \( \langle \text{targetnode } a = (-\text{Exit}) \rangle \) (targetnode \( a \to \ldots \to (-\text{Low}) \))

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

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

thus ?thesis by simp

next

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

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

and \( \text{targetnode } ax \to \ldots \to (-\text{Low}) \) by \((\text{auto elim: path-split-Cons})\)

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

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

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

obtain \( cf' \) \( \langle \text{fs'} where } \langle \text{simp} \rangle s' = cf' \# cf s' \)

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

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

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

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

have False

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

case True

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

thus ?thesis by simp

next

case False

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

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

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

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

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

thus ?thesis by simp

qed

thus ?thesis by simp

next

case \((\text{Return } Q \ p \ f)\)
from (valid-edge ax) (kind ax = Q→p) (valid-edge a) (intra-kind (kind a)) (sourcenode a = m) (sourcenode ax = m) have False by -(erule return-edges-only,auto simp:intra-kind-def) thus ?thesis by simp qed simp
with (same-level-path-aux cs as x) (as' = ax#asx) have same-level-path-aux cs ax by (fastforce simp:intra-kind-def)
show ?thesis
proof (cases targetnode a = targetnode ax)
case True
with (valid-edge a) (valid-edge ax) (sourcenode a = m) (sourcenode ax = m)
have a = ax by (fastforce intro:edge-def)
with (valid-edge a) (intra-kind (kind a)) (sourcenode a = m) (valid-edge ax) (sourcenode ax = 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 (targetnode a)), state-val (transfer (slice-kind S a) s) V = state-val (transfer (slice-kind S a) s') V by -(erule 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 (targetnode ax = ax#→*) (a = ax) have targetnode a = ax#→* (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 sourcenode a ∈ [HRB-slice S]CFG) apply auto
  apply (fastforce dest:transfer-intra-kind-in-slice simp:intra-kind-def)
  apply (auto simp:intra-kind-def)
  apply (erule slice-kind- upd) apply auto
by (erule kind-Predicate-notin-slice-slice-kind-Predicate) auto
from (valid-edge a) (intra-kind (kind a)) obtain cfx'
  where cfx':transfer (slice-kind S a) s' = cfx'#cfs' ∧ snd cfx' = snd cf'
  apply (cases cf')
  apply (cases sourcenode a ∈ [HRB-slice S]CFG) apply auto
  apply (fastforce dest:transfer-intra-kind-in-slice simp:intra-kind-def)
  apply (auto simp:intra-kind-def)
  apply (erule slice-kind- upd) apply auto
by (erule kind-Predicate-notin-slice-slice-kind-Predicate) auto
with cfx ∀ i < Suc (length cs), snd (s[i]) = snd (s’[i]) have snds:∀ i<Suc(length cs).
  snd (transfer (slice-kind S a) $i) =
  snd (transfer (slice-kind S a) $’[i])
  by auto(case-tac i,auto)
from rvs cfx cfx' have rvs:∀ i<length cs.
  ∀ V∈rv S (CFG-node (sourcenode (cs ! i))).
\[\begin{align*}
\text{fst} (\text{transfer} (\text{slice-kind} S a) s) & ! \text{Suc} i \quad V = \\
\text{fst} (\text{transfer} (\text{slice-kind} S a) s') & ! \text{Suc} i \quad V \\
\text{by fastforce}
\end{align*}\]

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

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

moreover

from \((\text{preds} (\text{slice-kinds} S \text{as}') s')\)  
\(< ax \quad \text{as}' = ax \# \text{as} \quad a = ax)\)

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

moreover

from \((\text{valid-edge} a \quad \text{intra-kind} (\text{kind} a)\)

have \((\text{length} (\text{transfer} (\text{slice-kind} S a) s) = \text{length} s\)  
\by\((\text{cases source-node} a \in [\text{HRB-slice \text{S}}]_{\text{CFG}})\)

\((\text{auto dest: slice-intra-kind-in-slice slice-kind-Upd}\)

\elim:\text{kind-Predicate-notin-slice-slice-kind-Predicate simp:intra-kind-def}\)

with \((\text{length} s = \text{Suc} (\text{length} cs)\)

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

moreover

from \((a = ax) \quad (\text{valid-edge} a \quad \text{intra-kind} (\text{kind} a)\)

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

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

\((\text{auto dest: slice-intra-kind-in-slice slice-kind-Upd}\)

\elim:\text{kind-Predicate-notin-slice-slice-kind-Predicate simp:intra-kind-def}\)

with \((\text{length} s' = \text{Suc} (\text{length} cs)\)

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

moreover

from \([H] \text{OF} \quad (\text{upd-cs \text{cs} \text{as} = []}) \quad (\text{same-level-path-aux cs as} \text{ax})\)

\((\forall c c \text{set} \text{cs} \text{. valid-edge c} \quad \text{target-node} a \text{ as} \rightarrow* (\text{-Low-})\)

\((\text{target-node} a \text{ as} \text{ax} \rightarrow* (\text{-Low-}) \text{ res'} \text{snds} \text{re calculation}\)

\((\text{as}' = ax \# \text{as} \quad a = ax)\)

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

next

case \text{False}

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

have \((\text{snd} (\text{hd} s) = \text{snd} (\text{hd} s')\) by\((\text{erule-tac x=0 in allE})\) fastforce

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

\((\text{source-node} ax = m) \quad (\text{as}' = ax \# \text{as}) \quad \text{False}\)

\((\text{intra-kind} a) \quad (\text{intra-kind} (\text{kind} ax))\)

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

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

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

\((\text{length} s = \text{Suc} (\text{length} cs) \quad \text{length} s' = \text{Suc} (\text{length} cs)\)

have \text{False} by\((\text{fastforce intro!:re-branching-edges-slice-kinds-False[of a ax]}))

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

qed
qed
next

\textbf{case (slpa-Call \(cs\) \(a\) as \(Q\) \(r\) \(p\) \(fs\))}

\textbf{note} \(IH\) \(= \langle\!\langle\!m \text{ as'} \!\rangle\!\rangle\).

\[\text{[upd-cs (a \# cs) as = []]; same-level-path-aux (a \# cs) as';}\]
\[\forall c \in \text{set (a \# cs). valid-edge c; m as'\rightarrow\rightarrow (-\text{Low}-); m \rightarrow\rightarrow (-\text{Low}-);}\]
\[\forall i < \text{length (a \# cs).} \forall V \in rv S (CFG-node (source-node (a \# cs ! i))).\]

\(\text{fst (s ! Suc i) V = fst (s' ! Suc i) V;}\)
\[\forall i < \text{Suc (length (a \# cs)).} \text{snd (s ! i) = snd (s' ! i);}\]
\[\forall V \in rv S (CFG-node m). \text{state-val s V = state-val s' V;}\]
\[\text{preds (slice-kinds S as) s; preds (slice-kinds S as') s';}\]
\[\text{length s = Suc (length (a \# cs)); length s' = Suc (length (a \# cs)).}\]
\[\Rightarrow \forall V \in \text{Use (-\text{Low}-). state-val (transfers(slice-kinds S as) s) V =}\]
\[\text{state-val (transfers(slice-kinds S as') s') V;}\]

\textbf{note} \(\text{rus} = \forall i < \text{length cs.} \forall V \in rv S (CFG-node (source-node (cs ! i)).}\)

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

\textbf{from} \(\langle m \rightarrow a \# \rightarrow\rightarrow (-\text{Low}-)\rangle \text{ have source-node a = m and valid-edge a}\)
\[\text{and target-node a = \text{as'\rightarrow\rightarrow (-Low-)} by(auto elim:path-split-Cons)\]

\[\text{from} \\forall c \in \text{set cs. valid-edge c; valid-edge a}\]
\[\text{have valid-edge c by simp}\]

\textbf{show} \(\text{case}\)

\textbf{proof(cases L = {\})}\]

\textbf{case True with UseLow show \(?\text{thesis}\) by simp}\n
\textbf{next}\n
\textbf{case False}\n
\textbf{show \(?\text{thesis}\) proof(cases as')}\n
\textbf{case Nil}\n
\textbf{with} \(\langle m \rightarrow a \# \rightarrow\rightarrow (-\text{Low}-)\rangle \text{ have m = (-Low-) by fastforce}\)
\[\text{by \((\text{rule Exit-successor-of-Low,simp})\);}\]

\textbf{from Low-source-Exit-Edge obtain a' where valid-edge a'}
\[\text{and source-node a' = (-Low-) and target-node a' = (-Exit-)}\]
\[\text{and kind a' = (\(\lambda s. \text{True})\) by blast}\]

\textbf{from} \(\langle \text{valid-edge c; source-node a = m \rangle ; m = (-\text{Low-})} \rangle\)
\[\langle \text{target-node a = (-Exit-)} \rangle \langle \text{valid-edge a'} \rangle \langle \text{source-node a'} = (-\text{Low-})\rangle\]
\[\langle \text{target-node a' = (-Exit-)} \rangle\]
\[\text{have a = a' by(fastforce dest:edge-det)}\]
\[\text{with (kind a' = (\(\lambda s. \text{True})\)) by simp}\]

\textbf{with} \(\langle \text{target-node a = (-Exit-)} \rangle \langle \text{target-node a = \text{as'\rightarrow\rightarrow (-Low-)}\rangle}\)
\[\text{have (-Low-) = (-Exit-) by \((\text{drule path-Exit-source,auto})\);}\]

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

\textbf{thus \(?\text{thesis}\) by simp}\n
\textbf{next}\n
\textbf{case (Cons ax asx)}

\textbf{with} \(\langle m \rightarrow a \# \rightarrow\rightarrow (-\text{Low-})\rangle \text{ have source-node ax = m and valid-edge ax}\)
\[\text{and target-node ax = \text{as'\rightarrow\rightarrow (-Low-)} by(auto elim:path-split-Cons)}\]

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

\textbf{obtain cf cfs where \(\text{[simp]: s = cf # cfs by(cases s)(auto simp: slice-kinds-def)}}\]

12
from \( \text{preds} \text{(slice-kinds } S \text{ as } s') s' \langle as' = ax \# asx \rangle \)

obtain cf' cf's' where [simp]:\( s' = cf' \# cf's' \)

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

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

proof (cases kind ax rule:edge-kind-cases)

  case Intra

  have False

  proof (cases sourcenode ax \in [HRB-slice S] CFG)

    case True

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

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

    by \(~\) (rule slice-intra-kind-in-slice)

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

    have unique:\( \exists a'. \text{valid-edge } a' \land \text{sourcenode } a' = \text{sourcenode } a \land \text{intra-kind } (\text{kind } a') \)by (rule call-only-one-intra-edge)

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

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

    with \( \langle \text{valid-edge a} \rangle \langle \text{sourcenode } a = m \rangle \langle \text{sourcenode } a = m \rangle \langle \text{intra-kind } (\text{kind } ax) \rangle \)

    unique

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

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

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

    have False by (simp add: slice-kinds-def)

    thus \?thesis by simp

  next

  case False

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

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

  by (fastforce intro:slice-kind-Call)

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

  have False by (simp add: slice-kinds-def)

  thus \?thesis by simp

qed

thus \?thesis by simp

next

  case (Return \( Q' \ p' f' \))

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

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

  have False by \(~\) (drule return-edges-only,auto)

  thus \?thesis by simp

qed simp

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

proof (rule ccontr)

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

  from this \( \langle \text{kind } a = Q: r \mapsto p fs \rangle \)

13
\textbf{have} slice-kind \textit{S} \textit{a} = (\lambda \textit{cf}. False):r\to r_{pfs}

\textbf{by}(rule slice-kind-Call)

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

\textbf{show} False \textbf{by}(simp add:slice-kinds-def)

\textbf{qed}

\textbf{with} \langle \text{preds} (slice-kinds \textit{S} (\textit{a} \neq \textit{as})) \textit{s} \rangle \langle \text{kind} \textit{a} = Q:Q\to pfs \textit{fs} \rangle

\textbf{have} \text{pred} (\text{kind} \textit{a}) \textit{s} \textbf{by}(fastforce dest:slice-kind-Call-in-slice simp:slice-kinds-def)

\textbf{from} \langle \text{source-node} \textit{a} \in [HRB-slice \textit{S} ]_{\text{CFG}} \rangle

\langle \text{source-node} \textit{a} = \textit{m} \rangle \langle \text{source-node} \textit{ax} = \textit{m} \rangle

\textbf{have} \text{source-node} \textit{ax} \in [HRB-slice \textit{S} ]_{\text{CFG}} \textbf{by simp}

\textbf{with} \langle \text{as}' = \textit{ax} \neq \textit{ax}x \rangle \langle \text{preds} (slice-kinds \textit{S} \textit{as}') \textit{s}' \rangle

\exists Q \ r \ p \ \textit{fs}. \ \text{kind} \textit{ax} = Q:Q\to pfs

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

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

\textbf{from} \langle valid-edge \textit{a} \rangle \textbf{have} source-node \textit{a} = \textit{m} \to \ast \ source-node \textit{a} 

\textbf{by}(fastforce intro:empty-path simp:intra-path-def)

\textbf{with} \langle source-node \textit{a} \in [HRB-slice \textit{S} ]_{\text{CFG}} \rangle

\langle valid-edge \textit{a} \rangle \langle \textit{V} \in Use (source-node \textit{a}) \rangle

\textbf{have} \textit{V} \in \text{rv} \textit{S} (CFG-node (source-node \textit{a})) \textbf{by}(auto intro!:rvt CFG-Use-SDG-Use simp:SDG-to-CFG-set-def source-nodes-def)

\textbf{with} \langle \forall \textit{V} \in \text{rv} \textit{S} (CFG-node \textit{m})\text{, state-val} \textit{s} \textit{V} = \text{state-val} \textit{s}' \textit{V} \rangle

\langle source-node \textit{a} = \textit{m} \rangle

\textbf{have} Use\forall \textit{V} \in Use (source-node \textit{a})\text{, state-val} \textit{s} \textit{V} = \text{state-val} \textit{s}' \textit{V} \textbf{by simp}

\textbf{from} \forall \textit{i} < \text{Suc} (length \textit{cs})\text{, snd} (s ! \textit{i}) = \text{snd} (s' ! \textit{i})\rangle

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

\textbf{with} \langle valid-edge \textit{a} \rangle \langle \text{kind} \textit{a} = Q:Q\to pfs \textit{fs} \rangle \langle valid-edge \textit{ax} \rangle

\exists Q \ r \ p \ \textit{fs}. \ \text{kind} \textit{ax} = Q:Q\to pfs \langle \text{source-node} \textit{a} = \textit{m} \rangle \langle \text{source-node} \textit{ax} = \textit{m} \rangle

\langle \text{pred} (\text{kind} \textit{a}) \textit{s} \rangle \langle \text{pred} (\text{kind} \textit{ax}) \textit{s}' \rangle \langle \textit{V} \in \text{Use} \rangle \langle \text{length} \textit{s} = \text{Suc} (length \textit{cs}) \rangle

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

\textbf{have} \langle simp\rangle : \text{ax} = \textit{a} \textbf{by}(fastforce intro!:CFG-equal-Use-equal-call)

\textbf{from} \langle \text{same-level-path-ax cs} \textit{as}' \rangle \langle \text{as}' = \textit{ax}\#\textit{asx} \rangle \langle \text{kind} \textit{a} = Q:Q\to pfs \textit{fs} \rangle

\exists Q \ r \ p \ \text{kind} \textit{ax} = Q:Q\to pfs

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

\textbf{from} \langle \text{target-node} \textit{ax} \# \textit{asx} \to \ast \text{(-Low-)} \rangle \textbf{have} \langle \text{target-node} \textit{a} \# \textit{asx} \to \ast \text{(-Low-)} \rangle

\textbf{by simp}

\textbf{from} \langle \text{kind} \textit{a} = Q:Q\to pfs \rangle \langle \text{upd-cs} \textit{cs} (\textit{a} \# \textit{as}) = [] \rangle

\textbf{have} \langle \text{upd-cs} (\textit{a} \# \textit{cs}) \textit{as} = [] \rangle \textbf{by simp}

\textbf{from} \langle \text{source-node} \textit{a} \in [HRB-slice \textit{S} ]_{\text{CFG}} \rangle \langle \text{kind} \textit{a} = Q:Q\to pfs \textit{fs} \rangle

\textbf{have} \langle \text{slice-kind} \textit{slice-kind} \textit{S} \textit{a} = \text{Q:Q}\to r_{pfs} (\text{csp} \textit{target-node} \textit{a}) \langle \text{HRB-slice} \textit{S} \rangle \textit{fs} \rangle

\textbf{by}(rule slice-kind-Call-in-slice)

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

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

\textbf{by} auto(case-tac \textit{i},auto)
from ⟨valid-edge a⟩ (kind a = Q:r→p/fs) obtain ins outs

where (p, ins, outs) ∈ set procs by (fastforce dest!::callee-in-procs)

with ⟨valid-edge a⟩ (kind a = Q:r→p/fs)

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 (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→p/fs) (p, ins, outs) ∈ set procs

(by ⟨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→p/fs) (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)) = (fs!i) (fst (hd s’))

using ⟨i < length ins⟩

∀ i < length ins. (params fs (fst (hd s)))!i = (params fs (fst (hd s’)))!i

by simp }

definition case:⟨is in-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 = empty

by (rule csppa-formal-in-notin-slice)

thus ?thesis by simp

qed }
hence eq:∀ i < length fs.
((cspp (targetnode a) (HRB-slice S) fs)!i)(fst cf) =
((cspp (targetnode a) (HRB-slice S) fs)!i)(fst cf')
by(simp add:cspp-def)
{ fix i assume i < length fs
hence (params (cspp (targetnode a) (HRB-slice S) fs)
(fst cf))!i =
((cspp (targetnode a) (HRB-slice S) fs)!i)(fst cf')
by(fastforce intro:params-nth)
moreover
from (i < length fs)
have (params (cspp (targetnode a) (HRB-slice S) fs)
(fst cf'))!i =
((cspp (targetnode a) (HRB-slice S) fs)!i)(fst cf')
by(fastforce intro:params-nth)
ultimately
have (params (cspp (targetnode a) (HRB-slice S) fs)
(fst cf))!i =
(params (cspp (targetnode a) (HRB-slice S) fs)(fst cf')!i)
using eq (i < length fs) by simp }
hence params (cspp (targetnode a) (HRB-slice S) fs)(fst cf) =
params (cspp (targetnode a) (HRB-slice S) fs)(fst cf')
by(simp add:list-eq-iff-nth-eq)
with slice-kind (\( (p,\text{ins,outs}) \in \text{set procs} \))
obtain cfz where [simp]:
  transfer (slice-kind S a) (cf#cfs) = cfz#cf#cfs
  transfer (slice-kind S a) (cf'#cfs') = cfz#cf'#cfs'
by auto

hence rv:∀ V∈rv S (CFG-node (targetnode a)).
  state-val (transfer (slice-kind S a) s) V =
  state-val (transfer (slice-kind S a) s') V
by simp

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

have res':\( \forall i<\text{length (a ≠ cs)} \).
  \( \forall V \in \text{rv S (CFG-node (source node ((a ≠ cs) ! i))}. \)
  \( \text{fst ((transfer (slice-kind S a) s))! Suc i) V =} \)
  \( \text{fst ((transfer (slice-kind S a) s'))! Suc i) V} \)
by auto(case-tac i,auto)

from \( \text{preds (slice-kinds S (a ≠ as)) s} \)

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

moreover

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

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

moreover

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

have length (transfer (slice-kind S a) s) =
  Suc (length (a ≠ cs)) by simp
moreover
from \(\langle\text{length } s' = \text{Suc (length } cs)\rangle\)
have \(\text{length (transfer (slice-kind } S a) \; s') =
\text{Suc (length } (a \# cs))\) by simp
moreover
from \(IH[\text{OF } \langle\text{upd-cs } (a \# cs) \; \text{as} = []\rangle\) (same-level-path-aux \((a \# cs) \; \text{as}x)\)
\(\forall c \in \text{set } (a \# cs). \text{valid-edge } c)\) \(\langle\text{targetnode } a - \text{as} \rightarrow (\text{Low}-)\rangle\)
\(\langle\text{targetnode } a - \text{as} \rightarrow (-\text{Low})\rangle\) \(\text{rvs' snds rv calculation} \langle\text{as} = \text{ax#asx}\rangle\)
show ?thesis by (simp add: slice-kinds-def)
qed
next
case (spla-Return \(cs \; a \; \text{as } Q \; p \; f \; c' \; cs')\)
note \(IH = \langle\forall m \; a' \; s' \; s''. \; \text{upd-cs } cs' \; \text{as} = []\rangle\) (same-level-path-aux \(cs' \; \text{as}';\)
\(\forall c \in \text{set } cs'. \text{valid-edge } c \; m - \text{as} \rightarrow (\text{Low}-); \; m - \text{as}'' \rightarrow (\text{Low}-)\);
\(\forall i < \text{length } cs'. \forall V \in \text{rv } S) \; (CFG-node (source-node (cs'! i))).\)
\(\text{fst (s } \text{Suc } i) \; V = \text{fst (s' } \text{Suc } i) \; V;\)
\(\forall i < \text{Suc (length } cs'). \text{snd (s } i \; i) = \text{snd (s' } i \; i);\)
\(\forall V \in \text{rv } S) \; (CFG-node m). \; \text{state-val } s \; V = \text{state-val } s' \; V;\)
preds (slice-kinds \(S \; \text{as}\) \(s);\) preds (slice-kinds \(S \; \text{as}' \; s'');
\(\text{length } s = \text{Suc (length } cs'); \; \text{length } s' = \text{Suc (length } cs'))\)
\(\text{implies } \forall V \in \text{Use (Low-), state-val (transfers(slice-kinds } S \; \text{as}) \; s) \; V) =
\text{state-val (transfers(slice-kinds } S \; \text{as}') \; s') \; V.\)

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

from \(\langle m - a \# \text{as} \rightarrow (\text{Low}-)\rangle\) have \(\text{source-node } a = m \; \text{and valid-edge } a\)
and \(\text{targetnode } a - \text{as} \rightarrow (\text{Low}-)\) by (auto elim: path-split-Cons)

from \(\forall c \in \text{set } cs, \text{valid-edge } c) \; (cs = c' \; \# \; cs')\)
have \(\text{valid-edge } c' \; \text{and } \forall c \in \text{set } cs'. \; \text{valid-edge } c\) by simp-all
show ?case
proof (cases \(L = \{\}\))
case True with UseLow show ?thesis by simp
next
case False
show ?thesis
proof (cases \(as''\))
case Nil

with \(\langle m - \text{as}'' \rightarrow (\text{Low}-)\rangle\) have \(m = (\text{Low}-)\) by fastforce

with \(\langle\text{valid-edge } a) \; \text{source-node } a = m\rangle\) have \(\text{targetnode } a = (\text{Exit}-)\)
by -(rule Exit-successor-of-Low,simp+)

from Low-source-Exit-edge obtain \(a'\) where valid-edge \(a'\)
and \(\text{source-node } a' = (\text{Low}-)\) and targetnode \(a' = (\text{Exit}-)\)
and \(\text{kind } a' = (\lambda s. \text{True})\) by blast

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

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

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

with \(\langle\text{targetnode } a = (\text{Exit}-)\rangle\) \(\langle\text{targetnode } a - \text{as} \rightarrow (\text{Low}-)\rangle\)
have \ (-Low-) = \ (\text{Exit}) \ by \ -(\text{drule path-Exit-source}, \text{auto})

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

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

next

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

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

and \ \text{target-node} \ ax \ \rightarrow as' \ (\text{-Low}) \ by (auto \ \text{elim:path-split-Cons})

from \ (\text{valid-edge} \ ax) \ (\text{valid-edge} \ ax) \ (\text{kind} \ a = Q \leftarrow pf)

\ \langle \text{source-node} \ ax = m \rangle \ \langle \text{source-node} \ ax = m \rangle

have \ \exists J \ f, \ \text{kind} \ ax = Q \leftarrow pf \ by (auto \ \text{dest:return-edges-only})

with \ (\text{same-level-path-ax} \ cs \ as') \ (as' = ax \# \text{asx}) \ (cs = c' \# \ cs')

have \ ax \ \in \ \text{get-return-edges} \ cs' \ \text{and} \ \text{same-level-path-ax} \ cs' \ \text{asx \ by \ auto}

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

have \ [\text{simp}]: ax = a \ by (rule \ \text{get-return-edges-unique})

from \ (\text{target-node} \ ax \ \rightarrow as' \ (\text{-Low}) \ have \ \text{target-node} \ ax \ a \ \rightarrow \ (\text{-Low})

by \ \text{simp}

from \ (\text{upd-cs} \ cs \ (a \ # \ as)) = []); \ \langle \text{kind} \ a = Q \leftarrow pf \rangle \ (cs = c' \# \ cs')

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

have \ \text{upd-cs} \ cs' \ as = [] \ by \ \text{simp}

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

obtain \ cf \ cf' \ \text{where} \ s = cf \# cf' \# cf's

by (cases \ s, auto, case-tac list, fastforce+)

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

obtain \ cf' \ cfx' \ \text{where} \ s' = cf' \# cfx' \# cf's'

by (cases \ s', auto, case-tac list, fastforce+)

from \ res \ (cs = c' \# \ cs') \ (s = cf \# cfx' \# cf's) \ (s' = cf' \# cfx' \# cf's')

res1: \forall i < \text{length} \ cs',

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

fst ((cfx' \# cf's') ! Suc i) V = fst ((cfx' \# cf's') \ Suc i) V

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

(fst cfx) V = (fst cfx') V

by auto

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

obtain \ Qx \ \text{rx px fsx where} \ \text{kind} \ c' = Qx: \text{rx} \rightarrow px \text{fsx}

by (fastforce dest!:only-call-get-return-edges)

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

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

proof

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

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

obtain \ a' \ where \ edge:valid-edge \ a' \ \text{source-node} \ a' = \ \text{source-node} \ c'

\ \text{target-node} \ a' = \ \text{target-node} \ a \ \text{a intra-kind} \ (\text{kind} \ a')

by -(\text{drule call-return-node-edge}, \text{auto simp:intra-kind-def})

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

obtain \ as \ where \ \text{target-node} \ a \ \rightarrow as' \# \ \text{parent-node} \ n'

\ \text{and} \ \forall V \in \ \text{HRB-slice} \ S \ \text{and} \ V \ \in \ \text{Use}_{SDG} \ n'

\ \text{and} \ \forall n'' \ \text{valid-SDG-node} \ n'' \ \land \ \text{parent-node} \ n'' \ \in \ \text{set} \ (\text{source-nodes} \ as)

\ \longrightarrow \ V \ \notin \ \text{Def}_{SDG} \ n'' \ \text{by} (\text{fastforce elim:rvE})
from (targetnode a = as→,* parent-node n') edge
have source-node c' = a'#as→,* parent-node n'
  by (fastforce intro:Cons-path simp:inttra-path-def)
from (valid-edge c') (kind c' = Qx:zx→pxfsx) have Def (source-node c') = 

  by (rule call-source-Def-empty)

  hence ∀ n'' . valid-SDG-node n'' ∧ parent-node n'' = source-node c'
    → V ∉ Def SDG n'' by (fastforce dest:SDG-Def-parent-Def)
    with all (source-node a' = source-node c')
    have ∀ n''. valid-SDG-node n'' ∧ parent-node n'' ∈ set (source-nodes (a'#as))

    → V ∉ Def SDG n'' by (fastforce simp:source-nodes-def)
    with (source-node c' = a'#as→,* parent-node n')
      (n' ∈ HRB-slice S) . (V ∈ Use SDG n')
    show V ∈ rv S (CFG-node (source-node c'))
    by (fastforce intro:rvI)

qed

show ?thesis

proof (cases source-node a ∈ [HRB-slice S] CFG)
  case True
  from (valid-edge c') (a ∈ get-return-edges c')
  have get-proc (target-node c') = get-proc (source-node a)
    by -(drule intra-proc-additional-edge,
         auto dest: get-proc-intra simp:inttra-kind-def)
  moreover
  from (valid-edge c') (kind c' = Qx:zx→pxfsx)
  have get-proc (target-node c') = px by (rule get-proc-call)
  moreover
  from (valid-edge a: (kind a = Q→fλ)
  have get-proc (source-node a) = p by (rule get-proc-return)
  ultimately have [simp]: px = p by simp
  from (valid-edge c') (kind c' = Qx:zx→pxfsx)
  obtain ins outs where (p, ins, outs) ∈ set procs
    by (fastforce dest!: callee-in-procs)
  with (source-node a ∈ [HRB-slice S] CFG)
    (valid-edge a: (kind a = Q→fλ)
  have slice-kind: slice-kind S a =
    Q→p(λ cf' rspp (target-node a) (HRB-slice S) outs cf' cf)
    by (rule slice-kind-Return-in-slice)
  with (s = cf ≠ cfx ≠ cfs) . (s' = cf ≠ cfx ≠ cfs)
  have sx: transfer (slice-kind S a) s =
    (rspp (target-node a) (HRB-slice S) outs (fst cfx) (fst cf),
     snd cfx ≠ cfs
     and sx': transfer (slice-kind S a) s' =
    (rspp (target-node a) (HRB-slice S) outs (fst cfx') (fst cf'),
     snd cfx' ≠ cfs')
    by simp-all
  with rv1 have rv1: ∀ i < length cs'.
  ∀ V ∈ rv S (CFG-node (source-node (cs' ! i))).
\[
\text{fst} \ (\text{transfer} \ (\text{slice-kind} \ S \ a) \ s) \ ! \ Suc \ i \ V = \\
\text{fst} \ (\text{transfer} \ (\text{slice-kind} \ S \ a) \ s') \ ! \ Suc \ i \ V
\]

by fastforce

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

\langle s = cf\#cfx\#cfx' \ \langle s' = cf'\#cfx'\#cfx'' \rangle

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)

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

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

proof

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

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

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

\text{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-set-cone-nth})

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

show \ ?thesis

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

\text{case} True

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

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

by(auto intro: \text{rspp-Actual-out-in-slice})

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

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

\text{proof}

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

from \ <\text{valid-edge} \ a> \ (\text{source-node} \ a = m)
have parent-node (CFG-node m) \rightarrow\ast parent-node (CFG-node m)
  by (fastforce intro:empty-path simp:intra-path-def)
with (sourcenode a ∈ [HRB-slice S] CFG)
  \ V ∈ Use (sourcenode a) ; (sourcenode a = m) ; valid-edge a
show \ V ∈ rv S (CFG-node m)
  by -(rule 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 cfx′)((ParamDefs (targetnode a))!i)
by (auto intro!:rspp-Actual-outnotin-slice)
from \ \ \ ∀ 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′))
  \ ((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 \(sz \, sx'\)

have \(rv \; \forall \ V \in_{rv} S \; (CFG\text{-node} \; (targetnode \; a))\),
  state-val \((transfer \; (slice-kind \; S \; a) \; s) \; V = \)
  state-val \((transfer \; (slice-kind \; S \; a) \; s') \; V\)
  by fastforce

from \(\langle \text{preds} \; (slice-kinds \; S \; (a \; \# \; as)) \; s \rangle\)
  have \(\langle \text{preds} \; (slice-kinds \; S \; as) \rangle\)
    \((transfer \; (slice-kind \; S \; a) \; s)\)
    by \((\text{simp \; add: \text{slice-kinds-def}})\)

moreover

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

moreover

from \(\langle \text{length} \; s = \text{Suc} \; (\text{length} \; cs) \rangle\)
  \(\langle cs = c' \; \# \; cs' \; sx \rangle\)
  have \(\langle \text{length} \; (transfer \; (slice-kind \; S \; a) \; s) = \text{Suc} \; (\text{length} \; cs') \rangle\)
    by \((\text{simp \; simp \; add:} \; s = cf\#cfx\#cfs)\)

moreover

from \(\langle \text{length} \; s' = \text{Suc} \; (\text{length} \; cs) \rangle\)
  \(\langle cs = c' \; \# \; cs' \; sx' \rangle\)
  have \(\langle \text{length} \; (transfer \; (slice-kind \; S \; a) \; s') = \text{Suc} \; (\text{length} \; cs') \rangle\)
    by \((\text{simp \; simp \; add:} \; s' = cf'\#cfx'\#cfs')\)

moreover

from \(\text{HH} \; \langle \text{OF} \; \langle \text{upd-cs \; cs'} \; as = [] \rangle \; \langle \text{same-level-path-aux} \; cs' \; ax \rangle \rangle\)
  \(\forall \; c \in \text{set} \; cs'. \; \text{valid-edge} \; c\)
  \(\langle \text{targetnode} \; a \; \text{as} \rightarrow* \; \text{(-Low-)} \rangle\)
  \(\langle \text{targetnode} \; a \; \text{as} \rightarrow* \; \text{(-Low-)} \rangle\)
  have \(\text{res' \; snds \; rv' \; calculation} \; \langle as' = ax\#asx \rangle\)

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

next

case \(\text{False}\)
from \(\text{this \; \langle \text{kind} \; a = Q\leftarrow pf \rangle}\)
  have \(\langle \text{slice-kind:a} \; \text{slice-kind} \; S \; a = (\lambda cf. \; \text{True})\leftarrow p(\lambda cf \; cf'. \; cf') \rangle\)
    by \((\text{rule \; slice-kind-Return})\)

with \(\langle s = cf \# cfx\#cfs \rangle\)
  \(\langle s' = cf'\#cfx'\#cfs' \rangle\)
  have \(\langle \text{simp}: \text{transfer} \; (\text{slice-kind} \; S \; a) \; s = cfx\#cfs \rangle\)
    \(\text{transfer} \; (\text{slice-kind} \; S \; a) \; s' = cf'\#cfx'\#cfs' \; \text{by simp-all}\)
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 \(\text{snds:} \forall \; i < \text{Suc} \; (\text{length} \; cs)\).
  \(\text{snd} \; (\text{transfer} \; (\text{slice-kind} \; a) \; s \; \# \; i) = \)
  \(\text{snd} \; (\text{transfer} \; (\text{slice-kind} \; a) \; s' \; \# \; i) \; \text{by fastforce}\)

from \(\text{rss1} \; \text{have} \; \text{rss'} \forall \; i < \text{length} \; cs'.\)
  \(\forall \; V \in_{rv} S \; (CFG\text{-node} \; (\text{source-node} \; (cs' \; \# \; i)))\).
  \(\text{fst} \; ((\text{transfer} \; (\text{slice-kind} \; a) \; s) \; \# \; \text{Suc} \; i) \; V = \)
  \(\text{fst} \; ((\text{transfer} \; (\text{slice-kind} \; a) \; s') \; \# \; \text{Suc} \; i) \; V \)
  by fastforce

from \(\forall \; V \in_{rv} S \; (CFG\text{-node} \; (\text{target-node} \; a))\).
  \(V \in_{rv} S \; (CFG\text{-node} \; (\text{source-node} \; c'))\)
  \(\forall \; V \in_{rv} S \; (CFG\text{-node} \; (\text{source-node} \; c'))\).

22
\[(\text{fst } cf)\ V = (\text{fst } cf')\ V\]

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

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

\textbf{moreover}
\textbf{from} \(\{\text{preds (slice-kinds S as') s' as' = ax#axs}\}
\textbf{have} \text{preds (slice-kinds S asx)
(transfer (slice-kind S a) s')
by(simp add:slice-kinds-def)}\)

\textbf{moreover}
\textbf{from} \(\{\text{length s = Suc (length cs) cs = c' \# cs'}\)
\textbf{have} \text{length (transfer (slice-kind S a) s) = Suc (length cs')
by(simp,simp add:s = cf#cfx#cfs)}\)

\textbf{moreover}
\textbf{from} \(\{\text{length s' = Suc (length cs) cs = c' \# cs'}\)
\textbf{have} \text{length (transfer (slice-kind S a) s') = Suc (length cs')
by(simp,simp add:s' = cf'#cfx'#cfs')}\)

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

\textbf{qed}
\textbf{qed}
\textbf{qed}

\textbf{lemma} \(rv\text{-Low-Use-Low}:
\textbf{assumes} m \rightarrow as* (\text{-Low-}) \text{ and m \rightarrow as'} \rightarrow ax* (\text{-Low-}) \text{ and get-proc m = Main
and } \forall V \in r\ S\ \text{(CFG-node m). cf V = cf' V
and preds (slice-kinds S as) [(cf,undefined)]
and preds (slice-kinds S as') [(cf',undefined)]
and CFG-node (\text{-Low-}) \in S
shows } \forall V \in \text{Use (\text{-Low-}),
\text{state-val (transfers(slice-kinds S as) [(cf,undefined)] V =
state-val (transfers(slice-kinds S as') [(cf',undefined)] V
}\}\text{proof(cases as)}\)

\textbf{case} Nil
\textbf{with} \(m \rightarrow as* (\text{-Low-})\) \textbf{have} \text{valid-node m and m = (\text{-Low-)}
by(auto intro:path-valid-node simp:vp-def)
\}\text{fix V assume V \in Use (\text{-Low-)}
\textbf{moreover}
\textbf{from} \(\{\text{valid-node m} \langle m = (\text{-Low-})\) \textbf{have} (\text{-Low-}) \rightarrow [] \rightarrow (\text{-Low-)}
by(fastforce intro:empty-path simp:intra-path-def)}\)

\textbf{23}
moreover
from (valid-node m) (m = (-Low-)) (CFG-node (-Low-) ∈ S)
have CFG-node (-Low-) ∈ HRB-slice S
by (fastforce intro:HRB-slice-refl)
ultimately have V ∈ rv S (CFG-node m) using (m = (-Low-))
by (auto intro!:rvI CFG-Use-SDG-Use simp:sourcenodes-def)

hence ∀ V ∈ Use (-Low-), V ∈ rv S (CFG-node m) by simp

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

next

case False
from (m − as′→_*/ (-Low-)) have m − as′→* (-Low-) by (simp add:vp-def)
from (m − as′→* (-Low-)) (m = (-Low-)) have 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 Nil ∀ V ∈ rv S (CFG-node m), cf V = cf′ V
∀ V ∈ Use (-Low-), V ∈ rv S (CFG-node m)

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

qed

next

case (Cons ax asx)
with (m − as′→_*/ (-Low-)) have sourcenode ax = m and valid-edge ax
and targetnode ax − asx→* (-Low-)
by (auto elim:path-split-Cons simp:vp-def)

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

next

case False

show ?thesis
proof (cases as′)
case Nil
with (m − as′→_*/ (-Low-)) have m = (-Low-) by (fastforce simp:vp-def)
with (valid-edge ax) (sourcenode ax = m) have targetnode ax = (-Exit-)
by (rule Exit-successor-of-Low,simp+)

from Low-source-Exit-edge obtain a′ where valid-edge a′
and sourcenode a′ = (-Low-) and targetnode a′ = (-Exit-)
and kind a′ = (As, True) by blast

from (valid-edge ax) (sourcenode ax = m) (m = (-Low-))
⟨targetnode ax = (-Exit-): valid-edge a′⟩ ⟨sourcenode a′ = (-Low-)⟩
⟨targetnode a′ = (-Exit-)⟩
have \( ax = a' \) by (fastforce dest:edge-det)
with (kind \( a' = (\lambda s. \text{True}) \)) have kind \( ax = (\lambda s. \text{True}) \) by simp
with (targetnode \( ax = (\text{-Exit}) \)) (targetnode \( ax \rightarrow asx \rightarrow* (\text{-Low}) \))
have (\( \text{-Low} \)) = (\text{-Exit}) by (drule path-Exit-source,auto)
with False have False by (drule Low-neq-Exit,simp)
thus \( ?\text{thesis} \) by simp

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

lemma \text{nonInterference-path-to-Low}:
assumes \( [cf] \approx L [cf] \) and (\text{-High}) \( \notin [\text{HRB-slice} \ S]_{CFG} \)
and CFG-node (\text{-Low}) \( \in S \)
and (\text{-Entry}) \( \rightarrow as \rightarrow J* (\text{-Low}) \) and \( \text{preds} \ (\text{kinds} \ axx) \ [(cf,undefined)] \)
and (\text{-Entry}) \( \rightarrow as \rightarrow*' (\text{-Low}) \) and \( \text{preds} \ (\text{kinds} \ asx) \ [(cf',undefined)] \)
shows map fst (transfers (kinds \ axx) [(cf,undefined)]) \approx L
map fst (transfers (kinds \ asx) [(cf',undefined)])
proof
from (\text{-Entry}) \( \rightarrow as \rightarrow J* (\text{-Low}) \) (\text{preds} (kinds \ axx) [(cf,undefined)])
(CFG-node (\text{-Low}) \( \in S \))
obtain \( asx \) where \( \text{preds} \ (\text{slice-kinds} \ S \ axx) \ [(cf,undefined)] \)
and \( \forall V \in \text{Use} \ (\text{-Low}) \)
\( \text{state-val} \ (\text{transfers} \ (\text{slice-kinds} \ S \ axx) [(cf,undefined)]) \ V \)
= \( \text{state-val} \ (\text{transfers} \ (\text{kinds} \ axx) [(cf,undefined)]) \ V \)
and slice-edges \( S [] \ axx = \text{slice-edges} \ S [] \ axx \)
\[
\begin{align*}
\text{and} & \quad \text{transfers} (\text{kinsd as}) \ [(\text{cf}, \text{undefined})] \neq [] \\
\text{and} & \quad (\text{Entry}) \rightarrow \ast \quad (\text{Low}) \\
\text{by} & \quad (\text{erule fundamental-property-of-static-slicing}) \\
\text{from} & \quad (\text{Entry}) \rightarrow \ast \quad (\text{Low}) \quad (\text{preds} (\text{kinsd as})) \ [(\text{cf'}, \text{undefined})] \\
\text{where} & \quad \text{preds} (\text{kinsd S s}) \ [(\text{cf'}, \text{undefined})] \\
\text{and} & \quad \forall \ V \in \text{Use} \ (\text{Low}) \\
\text{state-val} & \quad (\text{transfers} (\text{kinsd S s})) [(\text{cf'}, \text{undefined})] \ V = \\
\text{state-val} & \quad (\text{transfers} (\text{kinsd S s})) [(\text{cf'}, \text{undefined})] \ V \\
\text{and} & \quad \text{slice-edges} S \ [] \ as' = \\
\text{slice-edges} & \quad S \ [] \ as' \\
\text{and} & \quad \text{transfers} (\text{kinsd s}) [(\text{cf'}, \text{undefined})] \neq [] \\
\text{by} & \quad (\text{erule fundamental-property-of-static-slicing}) \\
\text{from} & \quad (\text{cf}) \approx_L \ [(\text{cf'})] \ \{(\text{High})\} \ [\text{HRB-slice S}]_{\text{CFG}} \\
\text{have} & \quad \forall \ V \in \text{rv S} \ (\text{CFG-node} \ (\text{Entry})) \\
\text{by} & \quad (\text{fastforce dest:lowEquivalence-relevant-nodes-Entry}) \\
\text{with} & \quad (\text{Entry}) \rightarrow \ast \quad (\text{Low}) \quad (\text{Entry}) \rightarrow \ast \quad (\text{Low}) \\
\text{where} & \quad (\text{Entry}) \rightarrow \ast \quad (\text{Entry}) \rightarrow \ast \quad (\text{Low}) \\
\text{preds} (\text{kinsd S s}) & \ [(\text{cf'}, \text{undefined})] \\
\text{preds} (\text{kinsd S s}) & \ [(\text{cf'}, \text{undefined})] \\
\text{have} & \quad \forall \ V \in \text{Use} \ (\text{Low}) \\
\text{state-val} & \quad (\text{transfers} (\text{kinsd S s})) [(\text{cf}, \text{undefined})] \ V = \\
\text{state-val} & \quad (\text{transfers} (\text{kinsd S s})) [(\text{cf}, \text{undefined})] \ V \\
\text{by} & \quad -(\text{rule rv-Low-UseLow,auto intro:get-proc-Entry}) \\
\text{with} & \quad \forall \ V \in \text{Use} \ (\text{Low}) \\
\text{state-val} & \quad (\text{transfers} (\text{kinsd S s})) [(\text{cf}, \text{undefined})] \ V = \\
\text{state-val} & \quad (\text{transfers} (\text{kinsd S s})) [(\text{cf}, \text{undefined})] \ V \\
\text{by} & \quad -(\text{rule rv-Low-UseLow,auto intro:get-proc-Entry}) \\
\text{obtain} & \quad \text{map } \text{fst} \ (\text{transfers} (\text{kinsd s}) [(\text{cf}, \text{undefined})]) \approx_L \text{map } \text{fst} \ (\text{transfers} (\text{kinsd s}) [(\text{cf}, \text{undefined})]) \\
\text{show} & \quad \text{thesis} \text{ by} (\text{fastforce simp:lowEquivalence-def UseLow neq-Nil-conv})
\end{align*}
\]

\text{qed}

\text{theorem} \ \text{nonInterference-path}: \\
\text{assumes} \ [\text{cf}] \approx_L [\text{cf'}] \ \text{and} \ (\text{High}) \ \text{notin} \ [\text{HRB-slice S}]_{\text{CFG}} \\
\text{and} \quad \text{CFG-node} \ (\text{Low}) \ \in \ S \\
\text{and} \quad (\text{Entry}) \rightarrow \ast \quad (\text{Exit}) \ \text{and} \ \text{preds} (\text{kinsd s}) \ [(\text{cf}, \text{undefined})] \\
\text{and} \quad (\text{Entry}) \rightarrow \ast \quad (\text{Exit}) \ \text{and} \ \text{preds} (\text{kinsd s}) \ [(\text{cf'}, \text{undefined})] \\
\text{shows} \map \text{fst} \ (\text{transfers} (\text{kinsd s}) [(\text{cf}, \text{undefined})]) \approx_L \map \text{fst} \ (\text{transfers} (\text{kinsd s}) [(\text{cf'}, \text{undefined})]) \\
\text{proof} \quad - \\
\text{from} \quad (\text{Entry}) \rightarrow \ast \quad (\text{Exit}) \ \text{obtain} \ x \ \text{xs where} \ as = x \# \text{xs} \\
\text{and} \quad (\text{Entry}) = \text{sourcenode x} \ \text{and} \ \text{valid-edge} \ x \\
\text{and} \quad \text{targetnode x} \ -\text{xs} \rightarrow \ast \quad (\text{Exit}) \\
\text{apply} \text{(cases as} = []) \\
\text{apply} \text{(clarsimp simp:up-def,drule empty-path-nodes,drule Entry-noteq-Exit,simp)
by (fastforce elim: path-split-Cons simp: vp-def)
from (\-Entry\(- as \to \sqrt{\cdot} \cdot \-Exit\)\) have valid-path as by (simp add: vp-def)
from (valid-edge x) have valid-node (targetnode x) by simp
hence inner-node (targetnode x)
proof (cases rule: valid-node-cases)
  case Entry
  with (valid-edge x) have False by (rule Entry-target)
  thus ?thesis by simp
next
  case Exit
  with (targetnode x \to \sqrt{\cdot} \cdot \-Exit\)\) have \(xs = []\)
  by \-(\text{drule 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) (valid-edge z) (\-(\-Entry\) = sourcenode x)\)
  (sourcenode z = (\-Entry\)) Exit (targetnode z = (\-Exit\))
  have \(x = z\) by (fastforce intro: edge-det)
with \(\text{preds (kinds as)} [[\text{cf}, \text{undefined}]]\) \((as = x \# xs) \to xs = []\)
  \(\text{kind z} = (\\(\lambda s. \text{False})\)\)
  have False by (simp add: kinds-def)
  thus ?thesis by simp
qed simp
from (\-Entry\(- as \to \sqrt{\cdot} \cdot \-Exit\)\) obtain \(x' \to xs'\) where \(xs = xs'@[x']\)
  and targetnode x \to \sqrt{\cdot} \cdot \-Low\)\) and kind \(x' = (\\(\lambda s. \text{True})\)\)
  by (fastforce elim: Exit-path-Low-path)
with (\-Entry\) = sourcenode x) (valid-edge x)
  have (\-Entry\) \to x \# xs' \to (\-Low\)\) by (fastforce intro: Cons-path)
from (valid-path as) \((as = x \# xs) \to xs = xs'@[x']\)
  have valid-path \((x \# xs')\)
  by (simp add: valid-path-def del: valid-path-aux.simps)
  (rule valid-path-aux-split, simp)
with (\-Entry\) = x \# xs' \to (\-Low\)\) have (\-Entry\) \to x \# xs' \to (\-Low\)\)
  by (simp add: vp-def)
from \((\text{preds (kinds as)} [[\text{cf}, \text{undefined}]]\) have \(as = (x \# xs')@[x']\) by simp
with \(\text{preds (kinds (x \# xs'))} [[\text{cf}, \text{undefined}]]\)
  by (simp add: kinds-def preds-split)
from (\-Entry\) - as' \to \sqrt{\cdot} \cdot \-Exit\)\) obtain \(y \to ys\) where \(as' = y \# ys\)
  and (\-Entry\) = sourcenode y and valid-edge y
  and targetnode y \to ys \to (\-Exit\)
  apply (cases as' = [])
  apply (clarsimp simp: vp-def, drule empty-path-nodes, drule Entry-noteq-Exit, simp)
  by (fastforce elim: path-split-Cons simp: vp-def)
from (\-Entry\) - as' \to \sqrt{\cdot} \cdot \-Exit\)\) have valid-path as' by (simp add: vp-def)
from valid-edge y have valid-node (targetnode y) by simp
hence inner-node (targetnode y)
proof (cases rule: valid-node-cases)
  case Entry
with \( \text{valid-edge } y \) \textbf{have} False \textbf{by}(\text{rule Entry-target})

\textbf{thus} \( \vdash \)thesis \textbf{by} simp

\textbf{next}

\textbf{case} Exit

\textbf{with} \( \text{targetnode } y \rightarrow ys \rightarrow \ast \) \textbf{(Exit-) have} \( \text{ys} = [] \)

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

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

\textbf{and} \( \text{source-node } z = \text{(Entry-)} \textbf{ and} \text{ targetnode } z = \text{(Entry-)} \)

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

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

\textbf{have} \( \text{ys} = \text{y} \textbf{ by}(\text{fastforce intro:edge-det}

\textbf{with} \( \text{preds } (\text{kinds } a\prime) [(\text{cf}', \text{undefined})]; (\text{as}' = y \# \text{ys'} \textbf{ (ys} = [])

\textbf{ (kind } z = (\lambda s. \text{False}) \sqrt{\text{ by}} \text{ simp}

\textbf{thus} \( \vdash \)thesis \textbf{ by} simp

\textbf{qed}

\textbf{simp}

\textbf{with} \( \text{targetnode } y \rightarrow ys \rightarrow \ast \) \textbf{(Exit-) obtain} \( \text{ys'} \textbf{ where} \text{ ys} = ys \otimes [y'] \)

\textbf{and} \( \text{targetnode } y \rightarrow ys \rightarrow \ast \) \textbf{(Low-) and} \( \text{kind } y' = (\lambda s. \text{True}) \sqrt{\text{by}} \text{ fastforce elim:Exit-path-Low-path}

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

\textbf{have} \( \text{(Entry-) } y \# \text{ys'} \rightarrow \ast \) \textbf{(Low-) \textbf{by}(fastforce intro:Cons-path}

\textbf{from} \( \text{valid-path } as' \) \textbf{(as'} = y \# \text{ys'} \textbf{ (ys} = ys \otimes [y']

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

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

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

\textbf{with} \( \text{(Entry-) } y \# \text{ys'} \rightarrow \ast \) \textbf{(Low-) have} \( \text{(Entry-) } y \# \text{ys'} \rightarrow \ast \) \textbf{(Low-}

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

\textbf{from} \( \text{as'} = y \# \text{ys'} \textbf{ (ys} = ys \otimes [y']\) \textbf{have} \( \text{as'} = (y \# \text{ys'} \otimes [y'] \textbf{ by} \text{ simp}

\textbf{with} \( \text{preds } (\text{kinds } a\prime) [(\text{cf}', \text{undefined})]

\textbf{have} \( \text{preds } (\text{kinds } (y \# \text{ys'})) [(\text{cf}', \text{undefined})]

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

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

\textbf{ (Entry-) } x \# \text{xs'} \rightarrow \ast \) \textbf{(Low-) \textbf{ (preds} (\text{kinds } (x \# \text{xs'})) [(\text{cf}, \text{undefined})]

\textbf{ (Entry-) } y \# \text{ys'} \rightarrow \ast \) \textbf{(Low-) \textbf{ (preds} (\text{kinds } (y \# \text{ys'})) [(\text{cf}', \text{undefined})]

\textbf{have map } \textbf{fst } (\text{transfers } (\text{kinds } (x \# \text{xs'})) [(\text{cf}, \text{undefined})]

\textbf{map } \textbf{fst } (\text{transfers } (\text{kinds } (y \# \text{ys'})) [(\text{cf}', \text{undefined})]

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

\textbf{with} \( \text{as} = x \# \text{xs'} \) \( \text{xs} = xs \otimes [x]\) \textbf{ (kind } x' = (\lambda s. \text{True}) \sqrt{\text{ by}} \text{ simp add:kinds-def transfers-split}

\textbf{show} \( \vdash \)thesis

\textbf{apply}(\text{cases transfers } (\text{map kind } x\prime) \textbf{ (transfer} \textbf{ (kind } x \) [(\text{cf}, \text{undefined})])

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

\textbf{by}(\text{cases transfers } (\text{map kind } y\prime) \textbf{ (transfer} \textbf{ (kind } y \) [(\text{cf}', \text{undefined})]),

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

\textbf{qed}

end
The second theorem assumes that we have a 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 soucrenode targetnode kind valid-edge Entry
get-proc get-return-edges procs Main Exit Def Use ParamDefs ParamUses
H L High Low +
SemanticsProperty soucrenode targetnode kind valid-edge Entry get-proc
get-return-edges procs Main Exit Def Use ParamDefs ParamUses sem identifies
for soucrenode :: 'edge ⇒ 'node and targetnode :: 'edge ⇒ 'node
and kind :: 'edge ⇒ ('var,'val,'ret,'pname) edge-kind
and valid-edge :: 'edge ⇒ bool
and Entry :: 'node ('(Exit')-') and get-proc :: 'node ⇒ 'pname
and get-return-edges :: 'edge ⇒ 'edge set
and procs :: ('pname × 'var list × 'var list) list and Main :: 'pname
and Exit::'node ('(Exit')-')
and Def :: 'node ⇒ 'var set and Use :: 'node ⇒ 'var set
and ParamDefs :: 'node ⇒ 'var list and ParamUses :: 'node ⇒ 'var set list
and sem :: 'com ⇒ ('var ⇒ 'val) list ⇒ 'com ⇒ ('var ⇒ 'val) list ⇒ bool
\((\{(1,\_,-)\} ⇒ (1,\_,-))\) \([0,0,0,0]\) 81
and identifies :: 'node ⇒ 'com ⇒ bool (\(\Delta \triangleright - \{51,0\}\) 80)
and H :: 'var set and L :: 'var set
and High :: 'node ('(High')-') and Low :: 'node ('(Low')-') +
fixes final :: 'com ⇒ bool
assumes final-edge-Low: \([\text{final} \ c; \ n \triangleq c]\)
⇒ \exists a. valid-edge a ∧ soucrenode a = n ∧ targetnode a = (Low-) ∧ kind a = \(\triangleright id\)
begins

The following theorem needs the explicit edge from (-High-) to n. An approach using a \textit{init} predicate for initial statements, being reachable from (-High-) via a \(\langle \lambda s. \text{True}\rangle\), edge, does not work as the same statement could be identified by several nodes, some initial, some not. E.g., in the program

while (True) \Skip; ; Skip two nodes identify this initial statement: the initial node and the node within the loop (because of loop unrolling).

**theorem nonInterference:**

assumes \([cf_1]\) \(\equiv_L \ [cf_2]\) and \((-High-) \notin \ |HRB\-slice\ \ S\) CFG
and CFG-node (-Low-) \(\in \ S\)
and valid-edge a and soucrenode a = (-High-) and targetnode a = n
and kind a = (\(\lambda s. \text{True}\)) and \(n \triangleq c \ and \ final \ c'\)
and \(\langle c,cf_1\rangle \Rightarrow \langle c',s_1\rangle \ and \ \langle c,cf_2\rangle \Rightarrow \langle c',s_2\rangle\)
shows \(s_1 \equiv_L s_2\)
proof –

from High-target-Entry-edge obtain az where valid-edge ax
and soucrenode ax = (-Entry-) and targetnode ax = (-High-)
and kind ax = (\(\lambda s. \text{True}\)) by blast
from \( n \equiv c \) \langle \cdot, \{c f x, r\} \rangle \Rightarrow \langle c', s_1 \rangle \\
\text{obtain } n_1 \ as \ cf s_1 \ where \ n \mapsto \ s_1 \Rightarrow s_1 \ and \ n_1 \equiv c' \\
\text{and } \text{preds } (\text{kinds } a s_1) \ [(c f x, \text{undefined})] \\
\text{and } \text{transfers } (\text{kinds } a s_1) \ [(c f x, \text{undefined})] = \text{cf s}_1 \text{ and map } \text{fst } \text{cf s}_1 = s_1 \\
\text{by } (\text{fastforce dest fundamental-property}) \\
\text{from } (\mapsto \ s_1 \Rightarrow s_1) \ 	ext{valid-edge } a' \text{ with } s_1 \ 	ext{source-node } a = n:// \langle \text{target-node } a = n; \\
\text{kind } a = (\lambda s. \text{True})_v \rangle \\
\text{have } (\text{High}) - a \# a \mapsto a \Rightarrow s_1 \text{ by } (\text{fastforce intro: Cons-path simp: vp-def valid-path-def}) \\
\text{from } (\text{final } e') \langle n_1 \equiv c' \rangle \\
\text{obtain } a_1 \text{ where } \text{valid-edge } a_1 \text{ and source-node } a_1 = n_1 \\
\text{and } \text{target-node } a_1 = (\text{Low}) \text{ and kind } a_1 = \text{id} \text{ by } (\text{fastforce dest: final-edge-Low}) \\
\text{hence } n_1 - [a_1] \Rightarrow (\text{Low}) \text{ by } (\text{fastforce intro: path-edge}) \\
\text{with } (\text{High}) - a \# a \mapsto a \Rightarrow n_1 \text{ have } (\text{High}) - (a \# a_1) \equiv [a_1] \Rightarrow (\text{Low}) \\
\text{by } (\text{fastforce intro: path-append simp: vp-def}) \\
\text{with } (\text{valid-edge ax} \text{ source-node } ax = (\text{Entry}) \text{ target-node } ax = (\text{High})) \\
\text{have } (\text{Entry}) - ax \# (a \# a_1) \equiv (\text{Low}) \text{ by } (\text{rule Cons-path}) \\
\text{moreover} \\
\text{from } (\text{High}) - a \# a_1 \Rightarrow a \Rightarrow n_1 \text{ have valid-path-aux } [] (a \# a_1) \\
\text{by } (\text{simp add: vp-def valid-path-def}) \\
\text{with } \text{kind } a_1 = \text{id} \text{ have } \text{valid-path-aux } [] ((a \# a_1) \equiv [a_1]) \\
\text{by } (\text{fastforce intro: valid-path-aux-Append}) \\
\text{with } \text{kind } ax = (\lambda s. \text{True})_v \text{ have valid-path-aux } [] (ax \# (a \# a_1) \equiv [a_1]) \\
\text{by simp} \\
\text{ultimately have } (\text{High}) - ax \# (a \# a_1) \equiv (\text{Low}) \\
\text{by } (\text{simp add: vp-def valid-path-def}) \\
\text{from } (\text{valid-edge a}) \text{ kind } a = (\lambda s. \text{True})_v \text{ source-node } a = (\text{High}) \\
\text{target-node } a = n) \\
\text{have get-proc } n = \text{get-proc } (\text{High}) \\
\text{by } (\text{fastforce dest: get-proc-intra simp: intra-kind-def}) \\
\text{with get-proc } (\text{High}) \text{ have get-proc } n = \text{Main } \text{ by simp} \\
\text{from } (\text{valid-edge a}_1) \text{ source-node } a_1 = n_1; \text{ target-node } a_1 = (\text{Low}) \text{ kind } a_1 = \text{id} \\
\text{have get-proc } n_1 = \text{get-proc } (\text{Low}) \\
\text{by } (\text{fastforce dest: get-proc-intra simp: intra-kind-def}) \\
\text{with get-proc } (\text{Low}) \text{ have get-proc } n_1 = \text{Main } \text{ by simp} \\
\text{from } (n \mapsto s_1 \Rightarrow s_1) \text{ have } n \mapsto s_1 \Rightarrow s_1 \text{ n_1} \\
\text{by } (\text{cases as}_1) \\
\text{auto dest: vpa Main slpa intro: get-proc } n_1 = \text{Main } (\text{get-proc } n = \text{Main}) \\
\text{simp: vp-def valid-path-def valid-call-list-def slp-def} \\
\text{same-level-path-def simp del: valid-path-aux-simps} \\
\text{then obtain } c f x r \text{ where } c f x \text{ transfers } (\text{map kind } a s_1) [(c f x, \text{undefined})] = [(c f x, r)] \\
\text{by } (\text{fastforce elim: slp-callbackstack-length-equal simp: kinds-def}) \\
\text{from } \text{kind } ax = (\lambda s. \text{True})_v \text{ kind } a = (\lambda s. \text{True})_v \text{ pred } (\text{kinds } a s_1) [(c f x, \text{undefined})] \text{ kind } a_1 = \text{id} c f x \\
\text{have pred } (\text{kinds } (ax \# ((a \# a_1) \equiv [a_1]))) [(c f x, \text{undefined})] \\
\text{by } (\text{auto simp: kinds-def pred-split}) \\
\text{from } (n \equiv c) \langle c, \{c f x, \{c f x\} \} \Rightarrow \langle c', s_2 \rangle)
\[ \text{obtain } n_2 \text{ as } n \to \alpha(n_2) \text{ where } n = \alpha(n_2)\text{ and } n_2 \triangleq c' \]
\[ \text{and preds (kinds as2) } [[cfs_2, \text{undefined}]] \]
\[ \text{and transfers (kinds as2) } [[cfs_2, \text{undefined}]] = cfs_2 \text{ and map fst cfs}_2 = s_2 \]
\[ \text{by (fastforce dest: fundamental-property)} \]
\[ \text{from } (n \to \alpha(n_2), \star) \text{ targetnode } a = (\text{(-Low-)}) \text{ (targetnode } a = n) \]
\[ \langle \text{kind } a = (\lambda s. \text{True}) \rangle \]
\[ \text{have } (\text{(-Low-)}) - a \#(\alpha(n_2)) \to \star \text{ by (fastforce intro: Cons-path simp: vs-def valid-path-def)} \]
\[ \text{from } (\text{final c'}) \langle n_2 \triangleq c' \rangle \]
\[ \text{obtain } a_2 \text{ where valid-edge } a_2 \text{ and soucenode } a_2 = n_2 \]
\[ \text{and targetnode } a_2 = (\text{(-Low-)}) \text{ and } kind a_2 = \ulcorner \text{id} \urcorner \text{ by (fastforce dest: final-edge-Low)} \]
\[ \text{hence } n_2 - a_2 \to \star (\text{-Low-}) \text{ by (fastforce intro: path-edge)} \]
\[ \text{with } (\text{(-Low-)}) - a \#(\alpha(n_2)) \to \star \text{ have } (\text{(-Low-)}) - (a \#(\alpha(n_2)) = (\alpha(n_2))) \to \star (\text{-Low-}) \]
\[ \text{by (fastforce intro: path-Append simp: vs-def)} \]
\[ \text{with } \langle \text{valid-edge } a \rangle \text{ (soucenode } a = (\text{(-Entry-)}) \text{ (targetnode } a = (\text{(-High-)}) \text{ have } (\text{(-Entry-)}) - ax\#((a \#(\alpha(n_2)))) \to \star (\text{-Low-}) \text{ by } (\text{rule Cons-path}) \]
\[ \text{moreover } \]
\[ \text{from } (\langle \text{(-Low-)}) - a \#(\alpha(n_2)) \to \star \text{ have valid-path-aux } \langle \text{final c'} \rangle \text{ (a} \#(\alpha(n_2)) \rangle \]
\[ \text{by (simp add: vs-def valid-path-def)} \]
\[ \text{with } \langle \text{kind } a_2 = \ulcorner \text{id} \urcorner \text{ have valid-path-aux } \langle ((a \#(\alpha(n_2))) = \langle \text{valid-path-aux } \langle ax\#((a \#(\alpha(n_2)))) \rangle \text{ by simp} \]
\[ \text{ultimately have } (\text{-Entry-)}) - ax\#((a \#(\alpha(n_2)))) \to \star (\text{-Low-}) \]
\[ \text{by (simp add: vs-def valid-path-def)} \]
\[ \text{from } (\langle \text{valid-edge } a \rangle \langle \text{kind } a = (\lambda s. \text{True}) \rangle \langle \text{soucenode } a = (\text{(-High-)}) \text{ (targetnode } a = n) \]
\[ \text{have get-proc } n = \text{get-proc } (\text{-High-}) \]
\[ \text{by (fastforce dest: get-proc-intra simp: intra-kind-def)} \]
\[ \text{with get-proc-High have get-proc } n = \text{Main by simp} \]
\[ \text{from } (\langle \text{valid-edge } a \rangle \langle \text{soucenode } a = n_2 \rangle \langle \text{targetnode } a_2 = (\text{-Low-}) \rangle \langle \text{kind } a_2 = \ulcorner \text{id} \urcorner \rangle \]
\[ \text{have get-proc } n_2 = \text{get-proc } (\text{-Low-}) \]
\[ \text{by (fastforce dest: get-proc-intra simp: intra-kind-def)} \]
\[ \text{with get-proc-Low have get-proc } n_2 = \text{Main by simp} \]
\[ \text{from } (n \to \alpha(n_2), \star) \text{ have } n - \alpha(n_2) \to \star \text{ by (cases as2)} \]
\[ \text{auto dest: vsa:Main-slap intro: get-proc } n_2 = \text{Main} \text{ (get-proc } n = \text{Main)} \text{ simp: vs-def valid-path-def valid-call-list-def slp-def}
\[ \text{same-level-path-def simp del: valid-path-aux.simps)} \]
\[ \text{then obtain } cfs' \]
\[ \text{where } cfs': \text{transfers (map kind as2) } [[cfs_2, \text{undefined}]] = [[cfs', r']] \]
\[ \text{by (fastforce elim: slp-callstack-length-equal simp: kinds-def)} \]
\[ \text{from } \langle \text{kind } ax = (\lambda s. \text{True}) \rangle \langle \text{kind } a = (\lambda s. \text{True}) \rangle \]
\[ \langle \text{preds (kinds as2)} \rangle \langle [[cfs_2, \text{undefined}]] \rangle \langle \text{kind } a_2 = \ulcorner \text{id} \urcorner \rangle cfs' \]
\[ \text{have preds (kinds } (ax\#((a \#(\alpha(n_2)))) = [[cfs_2, \text{undefined}]] \]
\[ \text{by (auto simp: kinds-def preds-split)} \]
\[ \text{from } ((cfs_2) \approx L \langle \text{cfs} \rangle \langle \text{(-Low-) } \notin \text{ (HRB-slice } S \rangle \langle \text{CFG} \rangle \langle \text{CFG-node } (\text{-Low-}) \in S \rangle)
\[ \langle \text{-Entry-)}) - ax\#((a \#(\alpha(n_2)))) \to \star (\text{-Low-}) \rangle \]

31
\[ \langle \text{preds (kinds (ax#((a#as_1)@[a_1])) (cf_1,undefined))} \rangle \]
\[ \langle \text{(-Entry-)} \rightarrow \text{ax#((a#as_2)@[a_2])→/^* (-Low-)}; \quad \text{preds (kinds (ax#((a#as_2)@[a_2])) (cf_2,undefined))} \rangle \]

\[ \text{have map fst (transfers (kinds (ax#((a#as_1)@[a_1])) (cf_1,undefined)) \approx_L} \]
\[ \text{map fst (transfers (kinds (ax#((a#as_2)@[a_2])) (cf_2,undefined))} \]}

by (rule nonInterference-path-to-Low)

with \( \text{kind ax = (λs. True)} \) \( \text{kind a_1 = (λs. True)} \) \( \text{kind a_2 = ↑id} \) \( \text{kind a_2 = ↑id} \)

\[ \langle \text{transfers (kinds as_1) (cf_1,undefined)} = \text{cfs_1} \rangle \quad \langle \text{map fst cfs_1 = s_1} \rangle \]
\[ \langle \text{transfers (kinds as_2) (cf_2,undefined)} = \text{cfs_2} \rangle \quad \langle \text{map fst cfs_2 = s_2} \rangle \]

show ?thesis by (cases s_1)(cases s_2,(fastforce simp:kinds-def transfers-split)+)+

qed

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 fulfill all properties of the NonInterferenceIntraGraph locale. Basically, we redefine the hitherto existing Entry and Exit nodes as new High and Low nodes, and introduce two new nodes NewEntry and NewExit. Then, we have to lift all functions to operate on this new graph.

3.1 Liftings

3.1.1 The datatypes
datatype 'node LDCFG-node = Node 'node
| NewEntry
| NewExit

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

3.1.2 Lifting basic definitions using 'edge and 'node
inductive lift-valid-edge :: ('edge ⇒ bool) ⇒ ('edge ⇒ 'node) ⇒ ('edge ⇒ 'node)
⇒ ('edge ⇒ ('var,'val,'ret,'pname) edge-kind ⇒ 'node ⇒ 'node ⇒ ('edge,'node,'var,'val,'ret,'pname) LDCFG-edge ⇒ bool

32
for valid-edge :: 'edge ⇒ bool and src :: 'edge ⇒ 'node and trg :: 'edge ⇒ 'node and knd :: 'edge ⇒ ('var', 'val', 'ret', 'pname) edge-kind and E :: 'node and X :: 'node

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

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

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

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

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

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

inductive-set lift-get-return-edges :: ('edge ⇒ 'edge set) ⇒ ('edge ⇒ bool) ⇒
  ('edge ⇒ 'node) ⇒ ('edge ⇒ 'node) ⇒ ('edge ⇒ ('var', 'val', 'ret', 'pname) edge-kind)
  ⇒ ('edge, 'node, 'var', 'val', 'ret', 'pname) LDCFG-edge
  ⇒ ('edge, 'node, 'var', 'val', 'ret', 'pname) LDCFG-edge set
  for get-return-edges :: 'edge ⇒ 'edge set and valid-edge :: 'edge ⇒ bool
  and src :: 'edge ⇒ 'node and trg :: 'edge ⇒ 'node
  and knd :: 'edge ⇒ ('var', 'val', 'ret', 'pname) edge-kind
  and e :: ('edge, 'node, 'var', 'val', 'ret', 'pname) LDCFG-edge
  where lift-get-return-edgesI:
    [e = (Node (src a), knd a, Node (trg a)); valid-edge a; a ∈ get-return-edges a;
     e' = (Node (src a'), knd a', Node (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 ⇒
\['var\ set \Rightarrow \ 'var\ set \Rightarrow (\ 'node\ LDCFG\text{-node} \times \ 'var\)\ set\\
\text{for}\ \text{Def::(}'node \Rightarrow \ 'var\ set\text{)}\ \text{and}\ \text{E::}'node\ \text{and}\ \text{X::}'node\
\text{and}\ \text{H::}'var\ set\ \text{and}\ \text{L::}'var\ set\\
\text{where}\ \text{lift-Def\-node}:\\
V \in \text{Def} \ L \implies (\text{Node} \ n, V) \in \text{lift-Def\-set} \ \text{Def} \ E \ H \ L\\
| \text{lift-Def\-High}:\\
V \in \text{H} \implies (\text{Node} \ E, V) \in \text{lift-Def\-set} \ \text{Def} \ E \ H \ L\\
\text{abbreviation}\ \text{lift-Def} :: (\ 'node \Rightarrow \ 'var\ set) \Rightarrow \ 'node \Rightarrow \ 'node\ \Rightarrow \ 'var\ set \Rightarrow (\ 'node\ LDCFG\text{-node} \Rightarrow \ 'var\ set)\\
\text{where}\ \text{lift-Def} \ \text{Def} \ E \ H \ L \ n \equiv \{V. \ (n, V) \in \text{lift-Def\-set} \ \text{Def} \ E \ H \ L\}\\
\text{inductive-set}\ \text{lift-Use\-set} :: (\ 'node \Rightarrow \ 'var\ set) \Rightarrow \ 'node \Rightarrow \ 'node\ \Rightarrow \ 'var\ set \Rightarrow (\ 'node\ LDCFG\text{-node} \Rightarrow \ 'var\ set)\\
\text{for}\ \text{Use::}'node \Rightarrow \ 'var\ set\ \text{and}\ \text{E::}'node\ \text{and}\ \text{X::}'node\
\text{and}\ \text{H::}'var\ set\ \text{and}\ \text{L::}'var\ set\\
\text{where}\ \text{lift-Use\-node}:\\
V \in \text{Use} \ L \implies (\text{Node} \ n, V) \in \text{lift-Use\-set} \ \text{Use} \ E \ H \ L\\
| \text{lift-Use\-High}:\\
V \in \text{H} \implies (\text{Node} \ E, V) \in \text{lift-Use\-set} \ \text{Use} \ E \ H \ L\\
| \text{lift-Use\-Low}:\\
V \in \text{L} \implies (\text{Node} \ X, V) \in \text{lift-Use\-set} \ \text{Use} \ E \ H \ L\\
\text{abbreviation}\ \text{lift-Use} :: (\ 'node \Rightarrow \ 'var\ set) \Rightarrow \ 'node \Rightarrow \ 'node\ \Rightarrow \ 'var\ set \Rightarrow (\ 'node\ LDCFG\text{-node} \Rightarrow \ 'var\ set)\\
\text{where}\ \text{lift-Use} \ \text{Use} \ E \ H \ L \ n \equiv \{V. \ (n, V) \in \text{lift-Use\-set} \ \text{Use} \ E \ H \ L\}\\
\text{fun}\ \text{lift-ParamUses} :: (\ 'node \Rightarrow \ 'var\ set\ list) \Rightarrow \ 'node\ LDCFG\text{-node} \Rightarrow \ 'var\ set\ list\\
\text{where}\ \text{lift-ParamUses} \ \text{ParamUses} \ (\text{Node} \ n) = \ \text{ParamUses} \ n\\
| \text{lift-ParamUses} \ \text{ParamUses} \ \text{NewEntry} = []\\
| \text{lift-ParamUses} \ \text{ParamUses} \ \text{NewExit} = []\\
\text{fun}\ \text{lift-ParamDefs} :: (\ 'node \Rightarrow \ 'var\ list) \Rightarrow \ 'node\ LDCFG\text{-node} \Rightarrow \ 'var\ list\\
\text{where}\ \text{lift-ParamDefs} \ \text{ParamDefs} \ (\text{Node} \ n) = \ \text{ParamDefs} \ n\\
| \text{lift-ParamDefs} \ \text{ParamDefs} \ \text{NewEntry} = []\\
| \text{lift-ParamDefs} \ \text{ParamDefs} \ \text{NewExit} = []
3.2 The lifting lemmas

3.2.1 Lifting the CFG locales

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

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

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

lemma lift-CFG:
  assumes wf:CFGExit-wf sourcenode targetnode kind valid-edge Entry get-proc
    get-return-edges procs Main Exit Def Use ParamDefs ParamUses
  and pd:Postdomination sourcenode targetnode kind valid-edge Entry get-proc
    get-return-edges procs Main Exit
  shows CFG src trg knd
    (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit 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'
      and src a = src a' and trg a = trg a'
      thus a = a'
proof (induct rule: lift-valid-edge.induct)
case lve-edge thus False by (erule lift-valid-edge.cases, auto dest: edge-det)
qed (auto elim: lift-valid-edge.cases)
next
  fix a Q r f
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
      and \text{knd} a = \text{Q:} r \rightarrow \text{Main} f
  thus False by (fastforce elim: lift-valid-edge.cases dest: Main-no-call-target)
next
  fix a Q' f'
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
      and \text{knd} a = \text{Q:} \leftarrow \text{Main} f'
  thus False by (fastforce elim: lift-valid-edge.cases dest: Main-no-return-source)
next
  fix a Q r p fs
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
      and \text{knd} a = \text{Q:} r \rightarrow p \text{fs}
  thus \exists \text{ins outs}. (p, \text{ins}, \text{outs}) \in \text{set procs}
      by (fastforce elim: lift-valid-edge.cases intro: callee-in-procs)
next
  fix a \text{Q:} r p fs
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
      and \text{knd} a = \text{Q:} r \rightarrow p \text{fs}
  thus lift-get-proc get-proc Main (\text{src} a) = lift-get-proc get-proc Main (\text{try} a)
      by (fastforce elim: lift-valid-edge.cases intro: get-proc-intra
          simp: get-proc-Entry get-proc-Exit)
next
  fix a Q' p f'
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
      and \text{knd} a = \text{Q:} \leftarrow p f'
  thus lift-get-proc get-proc Main (\text{src} a) = p
      by (fastforce elim: lift-valid-edge.cases intro: get-proc-return)
next
  fix a Q r p fs
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
      and \text{knd} a = \text{Q:} r \rightarrow p \text{fs}
  then obtain \text{ax where valid-edge ax and kind ax = Q:} r \rightarrow p \text{fs}
      and sourcenode ax \neq \text{Entry} \lor targetnode ax \neq \text{Exit}
      and \text{src} a = \text{Node} (\text{sourcenode ax}) and \text{try} a = \text{Node} (\text{targetnode ax})
      by (fastforce elim: lift-valid-edge.cases)
  from valid-edge ax \text{ kind ax = Q:} \rightarrow p \text{fs};
  have all: \forall \text{a'. valid-edge a' and targetnode a' = targetnode ax} \rightarrow
      (\exists Qx r x fsx. \text{kind a' = Qx:} r 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 try a′ = try a
 hence ∃ Qx rx fsx. knd a′ = Qx:rx→pfsx
 proof(induct rule:lift-valid-edge.induct)
   case (lve-edge ax′ e)
     note [simp] = ⟨e = (Node (sourcenode ax′), kind ax′, Node (targetnode ax′))⟩
     from ⟨try e = try a⟩ ⟨try a = Node (targetnode ax)⟩
     have targetnode ax′ = targetnode ax by simp
     with ⟨valid-edge ax′⟩ all have ∃ Qx rx fsx. knd a′ = Qx:rx→pfsx by blast
     thus ?case by simp
   next
   case (lve-Entry-edge e)
     from ⟨e = (NewEntry, (λs. True), Node Entry)⟩ ⟨try e = try a⟩
     ⟨try a = Node (targetnode ax)⟩
     have targetnode ax = Entry by simp
     with ⟨valid-edge ax⟩ have False by (rule Entry-target)
     thus ?case by simp
   next
   case (lve-Exit-edge e)
     from ⟨e = (Node Exit, (λs. True), NewExit)⟩ ⟨try e = try a⟩
     ⟨try a = Node (targetnode ax)⟩
     have False by simp
     thus ?case by simp
   next
   case (lve-Entry-Exit-edge e)
     from ⟨e = (NewEntry, (λs. False), NewExit)⟩ ⟨try e = try a⟩
     ⟨try a = Node (targetnode ax)⟩
     have False by simp
     thus ?case by simp
 qed }

thus ∀ a′. lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a′ ∧
  try a′ = try a → (∃ Qx rx fsx. knd a′ = Qx:rx→pfsx) by simp

next
 fix a Q p f
 assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
 and knd a = Q′:p′
 then obtain ax where valid-edge ax and kind ax = Q′:p′
 and sourcenode ax ≠ Entry ∨ targetnode ax ≠ Exit
 and src a = Node (sourcenode ax) and try a = Node (targetnode ax)
 by (fastforce elim:lift-valid-edge_cases)
 from ⟨valid-edge ax⟩ ⟨kind ax = Q′:p′⟩
 have all;∀ a′. valid-edge a′ ∧ sourcenode a′ = sourcenode ax →
  (∃ Qx fsx. kind a′ = Qx:fsx) by (auto dest:return-edges-only)
 { fix a′
   assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a′
   and src a′ = src a
   hence ∃ Qx fsx. knd a′ = Qx:fsx
   proof (induct rule:lift-valid-edge.induct)
case (lve-edge ax' e)
  note [simp] = \( e = (\text{Node (sourcenode ax')}, \text{kind ax'}, \text{Node (targetnode ax')}) \)
  from \( \langle \src e = \src a \rangle \langle \src a = \text{Node (sourcenode ax)} \rangle \)
  have sourcenode ax' = sourcenode ax by simp
  with \( \text{(valid-edge ax') all have } \exists Qx fx. \text{kind ax'} = Qx \leftarrow pfx \text{ by blast} \)
  thus \( ?\text{case by simp} \)
next
case (lve-Entry-edge e)
  from \( \langle e = (\text{NewEntry}, (\lambda s. \text{True}) \leftarrow, \text{Node Entry}) \rangle \langle \src e = \src a \rangle \langle \src a = \text{Node (sourcenode ax)} \rangle \)
  have False by simp
  thus \( ?\text{case by simp} \)
next
case (lve-Exit-edge e)
  from \( \langle e = (\text{Node Exit}, (\lambda s. \text{True}) \leftarrow, \text{NewExit}) \rangle \langle \src e = \src a \rangle \langle \src a = \text{Node (sourcenode ax)} \rangle \)
  have sourcenode ax = Exit by simp
  with \( \text{(valid-edge ax) have False by (rule Exit-source)} \)
  thus \( ?\text{case by simp} \)
next
case (lve-Entry-Exit-edge e)
  from \( \langle e = (\text{NewEntry}, (\lambda s. \text{False}) \leftarrow, \text{NewExit}) \rangle \langle \src e = \src a \rangle \langle \src a = \text{Node (sourcenode ax)} \rangle \)
  have False by simp
  thus \( ?\text{case by simp} \)
qed

thus \( \forall a'. \text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a'} \land \text{src a'} = \text{src a} \rightarrow (\exists Qx fx. \text{knd a'} = Qx \leftarrow pfx) \text{ by simp} \)

next
fix a Q r p fs
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
  and knd a = Q \leftarrow 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 \( \langle e = (\text{Node (sourcenode ax)}, \text{kind ax}, \text{Node (targetnode ax)}) \rangle \langle knd e = Q \leftarrow r \rightarrow p fs \rangle \)
  have kind ax = Q \leftarrow r \rightarrow p fs by simp
  with \( \text{(valid-edge ax) have get-return-edges ax} \neq \{\} \)
  by (rule get-return-edge-call)
  then obtain ax' where ax' \in \text{get-return-edges ax} \text{ by blast} \)
  with \( \langle e = (\text{Node (sourcenode ax)}, \text{kind ax}, \text{Node (targetnode ax)}) \rangle \langle \text{valid-edge ax} \rangle \)
  have \( \langle \text{Node (sourcenode ax')}, \text{kind ax'}, \text{Node (targetnode ax')} \rangle \in \text{lift-get-return-edges get-return-edges valid-edge} \text{ sourcenode targetnode kind e} \)
  by (fastforce intro:lift-get-return-edgesI)
  thus \( ?\text{case by fastforce} \)
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 lift-valid-edge valid-edge source-node target-node 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(source-node a') = p
by (rule get-proc-return)
have source-node a' ≠ Entry
proof
  assume source-node a' = Entry
  with get-proc-Entry (get-proc(source-node a') = p) have p = Main by simp
  with (kind a' = Q' ↪→ p; f') have kind a' = Q' ↪→ Main' by simp
  with (valid-edge a') show False by (rule Main-no-return-source)
qed
with (e' = (Node (source-node a'), kind a', Node (target-node 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 source-node
target-node kind a
and lift-valid-edge valid-edge source-node target-node kind Entry Exit a
thus ∃ Q r p f s. kind a = Q; r ↪→ p; f; s
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 f s. kind ax = Q; r ↪→ p; f; s
  by (rule only-call-get-return-edges)
  with (a = (Node (source-node ax), kind ax, Node (target-node ax)))
  show ?case by simp
qed
next
fix a Q r p f s a'
assume a' ∈ lift-get-return-edges get-return-edges
valid-edge source-node target-node kind a and kind a = Q; r ↪→ p; f; s
and lift-valid-edge valid-edge source-node target-node 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 (source-node ax), kind ax, Node (target-node ax)))
(\text{kind a} = \text{Q}\text{e}^\rightarrow_p \text{fs})
\text{have kind ax = Q}\text{e}^\rightarrow_p \text{fs by simp}
\text{with (valid-edge ax) (a' \in get-return-edges ax) have } \exists \text{Q}' \text{f'}, \text{kind a' = Q}'\text{e}^\rightarrow_p \text{f'}
\text{by -(rule call-return-edges)}
\text{with (e' = (Node (sourcenode a'), kind a', Node (targetnode a'))) show ?case by simp}
\text{qed}
\text{next}
\text{fix a Q' p f'}
\text{assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a}
\text{and kinds a = Q}'\text{e}^\rightarrow_p \text{f'}
\text{thus } \exists !a'. \text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a' }\wedge
(\exists \text{Q' r fs. kind a' = Q'}\text{r}^\rightarrow_p \text{fs}) \wedge a \in \text{get-return-edges get-return-edges valid-edge sourcenode targetnode kind a'}
\text{proof (induct rule:lift-valid-edge.induct)}
\text{case (lve-edge a e)}
\text{from (e = (Node (sourcenode a), kind a, Node (targetnode a)))}
\text{\text{\langle kind e = Q}'\text{e}^\rightarrow_p \text{f'} \text{have kind a = Q}'\text{e}^\rightarrow_p \text{f'} \text{by simp}}
\text{with (valid-edge a)}
\text{\text{\langle valid-edge a' \wedge (\exists \text{Q' r fs. kind a' = Q'}\text{r}^\rightarrow_p \text{fs}) \wedge a \in \text{get-return-edges a'}}
\text{\text{by (rule return-needs-call)}}
\text{then obtain a' Q r fs where valid-edge a' and kind a' = Q'}\text{r}^\rightarrow_p \text{fs}
\text{and a \in get-return-edges a'}
\text{and \text{imp}:\forall x. valid-edge x \wedge (\exists \text{Q} r fs. \text{kind x = Q'}\text{r}^\rightarrow_p \text{fs}) \wedge a \in get-return-edges x \longrightarrow x = a'}
\text{\text{by (fastforce elim:ex1E)}}
\text{let \text{\langle e' = (Node (sourcenode a'),kind a',Node (targetnode a'))}}
\text{have sourcenode a' \neq Entry}
\text{proof}
\text{\text{\langle assume sourcenode a' = Entry}}
\text{\text{\langle with (valid-edge a') (kind a' = Q'}\text{r}^\rightarrow_p \text{fs)}}
\text{\text{\langle show False by (rule Entry-no-call-source)}}
\text{qed}
\text{with (valid-edge a')}
\text{have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit ?e'}
\text{\text{\langle by (fastforce intro:lift-valid-edge.lve-edge)}}
\text{moreover}
\text{from (kind a' = Q'}\text{r}^\rightarrow_p \text{fs) have \text{\langle e' = Q'}\text{r}^\rightarrow_p \text{fs by simp}}
\text{moreover}
\text{from (e = (Node (sourcenode a), kind a, Node (targetnode a)))}
\text{\langle valid-edge a' \wedge a \in get-return-edges a'}}
\text{\text{\langle have e \in \text{get-return-edges get-return-edges valid-edge sourcenode targetnode kind e'}}
\text{\text{\langle by (fastforce intro:lift-get-return-edgesI)}}
\text{moreover}
\text{\{ fix x}
\text{\text{\langle assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit x}}
\text{\text{\langle and \exists \text{Q' r fs. kind x = Q'}\text{r}^\rightarrow_p \text{fs)}}
\text{\text{\langle and e \in \text{get-return-edges get-return-edges valid-edge)}}

40
from (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit x)  
\exists Q r fs. kind x = Q : r \hookrightarrow p fs \; obtain y \; where \; valid-edge y  
and x = (Node (sourcenode y), kind y, Node (targetnode y))  
by (fastforce elim: lift-valid-edge.cases)  
with (e \in lift-get-return-edges get-return-edges valid-edge  
sourcenode targetnode kind x) (valid-edge a)  
(e = (Node (sourcenode a), kind a, Node (targetnode a)))  
have x = ?e'  
proof (induct rule: lift-get-return-edges.induct)  
case (lift-get-return-edgesI ax ax' e)  
from (valid-edge ax) (ax' \in get-return-edges ax) have valid-edge ax'  
by (rule get-return-edges-valid)  
from (e = (Node (sourcenode ax'), kind ax', Node (targetnode ax')))  
(e = (Node (sourcenode a), kind a, Node (targetnode a)))  
have sourcenode a = sourcenode ax' and targetnode a = targetnode ax'  
by simp-all  
with (valid-edge a) (valid-edge ax') have [simp]; ax = ax' by (rule edge-det)  
from ax = (Node (sourcenode ax), kind ax, Node (targetnode ax))  
(\exists Q r fs, kind x = Q : r \hookrightarrow p fs \; have \exists Q r fs. kind ax = Q : r \hookrightarrow p fs by simp  
with (valid-edge ax) (ax' \in get-return-edges ax) imp  
have ax = a' by fastforce  
with (x = (Node (sourcenode ax), kind ax, Node (targetnode ax)))  
show ?thesis by simp  
qed  
ultimately show ?case by (blast intro:ex1I)  
qed simp-all  
next  
fix a a'  
assume a' \in 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 \exists a'', lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a'' \wedge  
src a'' = trg a \wedge trg a'' = src a' \wedge kind a'' = (\lambda cf. False) \checkmark  
proof (induct rule: lift-get-return-edges.induct)  
case (lift-get-return-edgesI ax ax' e')  
from (valid-edge ax) (ax' \in get-return-edges ax)  
obtain ax' where valid-edge ax' and sourcenode ax' = targetnode ax  
and targetnode ax' = sourcenode a' and kind ax' = (\lambda cf. False) \checkmark  
by (fastforce dest: intra-proc-additional-edge)  
let ?ex = (Node (sourcenode ax'), kind ax', Node (targetnode ax'))  
have targetnode ax \neq Entry  
proof  
assume targetnode ax = Entry  
with (valid-edge ax) show False by (rule Entry-target)  
qed  
with (?ex = (Node (sourcenode ax'), kind ax', Node (targetnode ax')))  
have (source-node ax' = targetnode ax) have source-node ax' \neq Entry by simp  
with (valid-edge ax')  
have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit ?ex
\[\begin{align*}
\text{by (fastforce intro:live-edge)} \\
\text{with } (e' = (\text{Node (sourcenode } a'), \text{kind } a', \text{Node (targetnode } a'))) \\
\quad (a = (\text{Node (sourcenode } ax), \text{kind } ax, \text{Node (targetnode } ax))) \\
\quad (e' = (\text{Node (sourcenode } a'), \text{kind } a', \text{Node (targetnode } a'))) \\
\quad (\text{valid-edge } ax') = (\text{targetnode } ax') = (\text{sourcenode } a') \\
\quad (\text{kind } ax') = (\lambda cf. \text{False}) \bot \\
\text{show } ?\text{case by simp} \\
\text{qed} \\
\end{align*}\]
case (lve-edge a e)
  from (e = (Node (sourcenode a), kind a, Node (targetnode a))) (knd e = Q: r→pfs)
  have kind a = Q: r→pfs by simp
  with (valid-edge a) have ∃!a'. valid-edge a' ∧ sourcenode a' = sourcenode a
  ∧ intra-kind(kind a') by (rule call-only-one-intra-edge)
then obtain a' where valid-edge a' ∧ sourcenode a' = sourcenode a
  and intra-kind(kind a')
  and imp: ∀ x. valid-edge x ∧ sourcenode x = sourcenode a ∧ intra-kind(kind x)
→ x = a' by (fastforce elim: ex1E)
let ?e' = (Node (sourcenode a'), kind a', Node (targetnode a'))
have sourcenode a ≠ Entry
proof
  assume sourcenode a = Entry
  with (valid-edge a) (kind a = Q: r→pfs) show False by (rule Entry-no-call-source)
qed
with (sourcenode a' = sourcenode a) have sourcenode a' ≠ Entry by simp
with (valid-edge a')
have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit ?e'
  by (fastforce intro: lift-valid-edge.lve-edge)
moreover
from (e = (Node (sourcenode a), kind a, Node (targetnode a)))
  (sourcenode a' = sourcenode a)
  have src ?e' = src e by simp
moreover
from (intra-kind(kind a')) have intra-kind (kind ?e') by simp
moreover
{ fix x
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit x
  and src x = src e and intra-kind (kind x)
from (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit x)
  have x = ?e'
proof (induct rule: lift-valid-edge.cases)
  case (lve-edge ax ex)
  from (intra-kind (kind x)) (x = ex) (src x = src e)
  (e = (Node (sourcenode ax), kind ax, Node (targetnode ax)))
  (e = (Node (sourcenode a), kind a, Node (targetnode a))):
  have intra-kind (kind ax) and sourcenode ax = sourcenode a by simp-all
  with (valid-edge ax) imp have ax = a' by fastforce
  with (x = ex) (ex = (Node (sourcenode ax), kind ax, Node (targetnode ax)))
  show ?case by simp
next
  case (lve-Entry-edge ex)
  with (src x = src e)
  (e = (Node (sourcenode a), kind a, Node (targetnode a)))
have False by simp
thus ?case by simp
next
case (lve-Exit-edge ex)
with ⟨src x = src e⟩
⟨e = (Node (sourcenode a), kind a, Node (targetnode a))⟩
have sourcenode a = Exit by simp
with ⟨valid-edge a⟩ have False by (rule Exit-source)
thus ?case by simp
next
case (lve-Entry-Exit-edge ex)
with ⟨src x = src e⟩
⟨e = (Node (sourcenode a), kind a, Node (targetnode a))⟩
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'←pf'
thus ∃!a'. lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a' ∧
try a' = try a ∧ intra-kind (knd a')
proof (induct rule:lift-valid-edge.induct)
case (lve-edge a e)
from ⟨e = (Node (sourcenode a), kind a, Node (targetnode a))⟩ ⟨knd e = Q'←pf'⟩
have kind a = Q'←pf' by simp
with ⟨valid-edge a⟩ have ∃!a'. valid-edge a' ∧ targetnode a' = targetnode a ∧
intra-kind(kind a') by (rule return-only-one-intra-edge)
then obtain a' where valid-edge a' and targetnode a' = targetnode a
and intra-kind(kind a')
and imp:∀ x. valid-edge x ∧ targetnode x = targetnode a ∧ intra-kind(kind x)
→ x = a' by (fastforce elim:ex1E)
let ?e' = (Node (sourcenode a'), kind a', Node (targetnode a'))
have targetnode a ≠ Exit
proof
assume targetnode a = Exit
with ⟨valid-edge a⟩ (kind a = Q'←pf') show False
by (rule Exit-no-return-target)
qed
with (targetnode a' = targetnode a) have targetnode a' ≠ Exit by simp
with ⟨valid-edge a'⟩ have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit ?e'
by (fastforce intro:lift-valid-edge.lve-edge)
moreover
from ⟨e = (Node (sourcenode a), kind a, Node (targetnode a))⟩
\(\langle \text{targetnode } a' = \text{targetnode } a \rangle\)

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

\(\{\text{fix } x\) assume lift-valid-edge \(\text{sourcenode } \text{targetnode } \text{kind } \text{Entry Exit } x\) and \(\text{trg } x = \text{trg } e\) and \(\text{intra-kind } (\text{knd } x)\)

from \(\langle \text{lift-valid-edge } \text{valid-edge } \text{sourcenode } \text{targetnode } \text{kind } \text{Entry Exit } x\)\)

have \(x = ?e'\)

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

\(\text{case } \langle \text{be-edge } ax \text{ ex} \rangle\)

from \(\langle \text{intra-kind } (\text{knd } x) \rangle\) \(\langle x = ex \rangle\) \(\langle \text{trg } x = \text{trg } e\rangle\)

\(\langle e = (\text{Node } (\text{sourcenode } ax), \text{kind } ax, \text{Node } (\text{targetnode } ax))\rangle\)

have \(\text{intra-kind } (\text{kind } ax)\) and \(\text{targetnode } ax = \text{targetnode } a\) by simp-all

with \(\text{valid-edge } ax\) imp have \(ax = a'\) by fastforce

\(\langle x = ex \rangle\) \(\langle ex = (\text{Node } (\text{sourcenode } ax), \text{kind } ax, \text{Node } (\text{targetnode } ax))\rangle\)

show ?case by simp

next

\(\text{case } \langle \text{be-Entry-edge } ex \rangle\)

with \(\langle \text{trg } x = \text{trg } e\rangle\)

\(\langle e = (\text{Node } (\text{sourcenode } ax), \text{kind } a, \text{Node } (\text{targetnode } ax))\rangle\)

have \(\text{targetnode } a = \text{Entry}\) by simp

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

thus ?case by simp

next

\(\text{case } \langle \text{be-Exit-edge } ex \rangle\)

with \(\langle \text{trg } x = \text{trg } e\rangle\)

\(\langle e = (\text{Node } (\text{sourcenode } ax), \text{kind } a, \text{Node } (\text{targetnode } ax))\rangle\)

have False by simp

thus ?case by simp

next

\(\text{case } \langle \text{be-Entry-Exit-edge } ex \rangle\)

with \(\langle \text{trg } x = \text{trg } e\rangle\)

\(\langle e = (\text{Node } (\text{sourcenode } ax), \text{kind } a, \text{Node } (\text{targetnode } ax))\rangle\)

have False by simp

thus ?case by simp

qed

ultimately show ?case by (blast intro: ex1I)

qed simp-all

next

fix \(a a' Q_1 r_1 p f s_1 Q_2 r_2 f s_2\)

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

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

and \(\text{knd } a = Q_1: r_1 \rightarrow p f s_1\) and \(\text{knd } a' = Q_2: r_2 \rightarrow p f s_2\)

then obtain \(x x'\) where \(\text{valid-edge } x\)

and \(a:a = (\text{Node } (\text{sourcenode } x), \text{kind } x, \text{Node } (\text{targetnode } x))\) and \(\text{valid-edge } x\)
x' 
and a' a' = (Node (sourcenode x'), kind x', Node (targetnode x'))
by(auto elim!:lift-valid-edge.cases)
with (kind a = Q1; r1 ↦ pfs1) (kind a' = Q2; r2 ↦ pfs2)
have (valid-edge x) (valid-edge x') have targetnode x = targetnode x'
bysimp-all
with (a a' show try a = try a' by simp)
next
from unique-callers show distinct-fst procs .
next
fix p ins outs
assume (p, ins, outs) ∈ set procs
from distinct-formal-ins[of this] show distinct ins .
next
fix p ins outs
assume (p, ins, outs) ∈ set procs
from distinct-formal-outs[of this] show distinct outs .
qed
qed

lemma lift-CFG-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 CFG-wf src trg knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewEntry
(lift-get-proc get-proc Main)
(lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind)
procs Main (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: CFG src trg knd
 lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit NewEntry
lift-get-proc get-proc Main
lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind
procs Main
by(fastforce intro: lift-CFG wf pd)
show ?thesis
proof
show 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 procs
thus length (lift-ParamUses ParamUses (src a)) = length ins

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

next
fix a assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
thus distinct (lift-ParamDefs ParamDefs (try a))

proof (induct rule: lift-valid-edge.induct)
  case (lce-edge a e)
    from ⟨valid-edge a⟩ have distinct (ParamDefs (targetnode a))
      by (rule distinct-ParamDefs)
    with ⟨e = (Node (sourcenode a), kind a, Node (targetnode a))⟩
    show ?case by simp
  qed

next
  case (lce-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'c→pfs' and (p, ins, outs) ∈ set procs
thus length (lift-ParamDefs ParamDefs (try a)) = length outs

proof (induct rule: lift-valid-edge.induct)
  case (lce-edge a e)
    from ⟨e = (Node (sourcenode a), kind a, Node (targetnode a))⟩ ⟨knd e = Q'c→pfs'⟩
    have kind a = Q'c→pfs' and try e = Node (targetnode a) by simp-all

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

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

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

next

fix \( n \) \( V \)

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

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

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

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

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

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

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

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

from \( \langle e = (\text{Node } (\text{source} \text{node } a), \text{kind } a, \text{Node } (\text{target} \text{node } a)) \rangle \) \( \langle \text{kind } e = Q:\text{r} \mapsto p f s \rangle \)

have \( \langle \text{kind } a = Q:\text{r} \mapsto p f s \rangle \) by simp

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

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

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

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

with \( V \in \text{Def } (\text{target} \text{node } a) \) show \( ?\text{case by (fastforce intro:lift-Def-node)} \)
  qed simp-all

next

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

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

and \( \text{knd } a = Q:\text{r} \mapsto p f s \) \( \langle p, \text{ins}, \text{outs} \rangle \in \text{set procs} \) and \( V \in \text{set ins} \)

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

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

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

from \( \langle e = (\text{Node } (\text{source} \text{node } a), \text{kind } a, \text{Node } (\text{target} \text{node } a)) \rangle \) \( \langle \text{kind } e = Q:\text{r} \mapsto p f s \rangle \)

have \( \langle \text{kind } a = Q:\text{r} \mapsto p f s \rangle \) by simp

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

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

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

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

with \( V \in \text{Def } (\text{target} \text{node } a) \) show \( ?\text{case by (fastforce intro:lift-Def-node)} \)
  qed simp-all

next

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

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

and \( \text{knd } a = Q:\text{r} \mapsto p f s \)
thus lift-Def Def Entry Exit H L (src a) = {}

proof (induct rule: lift-valid-edge.induct)
  case (lve-edge a e)
  show ?case
  proof (rule ccontr)
    assume lift-Def Def Entry Exit H L (src e) ≠ {}
    then obtain x where x ∈ lift-Def Def Entry Exit H L (src e) by blast
    from e = (Node (sourcenode a), kind a, Node (targetnode a))
    (kind e = Q:r→p)fs
    have kind a = Q:r→p by simp
    with (valid-edge a) have Def (sourcenode a) = {}
    by (rule call-source-Def-empty)
    have sourcenode a ≠ Entry
    proof
      assume sourcenode a = Entry
      with (valid-edge a) have kind a = Q:r→p by simp
      show False by (rule Entry-no-call-source)
    qed
    from e = (Node (sourcenode a), kind a, Node (targetnode a))
    have src e = Node (sourcenode a) by simp
    with (Def (sourcenode a) = {}): (x ∈ lift-Def Def Entry Exit H L (src e))
    (sourcenode a ≠ Entry)
    show False by (fastforce elim: lift-Def-set.cases)
  qed
  qed simp-all
next
fix n V
assume CFG.CFG.valid-node src try
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) n
and V ∈ ∪ set (lift-ParamUses ParamUses n)
hence ((n = NewEntry) ∨ n = NewExit) ∨ (∃ m. n = Node m ∧ valid-node m)
  by (auto elim: lift-valid-edge.cases simp: CFG.valid-node-def)
thus V ∈ lift-Use Use Entry Exit H L n apply –
proof (erule disjE)+
  assume n = NewEntry
  with (V ∈ ∪ set (lift-ParamUses ParamUses n)) show ?thesis by simp
next
assume n = NewExit
  with (V ∈ ∪ set (lift-ParamUses ParamUses n)) show ?thesis by simp
next
assume ∃ m. n = Node m ∧ valid-node m
then obtain m where n = Node m and valid-node m by blast
from (V ∈ ∪ set (lift-ParamUses ParamUses n)): (n = Node m)
have V ∈ ∪ set (ParamUses m) by simp
with (valid-node m) have V ∈ Use m by (rule ParamUses-in-Use)
  with (n = Node m) show ?thesis by (fastforce intro: lift-Use-node)
  qed
next
fix a Q p f ins outs V
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and knd a = Q→p f and (p, ins, outs) ∈ set procs and V ∈ set outs
thus V ∈ lift-Use Use Entry Exit H L (src a)
proof (induct rule:lift-valid-edge.induct)
  case (lee-edge a e)
    from (e = (Node (sourcenode a), kind a, Node (targetnode a))) (knd e = Q→p f)
    have knd a = Q→p f by simp
    from (valid-edge a) (kind a = Q→p f) (p, ins, outs) ∈ set procs (V ∈ set outs)
    have V ∈ Use (sourcenode a) by (rule outs-in-Use)
    from (e = (Node (sourcenode a), kind a, Node (targetnode a)))
    have src e = Node (sourcenode a) by simp
    with (V ∈ Use (sourcenode a)) show ?case by (fastforce intro:lift-Use-node)
qed simp-all
next
fix a V s
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and V ∈ lift-Def Def Entry Exit H L (src a) and intra-kind (knd a)
and pred (knd a) s
thus state-val (transfer (knd a) s) V = state-val s V
proof (induct rule:lift-valid-edge.induct)
  case (lee-edge a e)
    from (e = (Node (sourcenode a), kind a, Node (targetnode a)))
    (intra-kind (knd e)) (pred (knd e) s)
    have intra-kind (knd a) and pred (knd a) s
    and knd e = knd a and src e = Node (sourcenode a) by simp-all
    from (V ∈ lift-Def Def Entry Exit H L (src e)) (src e = Node (sourcenode a))
    have V ∈ Def (sourcenode a) by (auto dest: lift-Def-node)
    from (valid-edge a) (V ∈ Def (sourcenode a)) (intra-kind (knd a))
    (pred (knd a) s)
    have state-val (transfer (knd a) s) V = state-val s V
    by (rule CFG-intra-edge-no-Def-equal)
    with (knd e = knd a) show ?case by simp
next
case (lee-Entry-edge e)
from (e = (NewEntry, (λs. True) \ judgement, Node Entry)) (pred (knd e) s)
show ?case by (cases s) auto
next
case (lee-Exit-edge e)
from (e = (Node Exit, (λs. True) \ judgement, NewExit)) (pred (knd e) s)
show ?case by (cases s) auto
next
case (lee-Entry-Exit-edge e)
from (e = (NewEntry, (λs. False) \ judgement, NewExit)) (pred (knd e) s)
have False by (cases s) auto
thus ?case by simp
qed
next
  fix a s s'
  assume assms: lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit 
a
∀ V ∈ lift-Use Use Entry Exit H L (src a). state-val s V = state-val s' V 
intra-kind (knd a) pred (knd a) s pred (knd a) s'
  show ∀ V ∈ lift-Def Def Entry Exit H L (src a).
state-val (transfer (knd a) s) V = state-val (transfer (knd a) s') V
proof
  fix V assume V ∈ lift-Def Def Entry Exit H L (src a)
with assms
  show state-val (transfer (knd a) s) V = state-val (transfer (knd a) s') V
proof (induct rule: lift-valid-edge.induct)
  case (lve-edge a e)
  from (e = (Node (sourcenode a), kind a, Node (targetnode a)))(intra-kind (knd e)) have intra-kind (kind a) by simp
  show ?thesis by simp
  proof (cases Node (sourcenode a) = Node Entry)
    case True
    hence sourcenode a = Entry by simp
    from Entry-Exit-edge obtain a' where valid-edge a'
    and sourcenode a' = Entry and targetnode a' = Exit
    and kind a' = (λs. False)
    by blast
    have ∃ Q. kind a = (Q)
    proof (cases targetnode a = Exit)
      case True
      with (valid-edge a) (valid-edge a') (sourcenode a = Entry)
      (sourcenode a' = Entry) (targetnode a' = Exit)
      have a = a' by (fastforce dest: edge-det)
      with (kind a' = (λs. False)) show ?thesis by simp
next case False
with (valid-edge a) (valid-edge a') (sourcenode a = Entry)
(sourcenode a' = Entry) (targetnode a' = Exit)
(intra-kind (kind a)) (kind a' = (λs. False))
show ?thesis by (auto dest: deterministic simp: intra-kind-def)
qed
from True (V ∈ lift-Def Def Entry Exit H L (src e)) Entry-empty 
(e = (Node (sourcenode a), kind a, Node (targetnode a)))
have V ∈ H by (fastforce elim: lift-Def-set.cases)
from True (e = (Node (sourcenode a), kind a, Node (targetnode a)))(sourcenode a ≠ Entry ∨ targetnode a ≠ Exit)
have ∃ V ∈ H. V ∈ lift-Use Use Entry Exit H L (src e)
  by (fastforce intro: lift-Use-High)
with (∀ V ∈ lift-Use Use Entry Exit H L (src e).
state-val s V = state-val s' V; V ∈ H)
  have state-val s V = state-val s' V by simp
with (e = (Node (sourcenode a), kind a, Node (targetnode a)))(∃ Q. kind a = (Q)) (pred (knd e) s) (pred (knd e) s')

51
show ?thesis by (cases s, auto, cases s', auto)
next
case False
{ fix V' assume V' ∈ Use (sourcenode a)
  with (e = (Node (sourcenode a), kind a, Node (targetnode a))):
  have V' ∈ lift-Use Use Entry Exit H L (src e)
    by (fastforce intro: lift-Use-node)
}
with (∀ V ∈ lift-Use Use Entry Exit H L (src e).
  state-val s V = state-val s' V)
have (∀ V ∈ Use (sourcenode a). state-val s V = state-val s' V)
by fastforce
from (valid-edge a) this (pred (kind e) s) (pred (kind e) s')
(e = (Node (sourcenode a), kind a, Node (targetnode a))):
(intra-kind (kind e))
have (∀ V ∈ Def (sourcenode a). state-val (transfer (kind a) s) V =
  state-val (transfer (kind a) s') V)
by ~(erule CFG-intra-edge-transfer-uses-only-Use, auto)
from (V ∈ lift-Def Def Entry Exit H L (src e): False)
(e = (Node (sourcenode a), kind a, Node (targetnode a))):
have V ∈ Def (sourcenode a) by (fastforce elim: lift-Def-set.cases)
with (∀ V ∈ Def (sourcenode a). state-val (transfer (kind a) s) V =
  state-val (transfer (kind a) s') V)
(e = (Node (sourcenode a), kind a, Node (targetnode a))):
show ?thesis by simp
qed
next
case (lve-Entry-edge e)
from (V ∈ lift-Def Def Entry Exit H L (src e))
(e = (NewEntry, (λs. True), True, Node Entry))
have False by (fastforce elim: lift-Def-set.cases)
thus ?case by simp
next
case (lve-Exit-edge e)
from (V ∈ lift-Def Def Entry Exit H L (src e))
(e = (Node Exit, (λs. True), True, NewExit))
have False
by (fastforce elim: lift-Def-set.cases intro!: Entry-noteq-Exit simp: Exit-empty)
thus ?case by simp
next
case (lve-Entry-Exit-edge e)
thus ?case by (cases s) auto
qed
qed
next
fix a s s'
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and pred (kind a) s and snd (hd s) = snd (hd s')
and (∀ V ∈ lift-Use Use Entry Exit H L (src a). state-val s V = state-val s' V)

52
and \( \text{length} \ s = \text{length} \ s' \)

thus \( \text{pred} \ (\text{kind} \ a) \ s' \)

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

case (lee-edge \ a \ e)

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

|\( \text{pred} \ (\text{kind} \ e) \ s' \)|

have \( \text{pred} \ (\text{kind} \ a) \ s \ \text{and} \ \text{src} \ e = \text{Node} \ (\text{sourcenode} \ a) \) by simp-all

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

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

by (auto dest: lift-Use-node)

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

\( \text{snd} \ (\text{hd} \ s) = \text{snd} \ (\text{hd} \ s') \)

this \( \langle \text{length} \ s = \text{length} \ s' \rangle \)

have \( \text{pred} \ (\text{kind} \ e) \ s' \) by (rule CFG-edge-Uses-pred-equal)

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

show ?case by simp

next

case (lee-Entry-edge \ e)

thus ?case by (cases \( s' \)) auto

next

case (lee-Exit-edge \ e)

thus ?case by (cases \( s' \)) auto

next

case (lee-Entry-Exit-edge \ e)

thus ?case by (cases \( s \)) auto

qed

next

fix \( a \ Q \ r \ p \ fs \ ins \ outs \)

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

and \( \text{kind} \ a = Q: r \mapsto p \fs \ \text{and} \ (p, \ \text{ins}, \ \text{outs}) \in \text{set procs} \)

thus \( \text{length} \ fs = \text{length} \ ins \)

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

case (lee-edge \ a \ e)

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

\( \langle \text{kind} \ e = Q: r \mapsto p \fs \rangle \)

have \( \text{kind} \ a = Q: r \mapsto p \fs \) by simp

from \( \langle \text{valid-edge} \ a \ \text{kind} \ a = Q: r \mapsto p \fs \ \text{and} \ (p, \ \text{ins}, \ \text{outs}) \in \text{set procs} \rangle \)

show ?case by (rule CFG-call-edge-length)

qed simp-all

next

fix \( a \ Q \ r \ p \ fs \ a' \ Q' \ r' \ p' \ fs' \ s \ s' \)

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

and \( \text{kind} \ a = Q: r \mapsto p \fs \ \text{and} \ \text{kind} \ a' = Q': r' \mapsto p' \fs' \)

and lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit \( a' \)

and \( \text{src} \ a = \text{src} \ a' \ \text{and} \ \text{pred} \ (\text{kind} \ a) \ s \ \text{and} \ \text{pred} \ (\text{kind} \ a') \ s \)

from \( \langle \text{lift-valid-edge} \ \text{valid-edge} \ \text{sourcenode} \ \text{targetnode} \ \text{kind} \ \text{Entry} \ \text{Exit} \ a' \rangle \)

\( \langle \text{kind} \ a = Q: r \mapsto p \fs \ \text{and} \ \text{pred} \ (\text{kind} \ a) \ s \rangle \)

obtain \( x \) where \( a: a = (\text{Node} \ (\text{sourcenode} \ x), \text{kind} \ x, \text{Node} \ (\text{targetnode} \ x)) \)

and valid-edge \( x \) and \( \text{src} \ a = \text{Node} \ (\text{sourcenode} \ x) \)
and kind x = Q;→p;fs and pred (kind x) s
by (fastforce elim: lift-valid-edge_cases)
from lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a′
  ⟨kind a′ = Q;→p;fs′ ⟩ (pred (kind a′) s)
obtain x′ where a′:a′ = (Node (sourcenode x'), kind x', Node (targetnode x'))
  and valid-edge x′ and src a′ = Node (sourcenode x')
  and kind x′ = Q;→p;fs′ and pred (kind x') s
by (fastforce elim: lift-valid-edge_cases)
from (src a = Node (sourcenode x)) (src a′ = Node (sourcenode x'))
  ⟨src a = src a′ ⟩ have sourcenode x = sourcenode x′ by simp
from (valid-edge x) (kind x = Q;→p;fs) (valid-edge x′) (kind x′ = Q;→p;fs′)
  ⟨sourcenode x = sourcenode x′⟩ (pred (kind x) s) (pred (kind x′) s)
have x = x′ by (rule CFG-call-determ)
with a a′ show a = a′ by simp
next
fix a Q r p;fs i ins outs s s′
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and kind a = Q;→p;fs and i < length ins and (p, ins, outs) ∈ set procs
and pred (kind a) s and pred (kind a) s′
and ∀ V ∈ lift-ParamUses ParamUses (src a) ! i. state-val s V = state-val s′ V

thus params fs (state-val s) ! i = CFG.params fs (state-val s′) ! i
proof (induct rule: lift-valid-edge_induct)
case (lee-edge a e)
  from (e = (Node (sourcenode a), kind a, Node (targetnode a)))
    ⟨kind e = Q;→p;fs⟩
    ⟨pred (kind e) s⟩ (pred (kind e) s′)
  have kind a = Q;→p;fs and pred (kind a) s and pred (kind a) s′
    and src e = Node (sourcenode a)
    by simp_all
  from (∀ V ∈ lift-ParamUses ParamUses (src e) ! i. state-val s V = state-val s′ V)
    ⟨src e = Node (sourcenode a)⟩
    have ∀ V ∈ (ParamUses (sourcenode a))!i. state-val s V = state-val s′ V by simp
with ⟨valid-edge a⟩ ⟨kind a = Q;→p;fs⟩ (i < length ins)
    ⟨(p, ins, outs) ∈ set procs⟩ (pred (kind a) s) (pred (kind a) s′)
  show ?case by (rule CFG-call-edge_params)
qed simp_all
next
fix a Q′ p f′ ins outs cf cf′
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and kind a = Q′;→p;f′ and (p, ins, outs) ∈ set procs
thus f′ cf cf′ = cf′(lift-ParamDefs ParamDefs (try a) [:=] map cf outs)
proof (induct rule: lift-valid-edge_induct)
case (lee-edge a e)
  from (e = (Node (sourcenode a), kind a, Node (targetnode a)))
    ⟨kind e = Q′;→p;f′⟩
\begin{align*}
\text{have} & \quad \text{kind } a = Q' \leftarrow p_f' \quad \text{and} \quad \text{try } e = \text{Node} (\text{targetnode } a) \quad \text{by simp-all} \\
\text{from} & \quad (\text{valid-edge } a) \bowtie \text{kind } a = Q' \leftarrow p_f' \quad (p, \text{ ins, outs}) \in \text{set procs} \\
\text{have} & \quad f' \cdot \text{cf} = \text{cf'} (\text{ParamDefs} (\text{targetnode } a) := \text{map} \text{ cf outs}) \\
& \quad \text{by (rule CFG-return-edge-fun)} \\
\text{with} & \quad (\text{try } e = \text{Node} (\text{targetnode } a)) \quad \text{show} \ ?\text{case by simp} \\
\text{qed} & \quad \text{simp-all} \\
\text{next} & \\
\text{fix} & \quad a, a' \\
\text{assume} & \quad \text{lift-valid-edge valid-edge source node target node kind Entry Exit } a \\
& \quad \text{and} \quad \text{lift-valid-edge valid-edge source node target node kind Entry Exit } a' \\
& \quad \text{and} \quad \text{src } a = \text{src } a' \text{ and} \quad \text{try } a \neq \text{try } a' \\
& \quad \text{and} \quad \text{intra-kind } (\text{knd } a) \text{ and} \quad \text{intra-kind } (\text{knd } a') \\
\text{thus} & \quad \exists \ Q, Q' \quad \text{kind } a = (Q) \Rightarrow \text{knd } a' = (Q') \Rightarrow \wedge \\
& \quad \forall s. (Q s \rightarrow \neg Q s) \wedge (Q' s \rightarrow Q s) \\
\text{proof} & \quad \text{(induct rule: lift-valid-edge.induct)} \\
\text{case} & \quad (\text{lve-edge } a, e) \\
\text{from} & \quad (\text{lift-valid-edge valid-edge source node target node kind Entry Exit } a') \\
& \quad (\text{valid-edge } a) \bowtie (\text{source node } a, \text{kind } a, \text{Node} (\text{targetnode } a)) \\
& \quad (\text{src } e = \text{src } a' \cdot \text{try } e \neq \text{try } a') \cdot (\text{intra-kind } (\text{knd } a')) \cdot (\text{intra-kind } (\text{knd } a')) \\
\text{show} & \quad \text{?case} \\
\text{proof} & \quad \text{(induct rule: lift-valid-edge.induct)} \\
\text{case} & \quad (\text{lve-edge } a, e') \\
\text{from} & \quad (\text{source node } a, \text{kind } a, \text{Node} (\text{targetnode } a)) \\
& \quad (\text{src } e = \text{src } e') \\
\text{have} & \quad \text{source node } a = \text{Exit} \quad \text{by simp} \\
\text{with} & \quad (\text{valid-edge } a) \quad \text{have} \ False \quad \text{by (rule Exit-source)} \\
\text{thus} & \quad \text{?case by simp} \\
\text{qed} & \quad \text{auto} \\
\text{qed} & \quad (\text{fastforce elim: lift-valid-edge.cases})+ \\
\text{qed} & \\
\text{\textbf{lemma}} & \quad \text{lift-CFGExit:} \\
\text{assumes} & \quad \text{wf:CFGExit-wf source node target node kind valid-edge Entry get-proc} \\
& \quad \text{get-return-edges procs Main Exit Def Use ParamDefs ParamUses} \\
\text{and} & \quad \text{pd:Postdomination source node target node kind valid-edge Entry get-proc} \\
& \quad \text{get-return-edges procs Main Exit} \\
\text{shows} & \quad \text{CFGExit src try knd} \\
& \quad (\text{lift-valid-edge valid-edge source node target node kind Exit } a) \quad \text{NewEntry} \\
& \quad (\text{lift-get-proc get-proc Main}) \\
& \quad (\text{lift-get-return-edges get-return-edges valid-edge source node target node kind}) \\
& \quad \text{procs Main NewExit} \\
\text{proof} & \quad \text{(\textbf{interpret CFGExit-wf source node target node kind valid-edge Entry get-proc}} \\
& \quad \text{get-return-edges procs Main Exit Def Use ParamDefs ParamUses} \\
& \quad \text{by (rule wf)})
\end{align*}
interpret Postdomination sourcenode targetnode kind valid-edge Entry get-proc
get-return-edges procs Main Exit
by(rule pd)
interpret CFG:CFG src trg knd
lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit NewEntry
lift-get-proc get-proc Main
lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind
procs Main
by(fastforce intro:lift-CFG wf pd)
show ?thesis
proof
fix a assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and src a = NewExit
thus False by(fastforce elim:lift-valid-edge_cases)
next
show lift-get-proc get-proc Main NewExit = Main by simp
next
fix a Q p f
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and knd a = Q←↩p and trg a = NewExit
thus False by(fastforce elim:lift-valid-edge_cases)
next
show \exists a. lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a ∧
src a = NewEntry ∧ trg a = NewExit ∧ knd a = (\lambda s. False)√
by(fastforce intro:lve-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 sourcenode targetnode kind Entry Exit) NewEntry
(lift-get-proc get-proc Main)
(lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind)
procs Main NewExit (lift-Def Def Entry Exit H L) (lift-Use Use Entry Exit H L)
(lift-ParamDefs ParamDefs) (lift-ParamUses ParamUses)
proof –
interpret CFGExit-wf sourcenode targetnode kind valid-edge Entry get-proc
get-return-edges procs Main Exit Def Use ParamDefs ParamUses
by(rule wf)
interpret Postdomination sourcenode targetnode kind valid-edge Entry get-proc
get-return-edges procs Main Exit
by(rule pd)
interpret CFG-wf:CFG-wf src trg knd
lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit NewEntry
lift-get-proc get-proc Main
lift-get-return-edges get-return-edges valid-edge source-node target-node kind procs Main lift-Def Def Entry Exit H L lift-Use Use Entry Exit H L lift-ParamDefs ParamDefs lift-ParamUses ParamUses
by\(\text{(fastforce intro:lift-CFG-wf wf pd)}\)

interpret CFGExitCFGExit 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
by\(\text{(fastforce intro:lift-CFGExit wf pd)}\)

show ?thesis

proof

\begin{align*}
\text{show lift-Def Def Entry Exit H L NewExit} &= \{\} \\
\text{lift-Use Use Entry Exit H L NewExit} &= \{\}
\end{align*}

by\(\text{(fastforce elim:lift-Def-set.cases lift-Use-set.cases)}\)

qed

3.2.2 Lifting the SDG

lemma lift-Postdomination:
assumes wf:CFGExit-wf source-node target-node kind valid-edge Entry get-proc get-return-edges procs Main Exit Def Use ParamDefs ParamUses
and pd:Postdomination source-node target-node kind valid-edge Entry get-proc get-return-edges procs Main Exit
and inner:CFGExit.inner-node source-node target-node valid-edge Entry Exit nx
shows Postdomination 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

proof –

interpret CFGExit-wf source-node target-node kind valid-edge Entry get-proc get-return-edges procs Main Exit Def Use ParamDefs ParamUses
by\(\text{(rule wf)}\)

interpret Postdomination source-node target-node kind valid-edge Entry get-proc get-return-edges procs Main Exit
by\(\text{(rule pd)}\)

interpret CFGExitCFGExit 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
by\(\text{(fastforce intro:lift-CFGExit wf pd)}\)

\{ fix \emph{m} assume valid-node \emph{m} \\
then obtain \emph{a} where valid-edge \emph{a} and \emph{m} = source-node \emph{a} \lor \emph{m} = target-node \emph{a} \\
by\emph{(auto simp:valid-node-def)} \}

57
from \( \langle m = \text{sourcenode } a \lor m = \text{targetnode } a \rangle \) have CFG.CFG.valid-node src trg (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) (Node m)

proof
assume \( m = \text{sourcenode } a \)
show ?thesis
proof(cases \( m = \text{Entry} \))
case True
have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit (NewEntry,\( \lambda \cdot \text{True} \)) \( \backslash \text{by (fastforce intro: lve-Entry-edge)} \)
with \( m = \text{Entry} \) show ?thesis by (fastforce simp: CFGExit.valid-node-def)
next
case False
with \( m = \text{sourcenode } a \) \( \langle \text{valid-edge } a \rangle \)
have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit (Node (sourcenode a),\( \langle \text{kind } a, \text{Node (targetnode } a) \rangle \))
by (fastforce intro: lve-edge)
with \( m = \text{sourcenode } a \) show ?thesis by (fastforce simp: CFGExit.valid-node-def)
qed
next
assume \( m = \text{targetnode } a \)
show ?thesis
proof(cases \( m = \text{Exit} \))
case True
have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit (Node Exit,\( \lambda \cdot \text{True} \)) \( \backslash \text{by (fastforce intro: lve-Exit-edge)} \)
with \( m = \text{Exit} \) show ?thesis by (fastforce simp: CFGExit.valid-node-def)
next
case False
with \( m = \text{targetnode } a \) \( \langle \text{valid-edge } a \rangle \)
have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit (Node (sourcenode a),\( \langle \text{kind } a, \text{Node (targetnode } a) \rangle \))
by (fastforce intro: lve-edge)
with \( m = \text{targetnode } a \) show ?thesis by (fastforce simp: CFGExit.valid-node-def)
qed
qed

note lift-valid-node = this
{ fix \( n \) as \( n' \) cs m m' 
assume valid-path-aux cs as and \( m \rightarrow^* m' \) and \( \forall c \in \text{set } cs. \text{ valid-edge } c \) and \( m \neq \text{Entry} \lor m' \neq \text{Exit} \)
hence \( \exists cs' \) es. CFG.CFG.valid-path-aux knd (lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind) cs' es \( \land \)
list-all2 (\( \lambda c c'. c' = (\text{Node (sourcenode } c),\text{Kind } c,\text{Node (targetnode } c)) \)) cs cs'
\( \land \text{CFG.CFG.path src trg} \)
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) (Node m) es (Node m')
proof(induct arbitrary:m rule:vpa-induct)
case (vpa-empty cs)
from \( m \rightarrow^* m' \) have \([\text{simp}]; m = m' \) by fastforce

from \( m \rightarrow^* m' \) have valid-node \( m \) by\((\text{rule path-valid-node})\)

obtain \( cs' \) where \( cs' = \)
\( \map (\lambda c. (\text{Node (sourcenode c)}, \text{kind c}, \text{Node (targetnode c)})) cs \) by simp

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

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

next

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

note \( \text{IH} = \exists cs' \ es. \ \text{CFG.valid-path-aux knd} \)
\( \langle \text{lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind} \rangle \ cs' \ es \land \)
\( \text{list-all2} (\lambda c'. c' = (\text{Node (sourcenode c)}, \text{kind c}, \text{Node (targetnode c)})) cs \)
\( \langle \text{valid-node m} \rangle \ es (\text{Node m})' \)

from \( m - a \ # as \rightarrow^* m' \) have \( m = \text{sourcenode a} \ \text{and valid-edge a} \)
\( \text{and targetnode a} \ as \rightarrow^* m' \) by\(\text{(auto elim:path-split-Cons)}\)

show ?case

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

case True

with \( m = \text{sourcenode a} \ \langle m \neq \text{Entry} \lor m' \neq \text{Exit} \rangle \)
have \( m' \neq \text{Exit} \) by simp

from True have \( \text{targetnode a} = \text{Exit} \) by simp

with \( \langle \text{targetnode a} \ as \rightarrow^* m' \rangle \) have \( m' = \text{Exit} \)

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

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

have \( \text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit } ?e \)
by\(\text{fastforce intro:live-edge}\)

have \( \text{targetnode a} \neq \text{Entry} \)

proof

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

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

qed

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

from \( \text{IH} [\langle \text{targetnode a} \ as \rightarrow^* m' \rangle \ \langle c \in \text{set cs. valid-edge c} \ \text{this} \rangle] \)

obtain \( cs' \ es \)

where \( \text{valid-path:CFG.valid-path-aux knd} \)
(lift-get-return-edges get-return-edges valid-edge sourcenode
targetnode kind) cs cs'
and list:list-all2
(\x c'. c' = (Node (sourcenode c), kind c, Node (targetnode c))) cs cs'
and path:CFG.path src trg
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(Node (targetnode a)) cs cs' by blast
from (intra-kind (kind a)) valid-path have CFG.valid-path-aux knd
(lift-get-return-edges get-return-edges valid-edge sourcenode
targetnode kind) cs cs' (\x e es) by (fastforce simp: intra-kind-def)
moreover
from path (m = sourcenode a)
(lift-valid-edge valid-edge sourcenode 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 = (\forall m. \[m \in \set cs \rightarrow \forall c \in \set cs \rightarrow valid-edge c; m \neq Entry \lor m' \neq Exit\] \implies
\exists cs' es. CFG.valid-path-aux knd
(lift-get-return-edges get-return-edges valid-edge sourcenode
targetnode kind) cs cs' \land
list-all2 (\x c'. c' = (Node (sourcenode c), kind c, Node (targetnode c)))
(a \# cs) cs cs' \land CFG.path src trg
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(Node m) es (Node m')
from (m = a \# as \implies m') have m = sourcenode a and valid-edge a
and targetnode a = as \implies m' by (auto elim: path-split-Cons)
from \forall c \in \set cs. valid-edge c \lor valid-edge a
have \forall c \in \set cs. valid-edge c by simp
let ?e = (Node (sourcenode a), kind a, Node (targetnode a))
have sourcenode a \neq Entry
proof
  assume sourcenode a = Entry
  with (valid-edge a) (kind a = Q:r \implies p|fs)
  show False by (rule Entry-no-call-source)
qed
with (valid-edge a)
have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit ?e
  by (fastforce intro: lve-edge)
have targetnode a \neq Entry
proof
  assume targetnode a = Entry
  with (valid-edge a) show False by (rule Entry-target)
qed
hence targetnode a \neq Entry \lor m' \neq Exit by simp

60
from IH[OF ‹targetnode a ←−∗ 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
(λc c'. c' = (Node (sourcenode c), kind c, Node (targetnode c))) (a#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 list obtain cx esx where cs' = cx#csx
and cx: cx = (Node (sourcenode a), kind a, Node (targetnode a))
and list': list-all2
(λc c'. c' = (Node (sourcenode c), kind c, Node (targetnode c))) cs csx
by (fastforce simp: list-all2-Cons1)
from valid-path cx (cs' = cx#csx) (kind a = Q; r−−→ p; fs)
have CFG.valid-path-aux knd
(lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind) csx (??e#es) by simp
moreover from path (m = sourcenode a)
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit ?e)
have CFG.path src 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 list' by blast
next case (vpa-ReturnEmpty cs a as Q p f)
ote IH = (∃ m. (m ←−∗ m'; ∀ c ∈ set [], 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)))
[] 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 ←−∗ m' by (auto elim: path-split-Cons)
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) 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: live-edge)
have targetnode a ≠ Entry
proof
  assume targetnode a = Entry
  with ⟨valid-edge a⟩ show False by (rule Entry-target)
qed

hence targetnode a ≠ Entry ∨ m' ≠ Exit by simp

from IH[OF ⟨targetnode a = as→* m'⟩ - this] obtain es
  where valid-path:CFG.valid-path-aux kn
    (lift-get-return-edges get-return-edges valid-edge sourcenode
targetnode kind) es
  and path:CFG.path src try
    (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
    (Node (targetnode a)) es (Node m') by auto
from valid-path ⟨kind a = Q←p⟩
have CFG.valid-path-aux kn
  (lift-get-return-edges get-return-edges valid-edge sourcenode
targetnode kind) (?e#es) by simp
moreover
from path ⟨m = sourcenode a⟩
  ⟨lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit ?e⟩
have CFG.path src try
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
  (Node m) (?e#es) (Node m') by (fastforce intro:CFGExit.Cons-path)
ultimately show ?case using ⟨cs = []⟩ by blast
next
  case (vpa-ReturnCons cs a as Q p f c' cs')
  note IH = \( \forall m. m' - \# as→* m' \) \( \forall c \in \text{set } cs'. \text{ valid-edge } c; m \neq \text{Entry} \lor m' \)

  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 \( \forall c \in \text{set } cs. \text{ valid-edge } c \) \( \forall \# cs \) \( \forall c' \in \text{set } cs'. \text{ valid-edge } c' \) 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⟩ show False by (rule Exit-no-return-target)
  qed

proof
  assume targetnode a = Entry
  with ⟨valid-edge a⟩ show False by (rule Entry-target)
qed

with ⟨valid-edge a⟩ show False by (fastforce intro:lve-edge)
have targetnode a ≠ Entry
proof
assume \text{targetnode} a = \text{Entry}
\begin{align*}
\text{with} & \quad \langle \text{valid-edge} a \rangle \quad \text{show} \quad \text{False} \quad \text{by} (\text{rule} \quad \text{Entry-target}) \\
\text{qed} & \\
\text{hence} & \quad \text{targetnode} \ a \neq \text{Entry} \lor \ m' \neq \text{Exit} \quad \text{by} \quad \text{simp} \\
\text{from} & \quad \text{IH}[O\!F : \langle \text{targetnode} \ a \mathrel{-} \mathcal{as} \mathcal{-} \mathcal{\Rightarrow} \mathcal{m} \mathcal{'} \mathcal{\ni} \forall \mathcal{c} \mathcal{\in} \mathcal{cs} \mathcal{',} \quad \text{valid-edge} \ c \rangle \quad \text{this}] \\
\text{obtain} & \quad \text{csx es} \\
\text{where} & \quad \text{valid-path} : \quad \text{CFG}, \text{valid-path-aux} \ knd \\
& \quad \langle \text{lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind} \rangle \quad \text{csx es} \\
& \quad \text{and} \quad \text{list: list-all2} \\
& \quad \langle \lambda \mathcal{c}', \mathcal{c}' = (\text{Node} \ (\text{sourcenode} \ c), \text{kind} \ c, \text{Node} \ (\text{targetnode} \ c)) \rangle \quad \text{csx es} \\
& \quad \text{and} \quad \text{path: CFG.path src try} \\
& \quad \langle \text{lift-valid-edge valid-edge sourcecode targetnode kind Entry Exit} \rangle \\
& \quad \langle \text{Node} \ (\text{targetnode} \ a) \rangle \quad \text{es} \quad \langle \text{Node} \ m' \rangle \quad \text{by} \quad \text{blast} \\
\text{from} & \quad \langle \text{valid-edge} \ c' \rangle \quad \langle a \in \text{get-return-edges} \ c' \rangle \\
\text{have} & \quad ?e \in \text{lift-get-return-edges get-return-edges valid-edge sourcecode targetnode kind} \\
& \quad \langle \text{Node} \ (\text{sourcenode} \ c'), \text{kind} \ c', \text{Node} \ (\text{targetnode} \ c') \rangle \\
& \quad \text{by} (\text{fastforce intro: lift-get-return-edgesI}) \\
\text{with} & \quad \text{valid-path} \langle \langle \text{kind} \ a = \text{Q} \leftarrow \mathcal{p} \mathcal{f} \rangle \rangle \\
\text{have} & \quad \text{CFG.valid-path-aux} \ knd \\
& \quad \langle \text{lift-get-return-edges get-return-edges valid-edge sourcecode targetnode kind} \rangle \\
& \quad \langle \langle \text{Node} \ (\text{sourcenode} \ c'), \text{kind} \ c', \text{Node} \ (\text{targetnode} \ c') \rangle \# \text{csx} \rangle \quad \langle \langle ?e \# \text{es} \rangle \rangle \\
& \quad \text{by} \quad \text{simp} \\
\text{moreover} & \quad \text{from} \quad \text{list} \langle \langle \text{cs} = \mathcal{c'} \# \mathcal{cs} \rangle \rangle \\
\text{have} & \quad \text{list-all2} \\
& \quad \langle \lambda \mathcal{c}', \mathcal{c}' = (\text{Node} \ (\text{sourcenode} \ c), \text{kind} \ c, \text{Node} \ (\text{targetnode} \ c)) \rangle \quad \text{csx es} \\
& \quad \langle \langle \text{Node} \ (\text{sourcenode} \ c'), \text{kind} \ c', \text{Node} \ (\text{targetnode} \ c') \rangle \# \text{csx} \rangle \\
& \quad \text{by} \quad \text{simp} \\
\text{moreover} & \quad \text{from} \quad \text{path} \langle \langle \mathcal{m} = \text{sourcenode} \ a \rangle \rangle \\
& \quad \langle \langle \text{lift-valid-edge valid-edge sourcecode targetnode kind Entry Exit ?e} \rangle \rangle \\
\text{have} & \quad \text{CFG.path src try} \\
& \quad \langle \text{lift-valid-edge valid-edge sourcecode targetnode kind Entry Exit} \rangle \\
& \quad \langle \text{Node} \ m \rangle \quad \langle ?e \# \text{es} \rangle \quad \langle \text{Node} \ m' \rangle \quad \text{by} (\text{fastforce intro: CFGExit.Cons-path}) \\
\text{ultimately show} & \quad ?\text{case using} \langle \langle \text{kind} \ a = \text{Q} \leftarrow \mathcal{p} \mathcal{f} \rangle \rangle \quad \text{by} \quad \text{blast} \\
\text{qed} \} \\
\text{hence} & \quad \text{lift-valid-path}: \forall \mathcal{m} \mathcal{as} \mathcal{m}', \ [\mathcal{m} \mathcal{-} \mathcal{as} \mathcal{-} \mathcal{\Rightarrow} \mathcal{m}'; \ \mathcal{m} \neq \text{Entry} \lor \mathcal{m}' \neq \text{Exit}] \\
\Longrightarrow \exists \mathcal{es} \quad \text{CFG.CFG.valid-path'} \mathcal{src} \mathcal{try} \mathcal{knd} \\
& \quad \langle \text{lift-valid-edge valid-edge sourcecode targetnode kind Entry Exit} \rangle \\
& \quad \langle \text{lift-get-return-edges get-return-edges valid-edge sourcecode targetnode kind} \rangle \\
& \quad \langle \text{Node} \ m \rangle \quad \langle \text{Node} \ m' \rangle \\
& \quad \text{by} (\text{fastforce simp: vp-def valid-path-def CFGExit.vp-def CFGExit.valid-path-def}) \\
\text{show} & \quad ?\text{thesis} \\
\text{proof} & \\
\text{fix} \ n \quad \text{assume} \quad \text{CFG.CFG.valid-node} \mathcal{src} \mathcal{try} \\
& \quad \langle \text{lift-valid-edge valid-edge sourcecode targetnode kind Entry Exit} \rangle \ n \\
\text{hence} & \quad \langle \langle n = \text{NewEntry} \rangle \lor \langle n = \text{NewExit} \rangle \lor \langle \exists \mathcal{m} \mathcal{.} \ n = \text{Node} \ m \mathcal{\land valid-node} \ m \rangle \}
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 source-node target-node kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge source-node target-node kind)
NewEntry as n apply —
proof(erule disjE)+
assume n = NewEntry
hence CFG.CFG.valid-path\' src trg knd
(lift-valid-edge valid-edge source-node target-node kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge source-node target-node kind)
NewEntry [] n
by(fastforce intro: CFGExit.empty-path
simp: CFGExit vp-def CFGExit.valid-path-def)
thus ?thesis by blast
next
assume n = NewExit
have lift-valid-edge valid-edge source-node target-node kind Entry Exit
(NewEntry,(\lambda s. False),\rangle NewExit) by(fastforce intro: lve-Entry-Exit-edge)

hence CFG.CFG.path src trg
(lift-valid-edge valid-edge source-node target-node kind Entry Exit)
NewEntry [(NewEntry,(\lambda s. False),\rangle NewExit] NewExit
by(fastforce dest: CFGExit.path-edge)
with (n = NewExit). have CFG.CFG.valid-path\' src trg knd
(lift-valid-edge valid-edge source-node target-node kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge source-node target-node kind)
NewEntry [(NewEntry,(\lambda s. False),\rangle NewExit] n
by(fastforce simp: CFGExit vp-def CFGExit.valid-path-def)
thus ?thesis by blast
next
assume \exists m. n = Node m \land valid-node m
then obtain m where n = Node m and valid-node m by blast
from (valid-node m)
show ?thesis
proof(cases m rule: valid-node-cases)
case Entry
have lift-valid-edge valid-edge source-node target-node kind Entry Exit
(NewEntry,(\lambda s. True),\rangle Node Entry) by(fastforce intro: lve-Entry-edge)
with (m = Entry) (n = Node m) have CFG.CFG.path src trg
(lift-valid-edge valid-edge source-node target-node kind Entry Exit)
NewEntry [(NewEntry,(\lambda s. True),\rangle Node Entry] n
by(fastforce intro: CFGExit.Cons-path CFGExit.empty-path
simp: CFGExit.valid-node-def)
thus ?thesis by(fastforce simp: CFGExit vp-def CFGExit.valid-path-def)
next
case Exit
from inner obtain ax where valid-edge ax and intra-kind (kind ax)
and inner-node (source-node ax)
and target-node ax = Exit by(erule inner-node-Exit-edge)
hence lift-valid-edge valid-edge source-node target-node kind Entry Exit
(\text{Node (sourcenode ax),kind ax,Node Exit})
\text{by (auto intro:lift-valid-edge.lve-edge simp:inner-node-def})
\text{hence} \ CFG.\text{path src try}
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(\text{Node (sourcenode ax)}) [(\text{Node (sourcenode ax),kind ax,Node Exit})]
(\text{Node Exit})
\text{by (fastforce intro:CFGExit.Cons-path CFGExit.empty-path simp:CFGExit.valid-node-def})
\text{with (intra-kind (kind ax))}
\text{have slp-edge:CFG.CFG.same-level-path' src try knd}
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind)
(\text{Node (sourcenode ax)}) [(\text{Node (sourcenode ax),kind ax,Node Exit})]
(\text{Node Exit})
\text{by (fastforce simp:CFGExit.slp-def CFGExit.same-level-path-def intra-kind-def})
\text{have sourcenode ax} \neq \text{Exit}
\text{proof}
\text{assume sourcenode ax} = \text{Exit}
\text{with (valid-edge ax): show False by (rule Exit-source)}
\text{qed}
\text{have slp-edge:CFG.CFG.same-level-path' src try knd}
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
\text{by (fastforce intro:lve-Entry-edge})
\text{hence} \ CFG.\text{path src try}
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(\text{Node (sourcenode ax)}) [(\text{Node (sourcenode ax),kind ax,Node Exit})]
(\text{Node Exit})
\text{by (fastforce intro:CFGExit.Cons-path CFGExit.empty-path simp:CFGExit.valid-node-def})
\text{hence slp-edge':CFG.CFG.same-level-path' src try knd}
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind)
(\text{Node (sourcenode ax)}) [(\text{Node (sourcenode ax),kind ax,Node Exit})]
(\text{Node Exit})
\text{by (fastforce simp:CFGExit.slp-def CFGExit.same-level-path-def})
\text{from (inner-node (sourcenode ax)): have valid-node (sourcenode ax)}
\text{by (rule inner-is-valid)}
\text{then obtain asx where Entry \rightarrow asx \rightarrow \ast sourcenode ax}
\text{by (fastforce dest:Entry-path)}
\text{with (sourcenode ax} \neq \text{Exit}:
\text{have } \exists \text{es. CFG.CFG.valid-path' src try knd}
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind) (\text{Node Entry}) es (\text{Node (sourcenode ax)})
\text{by (fastforce intro:lift-valid-path)}
\text{then obtain es where CFG.CFG.valid-path' src try knd}
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind) (\text{Node Entry}) es (\text{Node (sourcenode ax)}) \text{by blast}
with slp-edge have \textit{CFG}.\textit{CFG}.\textit{valid-path}' \ src \ trg \ knd
\begin{itemize}
  \item \textit{CFG}.\textit{CFG}.\textit{valid-edge} valid-edge source-node target-node kind Entry Exit
  \item \textit{CFG}.\textit{CFG}.\textit{get-return-edges get-return-edges valid-edge source-node target-node kind}
  \item \textit{Node Entry} (es\@([\textit{Node (source-node ax), kind ax, \textit{Node Exit}]}) (\textit{Node Exit})
\end{itemize}
by \textit{rule \textit{CFGExit}.\textit{vp-slp-Append}}

with slp-edge' have \textit{CFG}.\textit{CFG}.\textit{valid-path}' \ src \ trg \ knd
\begin{itemize}
  \item \textit{CFG}.\textit{CFG}.\textit{valid-edge} valid-edge source-node target-node kind Entry Exit
  \item \textit{CFG}.\textit{CFG}.\textit{get-return-edges get-return-edges valid-edge source-node target-node kind}
  \item \textit{NewEntry} (es\@([\textit{Node (source-node ax), kind ax, \textit{Node Exit}]}) (\textit{Node Exit})
\end{itemize}
by \textit{rule \textit{CFGExit}.\textit{vp-slp-Append}}

with \(m = \text{Exit} \ (n = \text{Node m})\) \textbf{show} ?thesis by simp blast

next
case \textit{inner}

have \textit{lift-valid-edge} valid-edge source-node target-node kind Entry Exit
\(\textit{NewEntry},(\lambda s. \text{True}),\text{Node Entry}\) by \textit{fastforce intro lve-Entry-edge}

hence \textit{CFG}.\textit{path} src \ trg
\begin{itemize}
  \item \textit{lift-valid-edge} \textit{valid-edge} source-node target-node kind Entry Exit
  \item \textit{NewEntry} ([\textit{NewEntry},(\lambda s. \text{True}),\text{Node Entry}]) (\textit{Node Exit})
\end{itemize}
by \textit{fastforce intro \textit{CFGExit}.\textit{Cons-path \textit{CFGExit}.\textit{empty-path simp:CFGExit.\textit{valid-node-def}}}}

hence slp-edge: \textit{CFG}.\textit{CFG}.\textit{same-level-path}' \ src \ trg \ knd
\begin{itemize}
  \item \textit{CFG}.\textit{CFG}.\textit{valid-edge} \textit{source-node target-node kind Entry Exit}
  \item \textit{CFG}.\textit{CFG}.\textit{get-return-edges get-return-edges valid-edge source-node target-node kind}
  \item \textit{NewEntry} ([\textit{NewEntry},(\lambda s. \text{True}),\text{Node Entry}]) (\textit{Node Exit})
\end{itemize}
by \textit{fastforce simp CFGExit.slp-def CFGExit.same-level-path-def}

from \textit{valid-node m} obtain \textbf{as} where \textit{Entry – as \rightarrow \ast m}
by \textit{fastforce dest Entry-path}

with \(\textit{inner-node m}\)

have \(\exists es. \textit{CFG}.\textit{CFG}.\textit{valid-path}' \ src \ trg \ knd\)
\begin{itemize}
  \item \textit{CFG}.\textit{CFG}.\textit{valid-edge} \textit{source-node target-node kind Entry Exit}
  \item \textit{CFG}.\textit{CFG}.\textit{get-return-edges get-return-edges valid-edge source-node target-node kind}
  \item \textit{Node Entry} es (\textit{Node m})
\end{itemize}
by \textit{fastforce intro \textit{lift-valid-path simp:inner-node-def}}

then obtain \textit{es} \textbf{where} \textit{CFG}.\textit{CFG}.\textit{valid-path}' \ src \ trg \ knd
\begin{itemize}
  \item \textit{CFG}.\textit{CFG}.\textit{valid-edge} \textit{source-node target-node kind Entry Exit}
  \item \textit{CFG}.\textit{CFG}.\textit{get-return-edges get-return-edges valid-edge source-node target-node kind}
  \item \textit{Node Entry} es (\textit{Node m}) by blast
\end{itemize}

with slp-edge have \textit{CFG}.\textit{CFG}.\textit{valid-path}' \ src \ trg \ knd
\begin{itemize}
  \item \textit{CFG}.\textit{CFG}.\textit{valid-edge} \textit{source-node target-node kind Entry Exit}
  \item \textit{CFG}.\textit{CFG}.\textit{get-return-edges get-return-edges valid-edge source-node target-node kind}
  \item \textit{NewEntry} ([\textit{NewEntry},(\lambda s. \text{True}),\text{Node Entry}])@es
\end{itemize}
(Node m)
by \textit{rule \textit{CFGExit}.\textit{slp-Append}}

with \(\textit{n = Node m}\) show ?thesis by simp blast

qed
qed

next

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

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

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

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

thus \( \exists as. \text{CFG.\text{CFG}.valid-path'} \ src \ \try \ \knd \)

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

\( n \) as \text{NewExit} apply

proof(erule disjE)+

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

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

\((\text{NewEntry},(\lambda s. \text{False}) \land \text{NewExit})\) by(fastforce intro:lev-Entry-Exit-edge)

hence \( \text{CFG.\text{CFG}.path} \ src \ \try \)

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

\( \text{NewEntry} [(\text{NewEntry},(\lambda s. \text{False}) \land \text{NewExit})] \text{NewExit} \)

by(fastforce dest:CFGExit.path-edge)

with \( (n = \text{NewEntry}) \) have \( \text{CFG.\text{CFG}.valid-path'} \ src \ \try \ \knd \)

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

\( n [(\text{NewEntry},(\lambda s. \text{False}) \land \text{NewExit})] \text{NewExit} \)

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

thus \?thesis by blast

next

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

hence \( \text{CFG.\text{CFG}.valid-path'} \ src \ \try \ \knd \)

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

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

\( n [(\text{NewEntry},(\lambda s. \text{False}) \land \text{NewExit})] \text{NewExit} \)

by(fastforce simp:CFGExit.empty-path

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 \( m \) where \( n = \text{Node} m \) and \( \text{valid-node} m \) 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 \( ax = \text{Entry} \)

by(erule inner-node-Entry-edge)

hence lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit

\( (\text{Node} \text{Entry},\text{kind} ax,\text{Node} (\text{targetnode} ax)) \)

by(auto intro:lift-valid-edge.lev-edge simp:inner-node-def)

hence \( \text{CFG.\text{CFG}.path} \ src \ \try \)

(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(Node Entry) [(Node Entry,kind ax,Node (targetnode ax))]
(Node (targetnode ax))
by (fastforce intro: CFGExit.Cons-path CFGExit.empty-path
simp: CFGExit.valid-node-def)
with (intra-kind (kind ax))
have slp-edge: CFG.CG.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 az,Node (targetnode ax))]
(Node (targetnode ax))
by (fastforce simp: CFGExit.slp-def CFGExit.same-level-path-def
intra-kind-def)
have targetnode ax \neq Entry
proof
  assume targetnode az = Entry
  with (valid-edge az)
  show False by (rule Entry-target)
qed
have lift-valid-edge valid-edge source-node target-node kind Entry Exit
(Node Exit,(\lambda s. True),\gamma,NewExit) by (fastforce intro: lift-Exit-edge)

derby (fastforce intro: CFGExit.Cons-path CFGExit.empty-path
simp: CFGExit.valid-node-def)

hence slp-edge': CFG.CG.CG.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,(\lambda s. True),\gamma,NewExit)] NewExit
by (fastforce intro: CFGExit.Cons-path CFGExit.empty-path
simp: CFGExit.valid-node-def)

from (intra-node (target-node az)) have valid-node (target-node az)
by (rule inner-is-valid)
then obtain asx where target-node az \rightarrow asx \gamma Exit
by (fastforce dest: Exit-path)
with \langle target-node az \neq Entry \rangle
have \exists es. CFG.CG.CG.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 (target-node az)) es (Node Exit)
by (fastforce intro: lift-valid-path)
then obtain es where CFG.CG.CG.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 (target-node az)) es (Node Exit) by blast
with slp-edge have CFG.CG.CG.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-Append)  
with slp-edge' have CFG_CFG.valid-path' src try 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, (λs. True) ∨, NewExit)] NewExit  
by -(rule CFGExit.tp-slp-Append)  
with (m = Entry) (n = Node m) show ?thesis by simp blast  

next  

case Exit  

have lift-valid-edge valid-edge source-node target-node kind Entry Exit  
(Node Exit, (λs. True) ∨, NewExit) by (fastforce intro:rev-Exit-edge)  
with (m = Exit) (n = Node m) have CFG_CFG.path src try  
(lift-valid-edge valid-edge source-node target-node kind Entry Exit)  
(n [(Node Exit, (λs. True) ∨, NewExit)] NewExit)  
by (fastforce intro:CFGExit.Cons-path CFGExit.empty-path  
simp:CFGExit.valid-node-def)  
thus ?thesis by (fastforce simp:CFGExit.tp-def CFGExit.valid-path-def)  

next  

case inner  

have lift-valid-edge valid-edge source-node target-node kind Entry Exit  
(Node Exit, (λs. True) ∨, NewExit) by (fastforce intro:rev-Exit-edge)  

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

hence slp-edge:CFG_CFG.same-level-path' src try knd  
(lift-valid-edge valid-edge source-node target-node kind Entry Exit)  
(lift-get-return-edges get-return-edges valid-edge source-node target-node kind)  
(Node Exit) [(Node Exit, (λs. True) ∨, NewExit)] NewExit  
by (fastforce simp:CFGExit.slp-def CFGExit.same-level-path-def)  

from (valid-node m) obtain as where m → as → ∨ Exit  
by (fastforce dest:Exit-path)  

with (inner-node m)  

have ∃ es, CFG_CFG.valid-path' src try 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 try 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 try 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, (λs. True) ∪ NewExit)]]) NewExit
by −(rule CFG.Exit wp-slp-Append)
with (n = Node m) show ?thesis by simp blast
qed
qed
next
fix n n'
assume method-exit1:CFG.Exit.CFG.Exit.method-exit src knd
(lift-valid-edge valid-edge source node target node kind Entry Exit) NewExit n
and method-exit2:CFG.Exit.CFG.Exit.method-exit src knd
(lift-valid-edge valid-edge source node target node 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 (rule CFG.Exit.method-exit-cases)
  assume n = NewExit
  from method-exit2 show ?thesis
proof (rule CFG.Exit.method-exit-cases)
  assume n' = NewExit
  with (n = NewExit) show ?thesis by simp
next
fix a Q f p
assume n' = src a
  and (lift-valid-edge valid-edge source node target node kind Entry Exit) a
  and (knd a = Q ← p)
       hence lift-get-proc get-proc Main (src a) = p
by −(rule CFG.Exit.get-proc-return)
with CFG.Exit.get-proc-Exit lift-eq (n' = src a) (n = NewExit)
have p = Main by simp
with (knd a = Q ← p) have knd a = Q ← Main by simp
with (lift-valid-edge valid-edge source node target node kind Entry Exit) a
  have False by (rule CFG.Exit.Main-no-return-source)
  thus ?thesis by simp
qed
next
fix a Q f p
assume n = src a
  and (lift-valid-edge valid-edge source node target node kind Entry Exit) a
  and (knd a = Q ← p)
then obtain x where valid-edge x and src a = Node (source node x)
  and (knd x = Q ← p)
by (fastforce elim: lift-valid-edge cases)
       hence method-exit (source node x) by (fastforce simp: method-exit-def)
from method-exit2 show ?thesis
proof (rule CFG.Exit.method-exit-cases)
  assume n' = NewExit
  from (lift-valid-edge valid-edge source node target node kind Entry Exit) a
  ⟨knd a = Q ← p⟩
have lift-get-proc get-proc Main (src a) = p 
  by -(rule CFGExit.get-proc-return)
with CFGExit.get-proc-Exit lift-eq (n = src a) (n' = NewExit)
have p = Main by simp
with (knd a = Q'→p f') have knd a = Q'→Main f' by simp
with (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a)
have False by (rule CFGExit.Main-no-return-source)
thus ?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'→p' f'
then obtain x' where valid-edge x' and src a' = Node (sourcenode x')
  and kind x' = Q'→p' f'
  by (fastforce elim:lift-valid-edge.cases)
hence method-exit (sourcenode x') by (fastforce simp:method-exit-def)
with (method-exit (sourcenode x)) lift-eq (n = src a) (n' = src a')
  (src a = Node (sourcenode x)) (src a' = Node (sourcenode x'))
have sourcenode x = sourcenode x' by (fastforce intro:method-exit-unique)
with (src a = Node (sourcenode x)) (src a' = Node (sourcenode x'))
  (n = src a) (n' = src a')
show ?thesis by simp
qed
qed
qed

lemma lift-SDG:
assumes SDG:SDG sourcenode targetnode kind valid-edge Entry get-proc
get-return-edges procs Main Exit Def Use ParamDefs ParamUses
and inner;CFGExit.inner-node sourcenode targetnode kind valid-edge Entry Exit nx
shows SDG src try knd
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewEntry
  (lift-get-proc get-proc Main)
  (lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind)
  procs Main NewExit (lift-Def Def Entry Exit H L) (lift-Use Entry Exit H L)
  (lift-ParamDefs ParamDefs) (lift-ParamUses ParamUses)
proof –
interpret SDG sourcenode targetnode kind valid-edge Entry get-proc
get-return-edges procs Main Exit Def Use ParamDefs ParamUses
by(rule SDG)
have wf:CFGExit-wf sourcenode targetnode kind valid-edge Entry get-proc
get-return-edges procs Main Exit Def Use ParamDefs ParamUses
by(unfold-locales)
have pd:Postdomination sourcenode targetnode kind valid-edge Entry get-proc
get-return-edges procs Main Exit
by(unfold-locales)
interpret $\psi':\text{CFGExit-wf}$ src try knn
lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit NewEntry
lift-get-proc get-proc Main
lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind
procs Main NewExit lift-Def Def Entry Exit H L lift-Use Use Entry Exit H L
lift-ParamDefs ParamDefs lift-ParamUses ParamUses
by (fastforce intro: lift-CFGExit-wf wf pd)

interpret $\phi':\text{Postdomination}$ src try knn
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-Postdomination wf pd inner)

show ?thesis by unfold-locales
qed

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: lve $\equiv$ lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
and lget-proc: lget-proc $\equiv$ lift-get-proc get-proc Main
and lget-return-edges: lget-return-edges $\equiv$
lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind
and lDef: lDef $\equiv$ lift-Def Def Entry Exit H L
and lUse: lUse $\equiv$ lift-Use Use Entry Exit H L
and lParamDefs: lParamDefs $\equiv$ lift-ParamDefs ParamDefs
and lParamUses: lParamUses $\equiv$ 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 $\forall x$
and $H \cap L = \{\} \text{ and } H \cup L = \text{UNIV}$
shows NonInterferenceInterGraph src try knn 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 $\phi':\text{SDG}$ src try knn 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 $\lor$ try a = Node Entry

72
by (fastforce elim: lift-valid-edge.cases simp:lve)
next
show ∃ a. lve a ∧ src a = NewEntry ∧ trg a = Node Entry ∧ knd a = (λs. True)
  by (fastforce intro:lve-Entry-edge simp:lve)
next
fix a assume lve a and trg a = Node Entry
from lve a
have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
  by (simp add:lve)
from this (trg a = Node Entry)
show src a = NewEntry
proof (induct rule: lift-valid-edge.induct)
case (lve-edge a e)
  from ⟨e = (Node (sourcenode a), kind a, Node (targetnode a))⟩
  ⟨trg e = Node Entry⟩
  have targetnode a = Entry by simp
  with ⟨valid-edge a⟩ have False by (rule Entry-target)
  thus ?case by simp
qed simp-all
next
fix a assume lve a and trg a = NewExit
thus src a = NewEntry ∨ src a = Node Exit
  by (fastforce elim: lift-valid-edge.cases simp:lve)
next
show ∃ a. lve a ∧ src a = Node Exit ∧ trg a = NewExit ∧ knd a = (λs. True)
  by (fastforce intro:lve-Exit-edge simp:lve)
next
fix a assume lve a and src a = Node Exit
from lve a
have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
  by (simp add:lve)
from this ⟨src a = Node Exit⟩
show trg a = NewExit
proof (induct rule: lift-valid-edge.induct)
case (lve-edge a e)
  from ⟨e = (Node (sourcenode a), kind a, Node (targetnode a))⟩
  ⟨src e = Node Exit⟩
  have sourcenode a = Exit by simp
  with ⟨valid-edge a⟩ have False by (rule Exit-source)
  thus ?case by simp
qed simp-all
next
  from lDef show lDef (Node Entry) = H
  by (fastforce elim: lift-Def-set.cases intro:lift-Def-High)
next
from Entry-noteq-Exit lUse show lUse (Node Entry) = H
  by (fastforce elim: lift-Use-set.cases intro:lift-Use-High)
next
from Entry-noteq-Exit Use show Use (Node Exit) = L
by (fastforce elim:lift-Use-set.cases intro:lift-Use-Low)
next
from :H ∩ L = {} show H ∩ L = {}
next
from :H ∪ L = UNIV show H ∪ L = UNIV
qed
qed

end

References


